FAULT DETECTION AND RECTIFICATION ALGORITHMS

IN A QUESTION-ANSWERING SYSTEM

BY

ABDUL RAZAK HAMDAN

A Doctoral Thesis
submitted in partial fulfilment of the requirements
for the award of Doctor of Philosophy
of the Loughborough University of Technology
April, 1987
(Sha'ban,1407AH)

Supervisor: DR C.J.HINDE
Department of Computer Studies

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

IN THE NAME OF ALLAH
THE MERCIFUL THE COMPASSIONATE

وَقُلْ رَبِّ زِدْنِي عِلْمًا

O LORD, ADVANCE ME IN KNOWLEDGE

Khas untuk (Dedicated to):

Ayah, Bonda dan Bonda Mertua Ku

Zi

Nazrul, Aizat, ...

DECLARATION

I declare that I am responsible for the work submitted in this thesis, and that the original work is of my own except as specified in acknowledgements or in footnotes, and that neither the thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

Abdul Razak Hamdan

ACKNOWLEDGEMENT

First of all, my greatest and ultimate debt and gratitude are due to ALLAH, the almighty God. Without this will and great help this knowledge, which must be seen as very small in His presence, would never be existed.

Beyond these sentiments, I thank my supervisor, Dr C.J.HINDE, for his execellent guidance, advice amd willingness to assist me continuously throughout the programme of this work.

I acknowledge also, with gratitude, the Universiti Kebangsaan Malaysia and JPA, Malaysia for their finicial support to complete this project.

To my wife, I wish to express my endless gratitude for her patient and great support in every aspect in which she could help. I would like also to express my thanks to my children for their patience and also to my parents for bringing me up.

Many people have helped me throughout the duration of this research whom I wish to thank and whose names cannot be listed here.

May Allah bless all of them.

ABSTRACT

A Malay proverb "Jika sesat di hujung jalan, baleklah kepangkal jalan" roughly means "if you get lost at the end of the road, go back to the beginning". In going back to the beginning of the road, we learn our mistakes and hopefully will not repeat the same mistake again. Thus, this work investigates the use of formal logic as a practical tool for reasoning why we could not infer or deduce a correct answer from a question posed to a database.

An extension of the Prolog interpreter is written to mechanise a theorem proving system based on Horn clauses. This extension procedure will form the basis of the question-answering system Both input into and output from this system is in the form of predicate calculus. This system can answer all four classes of questions as classified by Chang & Lee [1973].

A natural language (a subset of English) interface which will be used in the question-answering system to enable the input and output in the form of that language is discussed especially in the context of using a reversible grammar. The reversible grammar which is based on Definite Clause Grammar rules is used both to analyse that language into predicate calculus and also to synthesize a sentence in that language from predicate calculus.

A technique to find out why we could not get the right answer to the question or enquiry posed to a data base is explored and explained. At first, a fault detection procedure which is an extension of the above question-answering system is written and discussed. Then a rectification procedure which will rectify the cause of failure in getting the right answer is investigated and reported here.

The system is capable of checking the non-existence of knowledge base clauses (factual or ruled), detecting the wrong references in the question and also making appropriate suggestions. In addition, the system can also find the set of conditions in order for certain rules to be true

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 The Problem

Logic programming which began in the early 1970's originated largely from advances in automatic theorem proving and artificial intelligence, and in particular from the development of the resolution principle (Robinson[1965]). Constructing automated deduction systems is, of course, central to the aim of achieving artificial intelligence.

The key idea underlying logic programming is programming by description. The programmer describes the application area and lets the program choose specific operations. Logic programs are easier to create and enable machines to explain their results and actions.

One of the main ideas of logic programming , which is due to Kowalski [1979] and [1979a], is that an algorithm consists of two disjoint components, the logic and the control. In a typical logic programming system, we can also view that the description of the control as an application-independent deductive inference procedure. Applying such a procedure to a description of an application area makes it possible for a machine to draw conclusions about the application area and to answer questions even though these answers are not explicitly recorded in the description. This capability is the basis for the technology of logic programming.

Departing from Kowalski's point of view about algorithms, we aim to write a theorem prover based on Prolog to include such controls as looping control which Prolog lacks, thus enabling the users to concentrate on the logic of their algorithms or programs.

Logic also can be viewed as having three interpretations (Lloyd[1984]). These interpretations are:

(a). Procedural interpretation.

(b). Database interpretation.

(c). Process interpretation.

The second interpretation means that a logic program is regarded as a database thus we obtain very natural way and powerful generalisation of relational databases. Databases may also contain some rules to describe the information in them. Thus we aim to write a question-answering system to deal with such databases. However, we may be frustrated with the unexpectedly unsuccessful query we have attempted. It may be due to the non-existent fact or the lack of knowledge. The other reason is that we may ask the wrong question such that we cannot get the expected or required result. So we attempt to rectify this matter. To rectify this matter, we divide the program into two main algorithms:

[1] Fault detection algorithm.

[2] Fault rectification algorithm.

The first algorithm is to find the reasons and the second one will rectify them. The main idea of both algorithms is stated in a Malay proverb which says "Jika sesat di hujung jalan, baleklah ke pangkal jalan". The Malay proverb roughly means "if you get lost at the end of the road, go back to the beginning". During the process of going back, we learn all our mistakes and rectify them accordingly and hopefully we will not fall into the same traps again.

We also attempt to incorporate a subset of English grammar based on DCGS (Pereira and Warren[1980]) into the theorem prover and the detecting and rectifying fault algorithms.

1.2 Results

We have written a Prolog-based theorem prover with a looping test which will terminate in any proving except in the case of occur check. All the knowledge clauses are in the form of Horn clauses after converting from PC. Following this termination of the theorem prover, and by using the backtracking techniques as implemented in Prolog, we are able to detect faults which occurred at different levels of any unexpectedly unsuccessful proving. The main idea of the detecting and rectifying fault algorithm is as follows:

(1) read Q, the question to be proved.
(2) convert the negation of Q into Horn clauses.
(3) deduce an empty clause from each resulting Horn clauses
by assuming any failed subgoal is true
and record all failed subgoal.
(4) Rectify each set of failed subgoals (reason clauses).

Algorithm 1: The detecting and rectifying fault algorithms

Steps (3) and (4) are actually fault detection and fault rectification respectively. Here, the fault detection algorithm is blended together with the theorem prover. The fault detected is a type of non-existent clause or fact. As said above that the fault detection algorithm can isolate the fault at different levels. For example:

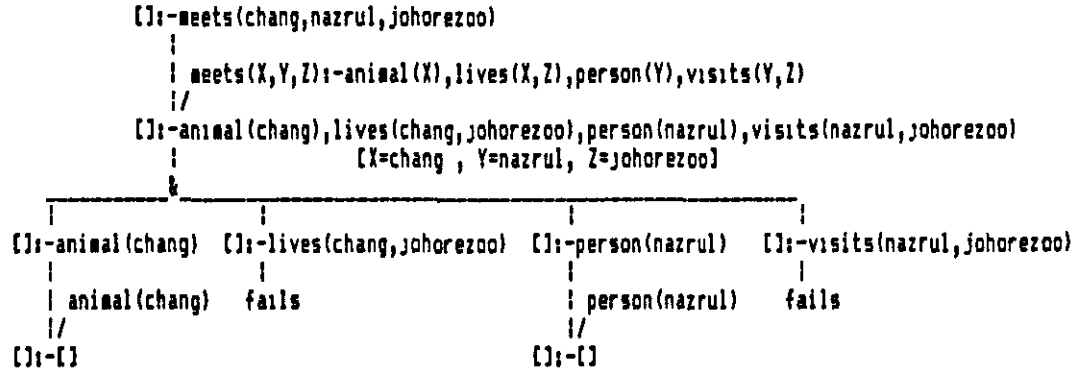


Fig 1.2.1: The proving tree of "meets(chang,nazrul,johorezoo)"

Referring to Fig 1.2.1 above and by using ordinary Prolog's backtracking, we can only detect one non-existent clause or fact, i.e *"visits(nazrul,johorezoo)"*. But by using the fault detection algorithm, both non-existent clauses or facts, i.e *"lives(chang,johorezoo)"* and *"visits(nazrul,johorezoo)"*, can be detected where both of them are members of the same set of reason clauses. All members of the same set of reason clauses must be true in order the goal to be true, i.e in this case, both reason clauses must be true in order the query (goal) *"meets(chang,nazrul,johorezoo)"* to be true.

The fault rectification algorithm assumes at first that the query contains wrong references and if that is not the case, then it assumes that the database lacks knowledge. In the first case, the matching is carried out between the reason clause and the knowledge base (KB) clauses. The following are some of the examples of the suggestions made by the rectification algorithm:

(a) if "johorezoo" of the question clause "meets(chang,nazrul,johorezoo)" is substituted with "londonzoo"

(b) if "meets" of the question clause "meets(chang,nazrul,johorezoo)" is substituted with "encounters"

(c) if the following clauses are true:

```
lives(chang,johorezoo)
visits(nazrul,johorezoo)
```

(d) if "all(_1,...)" is replaced with "exists(_1,...)"

(e) if "every man" is replaced with "a man".

The system is also able to accept input in both PC and a subset of English sentence. The example (e) above is produced when the input is in the form of an English sentence. The type of the answer of the query will depend on the type of the input, i.e either PC or English sentence.

1.3 Related work

1.3.1 Logic programming and automatic theorem prover

Logic programming which has begun in the early 1970's originated largely from advances in automatic theorem proving and artificial intelligence, and in particular from the development of the resolution principle(Robinson[1965]). The key idea underlying logic programming is programming by description. Logic programming differs fundamentally from conventional programming in requiring us to describe the logical structure of problem rather than making us prescribe how the computer is to go about solving them.

Building on work of Herbrand[1930], there was much activity in theorem proving in early 1960's by Prawitz[1960], Gilmore[1960], Davis and Putnam[1960] and others. This effort culminated in 1965 with the publication of the landmark paper by Robinson [1965], which introduced an inference rule called the resolution principle which is particularly well-suited to automation on a computer. In 1972, Kowalski and Colmerauer were led to the fundamental idea that logic can be used as a programming language. Before that (1972), logic had only ever been used as a specification or declarative language in computer science. However, what Kowalski[1974] showed is that logic has a procedural interpretation, which makes it very effective as a programming language.

One of the main ideas of logic programming , which is due to Kowalski [1979] and [1979a], is that an algorithm consists of two disjoint components, the logic and the control, i.e

"*Logic + control = Algorithm*". Thus, ideally, the programmer should only have to specify the logic component of an algorithm and the control should be exercised solely by the logic programming system. Unfortunately, this ideal has not yet been achieved with current logic programming systems (Lloyd[1984]). In order for this to be achieved there are two broad problem which have to be solved, i.e control problem (Genesereth et al.[1983], Lloyd[1984], Genesereth and Ginsberg[1985]) and negation problem (Clark[1978], Reiter[1978] and Shepherdson[1984]).

Apart from the procedural interpretation (Kowalski[1974]), logic also has two other interpretations, i.e Database interpretation (Lloyd[1983]; Gallaire and Minker[1978] and [1981]; Gallaire et al. [1984]) and process interpretation (Clark and Gregory[1981] and [1983]; Shapiro[1983] and Shapiro and Takeuchi[1983]).

Most logic programming system use clausal form. However, logic programming is by no means limited to PROLOG which is based on Horn clauses. Various non-clausal resolution have been developed, for example, Storm [1974], Wilkins [1974], Bibel [1976], Nilsson [1979], Manna and Waldinger[1980], Bowen [1982], Murray [1982] and Stickel[1982]. Other methods of non-resolution theorem-proving are such as natural deduction (Bledsoe [1977], Hanson et al.[1982], and, Haridi and Sahlin[1983]) and matrices and connections (Prawitz [1976], Andrews [1981] and Bibel[1983]).

It is clear that logic thus provides a single formalism for apparently diverse parts of computer science. This range of

applications assures that logical inference is about to become the fundamental unit of computation (Lloyd[1984]). This view is strongly supported by the Japanese fifth generation computer project where logic programming has been chosen to provide the core programming language for this very ambitious 10 years project (Moto-Oka[1982]).

1.3.2 Debugging

Computer software is also very costly in terms of money and labour. Boehm[1976], Brooks[1975], Myers[1978], and Yourdan and Constantine[1979] indicate that testing and debugging alone represent approximately half the cost of new system development. As error detection and error correction are now considered to be the major cost factors in software development, it is worth spending effort to make sure that the programs we are writing are going to work. The area of computer program debugging is also one of the key phases in the system software cycle.

Many mathematical models such as the J-M model (Jelinski and Moranda[1972]), the probabilistic model (Shooman [1972]), the execution-time theory model (Musa[1979]), Fault Removal model (Littlewood [1981]) etc (see Sallih [1986] for a review of the said models and others) have been developed to describe the behaviour of software package errors and then to get some measures from which the reliability of these software packages can be calculated.

Software reliability is important as hardware reliability as the increasing usage of computer systems especially in very critical fields such as air traffic control etc. For

example, in 1960 the US defence system software (NORADS) had wrongly identified the rising moon as a rocket from the USSR. There is a difference between these two reliability as Littlewood[1980], says "a hardware device is certain to fail eventually, whereas a program if perfect is certain to remain failure free".

It has also been suggested that one way to eliminate the need for debugging is to provide a correctness proof of the program. Naur and Randell [1969] suggested that we can dispense with testing altogether when we have given the proof of correctness of the program. But, Goodenough and Gehart[1975] found seven bugs in a simple text formatter program described and informally proved by Naur [1969]. So, the informal or formal proofs of program correctness do not guarantee that the program is correct.

However, as pointed out by Goodenough and Gehart [1975], that the practise of proving program correctness is useful for improving reliability, but suffers from the same types of errors as programming and testing, namely, failure to find and validate all special cases relevant to a design, its specification, the program and its proof. Gries [1981] also agreed that even though we can become more proficient in programming, we will still make errors, even if only of a syntactic nature. Hence some testing will always be necessary. But, he does not refer to the testing process as debugging, and suggests that the test is to increase our confidence in a program we are quite sure is correct; finding an error should be the exception rather than the rule.

The area of debugging crucial to software development and maintenance is semantic debugging. Syntactic errors are defined for the purposes of computer programming as errors that compilers recognise, and the use of high level programming with a strong-typing mechanism, such as Pascal, Algol-like languages will help toward finding syntactic errors. The idea behind PASCAL and ALGOL is to move semantic errors more and more into the syntactic areas, many errors not spotted by FORTRAN would appear as syntactic errors in ALGOL. Semantic errors are those that compiler cannot recognise and the adoption of structured programming techniques will help a little bit in removing such errors. Vessey [1986] view debugging from either a process viewpoint or a functional viewpoint. She also studied the relation between novice and expert programmers.

Adam and Laurent [1980] discussed a debugging system called LAURA which have been designed to detect or localize the errors it may contain. Shapiro[1982] also tried to lay theoretical foundations for program debugging, with the goal of partly mechanising this activity. In particular, Shapiro attempted to formalise and develop algorithmic solutions to the following two questions·

(1) How do we identify a bug in a program that behaves incorrectly?
(2) How do we fix a bug, once one is identified?

The algorithms Shapiro developed are interactive, as they rely on the availability of answers to such queries. He integrated both diagnosis and bug-correction algorithms into a debugging algorithm. A debugging algorithm accepts as input a program (or empty one) to be debugged and a list of input/output samples, which partly define the behaviour of the target program.

Rule learning techniques can also be considered as debugging program techniques. The task tackled by rule learning techniques is to modify a set of rules of the form hypothesis implies conclusion. This set of rules can be considered as a program especially written in Prolog clauses. The basic rule learning technique is as follows:

```
Until the rules are satisfactory:
1. Identify a fault with a rule
2. Modify the rule to remove the fault.
```

Bundy et al.[1985] give an excellent review and comparison of rule learning techniques. Bundy et al[1985] also classify these faults as factual or control one. Most identifying faults techniques are by comparing the ideal trace (graph) with a learning (program) trace (graph) (for eg. Bradzil[1981]). Shapiro's techniques as briefly described above is also one of the technique to identify a fault. A lot of techniques are used to modify the faulty rules such as reordering them (eg. Bradzil[1981]), adding extra condition(s) to them (eg. Bradzil [1981], Waterman [1970]), instantiating them (eg. Bradzil [1981], Shapiro [1982]), updating them (eg. Waterman[1970], Mitchell et al.[1981] and [1983]) or asking a ground oracle to the user (Shapiro[1982]).

In proving a theorem or making an enquiry to the database, we may also get negative answer due to some faults. Bourne[1977] examined the frequency of spelling errors in a sample drawn from 11 machine-readable bibliographic databases and concluded that errors are not only in the input queries, but also in the database itself.

1.4 Outline of the thesis

The target and the implementation language for the algorithms and systems developed in this thesis is Prolog. Basic concepts of predicate calculus, other logics and knowledge representations, logic programming and Prolog are discussed in Chapter 2. Resolution concepts and the control strategies for the implementation of the resolution are also discussed in Chapter 2.

In Chapter 3, we develop a theorem prover based on Prolog. The Prolog-based theorem prover will accept an input in the form of PC which will subsequently be converted into Horn clauses. All the conversion processes and the implementation are described in this chapter. The theorem provers are implemented and two search strategies are compared, i.e a depth-first method and a breadth-first method.

Various techniques based on Definite Clause Grammars are studied and compared in Chapter 4 for analysing and synthesizing an English sentence from and to PC. This subset of English grammar then is interfaced into the theorem prover developed in previous Chapter 3.

The fault detection and rectification algorithms which are developed by extending the theorem prover implemented using a depth-first method are discussed in Chapter 5 and 6 respectively. Some examples are also given in Chapter 6.

In Chapter 7, a subset of English grammar discussed in Chapter 4 is then interfaced with the fault detection and

rectification algorithms. Comments on the implementation of various techniques discussed in Chapter 4 and some examples are given in this chapter.

A discussion of the thesis and suggestions of further work are given in the last chapter, i.e Chapter 8.

CHAPTER 2

FORMAL LOGIC AND LOGIC PROGRAMMING

2.1 A Predicate Calculus

In order to solve the complex problems encountered in artificial intelligence (AI), one needs both a large amount of knowledge and some mechanism for manipulating that knowledge to create a solution to a new problems. A variety of ways of representing knowledge (facts) have been exploited in AI's problem solvers. Specific knowledge representation models allow for more specific, more powerful inference mechanisms that operate on them.

In many applications, the information to be encoded into the global database of a production system originates from descriptive statements that are difficult or unnatural to represent by simple structures like arrays or sets of numbers. Intelligent information retrievel, robot problem solving, and mathematical theorem proving, for example, require the capability for representing, retrieving and manipulating sets of statements.

One particular way of representing facts or knowledge is the language of logic. Logic is a way of representing various statements (propositions) about the world so that it would be possible to formally check whether these representations were valid or not.

The logical formalism is appealing because it immediately suggests a powerful way of deriving new knowledge from old- i.e mathematical deduction. In this formalism, we can conclude that a new statement is true by proving that it follows from the statements that are already known. Thus the

idea of proof can be extended to conclude a deduction, a way of deriving answers to questions and solutions to problems.

In this project, a first order calculus (logic) is adopted. The first order predicate calculus is a formal language in which a wide variety of statements can be expressed.

Predicate calculus (PC) is a branch of symbolic logic, and is designed to express various statements about the world.

Now, let us define what is a PC. A PC (Predicate Calculus and not Personal Computer) is a language and it is defined by its *syntax*. To specify a syntax we must specify the alphabet of symbols to be used in the language and how these symbols are to be put together to form legitimate expressions in the language. The legitimate expressions of the PC are called the *well-formed formulas* (wffs).

2.1.1 The syntax and semantics of atomic formulas

The elementary components of the PC language are constant symbols, function symbols, variable symbols and predicate symbols set off by brackets, parentheses, and commas. For example, the meaning of sentences is a proposition which consists of terms, which are of two types, predicate and arguments. The predicates are the relation names and usually correspond to verbs in sentences and arguments are the objects that are related and usually correspond to nouns. In the sentence below:

Aizat likes Nazrul

we have a relation (predicate) express in "likes" and two objects (arguments) express in "Aizat" and "Nazrul". Thus

"likes(Aizat,Nazrul)" could be a simple atomic formula for the above sentence. In the above example sentence, "Aizat" and "Nazrul" are constants which indicate a particular individual or class of individuals. By using variables, it may be possible to expand the representation to stand for different individuals at different times, where the individuals or its class remains unspecified, for instance, "likes(X,Y)" where X and Y are variables.

In general, atomic formulas are composed of predicate symbols and terms. All constants and variables symbols are terms. The function which its arguments are terms, is also a term.

In the PC, a wff can be given an interpretation by assigning a correspondence between the elements of the language and the relations, entities, and functions in the domain of discourse. To each predicate symbol, we must assign a corresponding relation in the function symbol, a function in the domain. These assignments define the semantics of the PC language. Once an interpretation from an atomic formula has been defined, we say the formula has value T (true) just when the corresponding statement about the domain is true and that is has value F (false) just when the corresponding statement is false. So, all PC logic must have either the value "true"(T) or "false"(F).

2.1.2 Connectives

Atomic formulas are merely the elementary building blocks of the PC language. We can combine atomic formulas to form more complex wffs by using connectives such as "$\wedge$" *(and)*, "$\vee$"*(or)*, "*--)*"*(implies)*, "*(--)*"*(equivalent)* and "~"*(not)*. Although "~" is called a connective, it is really not used to connect two formulas. It is used to negate the truth value of a formula from "true" to "false", and vice-versa. A formula with "~" in front of it is called a *negation*. Formulas built by connecting other formula by "and" and "or" signs are called *conjunctions* and *disjunctions* respectively.

A formula built by connecting two formulas with an "implies" sign is called an *implication* and it usually represents "if-then" statements. The left-hand and right-hand sides of an implication are called *antecedent* (or *condition*) and *consequent* (or *conclusion*) respectively. If two formulas are connected by "equivalent to" sign then it is called an *equivalence*.

Any conjunctions or disjunctions composed of wffs, and the negation of a wff are also wffs. An implication and equivalence are also wffs if both the antecedent and the consequent are wffs.

The following are the truth table for the described connectives above:

A	B	~A	A $\wedge$ B	A $\vee$ B	A--)B	A(--)B
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Table 2.1: The truth tables

It can be seen from the above truth table that an implication has a value T if either the consequent has value T (regardless of the value of the antecedent) or if the antecedent has value F (regardless of the value of the consequent); otherwise the implication has value F. This definition of implicational truth value is sometimes at odds with our intuitive notion of the meaning of "implies". For example, the predicate calculus representation of the sentence "If the sun is made of strawberry, then horse can fly" has value T. The truth values of an implication is equivalent to "~A $\vee$ B". Furthermore, "A<-->B" is equivalent to "(A-->B) $\wedge$ (B-->A)".

An atomic formula and the negation of an atomic formula are both called literals. If the PC does not contain any variables, then the PC is called propositional calculus. For example, the sentence "he is sick, so he needs a doctor" can be written in propositional calculus as "p-->q" where p="he is sick" and q="he needs a doctor".

A formula which is true under all its interpretations is called a *tautology* (or a *valid formula*). A formula is invalid iff it is not valid, i.e there exists an interpretation under which it is false. A formula which is false under all its interpretation is called a *contradiction* (or *unsatisfiable* or *inconsistent*). A formula is *consistent* (*satisfiable*) iff it is not inconsistent (unsatisfiable), i.e there exists an interpretation under which it is true. A formula F is said to be a *logical consequence* of a formula, or set of formulas, S, if F is satisfied by all interpretations which satisfy S.

2.1.3 Quantification.

To be able to account for modifiers such as brave in "brave men", some notion of "who is", "who are", "which is" etc. are needed. Moreover distinctions need to be made between "the brave men", "brave men" or "the brave man" and " a brave man". Quantifiers are used in PC to process these variables. Quantifiers which indicates how many of variable's instantiations need to be true for the whole proposition to be true are of two types, namely universal ($\forall$) and existential ($\exists$). "$\forall$" is called the universal quantifier because it talks about everything in the universe and "$\exists$" is the existential quantifier because it talks about the existence of some objects.

Any expression obtained by quantifying a wff over a variable is also a wff. The PC is called first order because it does not allow quantification over predicate symbols or function symbols. Thus formulas like "$\forall$p, p(X)" are not wffs in first order PC, and this is called as a second order logic or higher depending the level of the quantification of the predicate symbols (see next section).

2.2 Other logics and knowledge representations.

In the last section we have described a classical logic, i.e a predicate calculus (logic), and also its usage in representing knowledge. In the following two subsections we will discuss other logics and also other structured knowledge representations in brief respectively.

2.2.1 Other logics.

As described in the last section, the predicate logics (calculus) can be very useful for solving problems in a wide variety of domains. Unfortunately, there are many other interesting domains where a predicate logic does not provide a good way of representing and manipulating the important information such as:

"It is very cold today." How can relative degrees of cold (or heat) be represented?

"Chinese people often have small eyes". How can the amount of certainty be represented?

"If there is no evidence to the contrary, assume that any boy you meet knows how to ride a bicycle." How can we represent that one fact should be inferred from the absence of another?

"I know Nazrul thinks the Everton will win but I think they are going to lose." How can several different belief systems can be represented at once?

Another point is that the way predicate calculus (PC) makes assumptions about the relationships between conditions (antecedents) and conclusions (consequents) do not apply to

common sense reasoning in that, it is never necessary to withdraw any conclusions when additional facts became known. For example:

> If we had proved that : A-->C
> then C will continue to be true given any additional fact B i.e A&B -->C.

So PC's conclusions are additive (monotonic) and never to be revised. This is clearly not acceptable in the real world where we often have to modify or withdraw conclusions as new facts become available (non-monotonic).

Summarily, the use of logic in automated knowledge processing has received various criticisms, and generally the most common being :

- ++ That logic is not expressive enough, i.e that there is too great limit on what can be represented.
- ++ That logic cannot handle incomplete, uncertain, imprecise vague, and/or inconsistent knowledge.
- ++ That the algorithms for manipulating knowledge, which derive from logic are inefficient.

The misconception of the logic encompassing first order propositional and predicate logic (classical logic) only are causing such criticisms. In actual fact, there are many other logics , most of which were specifically designed to overcome certain deficiencies of classical logic (some of them as shown above). However some of the received criticisms could be arguably deserved.

A logic consists of a well-defined notation for the representation of knowledge, together with well-defined methods for interpreting and manipulating the knowledge which is represented. Therefore, Frost[1986] concludes that people who criticise logic are unwittingly condoning the use of ill-defined methods for knowledge processing.

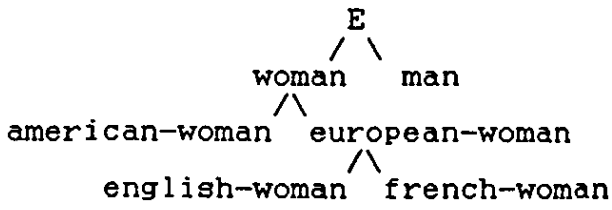
A variety of techniques for handling these problems within a computer program have been proposed, including non-monotonic logic, fuzzy logic, and probabilistic reasoning. The use of other logics are also employed such as many-sorted, situational, non-monotonic, many-valued, modal, temporal, epistemic, higher-order and intensional logics. Now we will briefly described some of the other logics:

2.2.1.1 Many-sorted logic

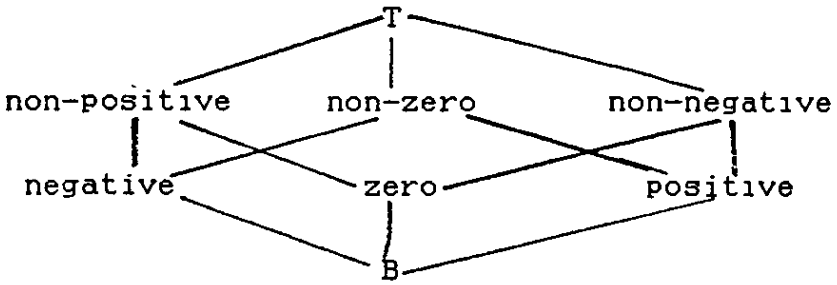
In classical first order predicate logic, a relational structure contains a single domain E of entities. Subsets of this domain are defined by use of 'unary' (one-place) predicates. In a many-sorted logic, the universe of discourse is regarded as comprising a relational structure in which the entities in the domain E are regarded as being of various sorts. The sorts are related to each other in various ways to form a 'sort structure'. There are different kinds of sort structure (Frost [1986]):

(a) Structure in which the sorts are all disjoint. For example E might consist of entities of sort: man, woman, car, bike.

(b) Structure in which the sorts are related in a 'subset' tree structure. For example:



(c) Structure in which the sorts are related in a lattice. This is the most general sort structure. For example, the following is a two-layers lattice:



where T and B is the *top* and *bottom* (empty) sorts respectively. The top and the bottom sorts are the most general (totally unspecified) and the most specific (over specified) one respectively. The bottom sort is also inconsistent. The first and second layer are set of all numbers and set of non-zero numbers respectively.

By dividing the entities in the domain of a relational structure into different sorts can help to improve the efficiency of mechanised reasoning. This is achieved when the "search space" is reduced,for examples, the meaningless assertions such as the "ford is married to the rover" can be easily detected. It should be noted, however many-sorted logics are no more expressive than unsorted logics (Enderton[1972]).

2.2.1.2 Situational logic

In many applications there is a need to store and manipulate knowledge which represents a changing universe of discourse rather than the usual static relational structures. In order to cater this type of application, 'Situational logic' was developed (McCarthy and Hayes[1969]).

In situational logic, all predicates are given an extra argument which denotes the "situation" in which the formula is true. A situation is a time interval over which no state (of interest) changes truth value. In other words, a state is something that is true for a while, false for a while, and so on. For example,

```
under(block1,block2,situation1).
~under(block1,block2,situation2).
```

The first formula states that block1 is under block2 in situation "situation1". On the other hand, the second one states that block1 is not under block2 in situation "situation2". In other words, the statement "block1 is under block2" is true in "situation1" and is false in "situation2". The transformation of "situation1" to "situation2" was caused by an "event": the event of moving block2 (or block1) elsewhere.

2.2.1.3 Non-monotonic logics

In non-monotonic logic, the addition of an assertion to a theory may invalidate conclusions which could previously have been made. On the other hand, in monotonic logic (such as predicate logic) the number of statements known to be true is strictly increasing over time. Thus, no checks are needed when asserting a new statement in the system and also no need to remember the statements used for each

successfully proven statement in monotonic logics. There are three types of circumstance in which non-monotonic reasoning may be appropriate (or in which monotonic reasoning are not very good at dealing with):

(a) In problem solving where temporary assumptions are made. For example, we have made an assumption that everybody can come to a birthday party. Until this has been proved otherwise, we will assume this assumption is a correct one.

(b) When the knowledge is incomplete, default assumption must be made which may be invalidated when more knowledge becomes available. This construction of the guesses is known as *default reasoning*. For example that we know that all birds can fly and emu is a bird, so we conclude (or believe) by default that emu can fly. But later we discovered that emu cannot fly. So we then conclude that all birds can fly except emu.

(c) When the universe of discourse is changing (as in situational logic). In this case, it is not concerned with default reasoning in the presence of incomplete knowledge but rather reasoning with out-of-date knowledge.

One implemented system that supports non-monotonic reasoning is a Truth Maintenance System (TMS) of Doyle (Doyle [1979], and Doyle[1982]). In his system, each statement or rule is called a *node*, and is, at any point in one of two states: IN if is believed to be true and OUT otherwise (because no reasons for believing it to be true of because none of the possible reasons is currently valid).

2.2.1.4 Many-valued Logics

We have described so far that the logics have all been two-valued, i.e either "true" or "false". However, there is a large amount of literatures which is concerned with logics which have more than two values. A survey of the literatures on many-valued logic can be found in Rescher [1969]. However, one of the best known many-valued logics is a three-valued logic proposed by Lukasiewicz where he introduced another "intermediate" (or undecided) value on top of "true" and "false" values. As the number of values (the degrees of truth value) of many-valued logic can be infinite, this can represent the measure of vagueness concept such as "very tall".

2.2.1.5 Fuzzy logic

Fuzzy logic is useful for dealing with "vague" concept such as "tall" where conventional logics cannot accommodate. For example:

He is tall and has curly hair.

There is much "uncertainty" in this statement. How tall he is and how curly his hair, for instance. Fuzzy logic can accommodate such uncertainty and does so by an approach to "semantics" which is quite distinct from that used in conventional logic. Various forms of fuzzy logic (Zadeh[1965], Zadeh[1973], Zadeh[1983]) have been proposed, some of which have been used to solve problems in control (Mamdani[1974]), in expert systems (ISIS system), in reasoning (Baldwin [1981]), and also have been incorporated in Prolog system (Hinde [1983] and [1986]), although the notion of fuzziness has a variety of interpretations.

2.2.1.6 Modal logic

The logics discussed so far cannot accommodate the distinction between possible worlds or environments; neither can they accommodate states of affairs which exist in people's beliefs, moral codes etc. In order to deal such things, "modal logic" was developed.

Synder[1971] has described a modal logic as a logic which allows us to reason with statements which are in subjunctive moods rather than in the indicative mood. Subjunctive statements assert what must be, ought to be, might be, is believed to be, hoped to be, will be in the future and so on. On the other hand, the indicative statements, where classical truth-functional logics are concerned with it, simply assert what is.

Modal statements can be detected by the presence of modal operators such as: "It is possible that", "It is not possible that", "It is impossible that"," It is necessary that", "It is not necessary that", "It is permissible that", "It will always be the case that", "It is known that" and so on. For examples:

(a) Tom has brain-tumour.
(b) It is necessarily true that Tom has or has not brain-tumour.
(c) It is possible that Tom has brain-tumour.

Statements (a) is an indicative statement, thus it is not modal. Both statements (b) and (c) true are modal as the operators "It si necessarily true" and "It is possible" presence in them respectively. Statement (c) is true if (a) true but may be interpreted as true or false if (a) is false.

In modal logic, we can capture modality by means of quantifiers over worlds or interpretations, i.e

(b) ∀W, (Tom has brain-tumour) \/ (Tom has not brain-tumour).
(c) ∃W, (Tom has brain-tumour).

where Ws are possible worlds, so we get:

necessity ∀W: all possible worlds.
possibility ∃W: one or more worlds.

Modal logics, then, is concerned with states of affairs of possible worlds in addition to the one that exists. There are various types of modality such as alethic modality (possibility and necessity; both statements (b) and (c) are examples of this), temporal modality (modes for sometime and always), deontic modality (modes for permission and obligation) and epistemic modality (mode for knowing and believing).

2.2.1.7 Higher-order logic

Any expression obtained by quantifying a wff over a variable is also a wff. The predicate calculus (logics) is called first order because it does not allow quantification over predicate symbols or function symbols. The following are the descriptions of some of higher-order logics:

(a) Second Order logic.

A second order logic is a logic which variable functions and predicates are allowed and can be quantified over. For example, we want to assert that if two objects, X and Y, are equal then they have some properties denoted by a variable predicate P:

∀P ∀X ∀Y X=Y --> (P(X) <--> P(Y))

First order logic only allows variables ranging over objects (i.e in this case, over X and Y only).

(b) Third Order logic.

By allowing variable functionals and quantification over them takes us into Third Order logic. The such functionals are for examples differentiation or integration functions, i.e $\int$ takes cos to sin.

(c) Omega Order Logic.

From (a) and (b) above, we can iterate the process, allowing functions of functionals (Fourth Order Logic), functions of functions of functionals (Fifth Order logic) and so on and then takes us to Omega Order Logic (Bundy[1983]). Church [1940] proposed a nice way of capturing all the sorts of functions, functional etc which is called Typed Lambda Calculus. This is one version of Omega Order Logic. All various expression such as formula, propositions, terms, functions, predicates, connectives, variables and constants can be defined in a uniform manner using the terminology of the Lambda Calculus (Lambda Operator). Lambda expression is as follows:

$$\lambda X[...X...]$$

where ...X... denotes a formula which has X as a free variable, for example, λX[loves(X,God)] denotes the (function of) the set of individuals that loves God.

2.2.1.8. Intensional logic

Intensional logic which was developed by Montague (Montague[1973], Montague[1974]) employs many of the logic concepts as explained before such as a hierarchy, higher-order quantification (variables and quantifiers for each type- a higher order logic), lambda abstraction for all types, tenses, modal operators, syntactic mechanisms for dealing with intensions and extensions.

The language of Intensional Logics was used by Montague as an intermediate translation language in a system called PTQ (for Proper Treatment of Quantification in Ordinary English) which is used to derived a semantic interpretation of a fragment of English language, for example, "a woman" is translated into:

$$\lambda P[\ \exists X,\ woman'(X) \dashrightarrow P(X)]$$

Hence, "a woman jogs and talks" is translated into:

$$\lambda P[\ \exists X,\ woman'(X) \dashrightarrow P(X)](\ \lambda Y[\ jogs'(Y) \wedge talks'(Y)])$$

which, by lambda conversions, becomes:

$$\exists X,\ woman'(X) \wedge jogs'(X) \wedge talks'(X)$$

Nishida and Doshita[1983] have applied the Montague's method in automatic translation of English into Japanese. Montague's method also have been applied in "historical database system" (Clifford and Warren[1983]), where a historical database system contains knowledge representing some time-varying universe of discourse.

2.2.2 Other structured knowledge representation

We have already discussed the language of formal logic which allow us to represent various aspects of the universe. The major advantage of these representation, particularly predicate calculus (logic), is that they can be combined with simple, powerful inference mechanisms, such as resolution, that make reasoning with the facts easy. However, they do not in general allow us to structure this knowledge to reflect the structure of that part of the universe which is being represented. In other words, the objects in formal logic representation are so simple that much of the complex structure of the world cannot be described easily.

The world, for example, contains individual objects, each of which has several properties, including relationship to others. It is often useful to collect those properties together to form a single description of complex object. One advantage of such a scheme is that it enables a systems to focus its attention on entire objects without also having to consider all the other facts it knows. For example, we have the information represented in first order logic as follows:

```
is_married(Razak,Rodziah).      /* f1 */
is_married(John,Joan).          /* f2 */
is_employed(Razak,UKM).         /* f3 */
is_employed(John,LUT).          /* f4 */
has_eye_colour(Razak,brown).    /* f5 */
has_eye_colour(John,blue).      /* f6 */
```

Here, the order of assertions is irrelevant. In formal logic, there is no facility for clustering formulas such f1, f2 and f3 which are related to a particular object (in this case, Razak) to form a single description of a complex object.

So, the appropriateness of a representation depends on the application. A good system for the representation of complex structured knowledge in a particular domain should possess the following four properties (Rich [1983]).

(a) Representational Adequacy - the ability to represent all of the kinds of knowledge that are needed in that domain.

(b) Inferential Adequacy - the ability to manipulate representational structures in such a way as to derive new structures corresponding to new knowledge inferred form old.

(c) Inferential efficiency - the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mechanisms in the most promising directions.

(d) Acquisitional efficiency - the ability to acquire new information easily. The simplest case involves direct insertion, by a person, of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.

Several techniques have been developed for accomplishing these objective and can roughly be divided into two types:

[1] *declarative method*: most of knowledge is represented as a static collection of facts accompanied by as small set of general procedures for manipulating them (e.g predicate logic). Each fact need only be stored once, regardless of the number of different ways in which it can be used. This method also allows new facts to be added to the system easily without changing either the other facts or the small procedure.

[2] *procedural method*: the bulk of the knowledge is represented as procedures for using it. Knowledge of how to do things and knowledge that does not fit well into many simple declarative schemes are easy to be represented by using this method, for example, the default reasoning (non-monotonic logic). It is also easy to represent heuristic knowledge of how to do things efficiently.

However, in practice, most representations employ a combination of both methods (Brachman & Smith[1980]). There is a variety of knowledge structures where each of them is a data structure in which knowledge about particular problem domains can be stored. Thus knowledge structures will sometimes mean a complete database of information about a particular domain and will sometimes refer to substructures within the larger structure. We use the phrase knowledge structure to describe these representational schemes, because of the heavy emphasis on the structure of the representation. Such knowledge structures are semantic nets, frames, conceptual dependency and scripts. We will described briefly all these structures and for more detail see books on Artificial Intelligent such as Frost[1986], Rich[1983], Charniak & McDermott[1985], Nilsson[1980].

2.2.2.1 Semantic nets

Semantic nets is closely related to first order logic and were first used by Quillian[1968] and, independently, Raphael[1968] to represent originally the meanings of english words. Semantic nets is a directed graph in which nodes represent entities and arcs represent binary

relationships between entities. A single entity is represented by a single node. Arcs are labelled with the names of the relationships types. It is general enough to be able to describes both events and objects (entities).

Many complex objects can be decomposed into simpler ones. These decompositions yield two very common and useful properties of objects ,i.e ISA and ISPART relationships. ISA relationship shows the relationships between objects in a hierarchical taxonomy. On the other hand, ISPART relationship shows the relationships between the objects that are made up of a set of components and so forth. In semantic net, apart from ISA and ISPART relationships, there are other relationships which can be used such as "HEIGHT", "COLOUR" etc.

Those relationship as said before can be represented as a graph and it is very useful to think like it (see Fig. 2.2.2.1 below). A fragment of a typical semantic net is as follows:

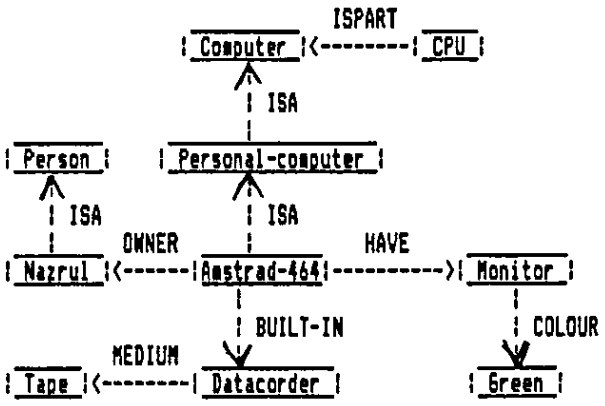


Fig. 2.2.2.1: A semantic network

As said before that semantic net is related to first order predicate calculus. It is clear that it can be used to

represent two-placed predicates in predicate calculus. For example, some of the arcs from Fig. 2.2.2.1 could be represented in predicate logic as follow:

```
ISA(Amstrad-464,Personal-computer).
ISA(Nazrul,Person).
COLOUR(Monitor,Green).
HAVE(Amstrad-464,Monitor).
```

Semantic nets are not restricted to represent first order predicate only, but can also represent other predicates. They are usually represented using some kind of attribute-value memory structure. For example, the above semantic networks shown in Fig 2.2.2.1 would be represented in Prolog (see section 2.4 for more explanations of it) as follows:

```
f(ispart,cpu,computer).

f(isa,personal-computer,computer).

f(isa,amstrad-464,personal-computer).
f(have,amstrad-464,monitor).
f(built-in,amstrad-464,datacorder).
f(owner,amstrad-464,nazrul).

f(isa,nazrul,person).

f(colour,monitor,green).

f(medium,datacorder,tape).
```

It can be seen that there are six atoms, i.e computer, personal-computer, amstrad-464, monitor, datacorder and nazrul. The predicates are in the form of *f(R,A,P)* where R is a relationship (binary), A is an atom and P is a property-list of A. It should be noted here that A and P are both entities. By representing in Prolog in the above ways, we are able to get the information about the relationship between the entities easily where it will be difficult to know the relationship between entities if the predicate is in the form of "*R(A,P)*", for example, *isa(nazrul,person)* due to Prolog's implementation.

Work on semantic networks stem from many sources. The pioneers, Quillian[1968] and Raphael[1968] suggest the usage in cognitive psychology and computer science. Other usage of semantic networks are such as in representing and learning information about configurations of blocks (Winston[1975]), in cognitive psychology (Anderson and Bower[1973], Rumelhart and Norman[1975]- they proposed memory models based on networks), in natural language processing (Simmons[1973], Walker[1978]) and in database management (Mylopoulus et al[1976]). Several different types of semantic networks were described in Findler[1979]. Hendrix [1977] and [1979] also developed another type of semantic networks which is called a partitioned semantic network.

2.2.2.2 Frame

People always analysed a new situation by using their previous experience by evoking appropriate stored structures and the fill them in with the details of the current event or situation. A general mechanism designed for the computer representation of such common knowledge is a *frame*.

Frame is often used to describe a collection of attributes that a given object, such as a chair, normally possesses. A frame is a data structure which represents an entity type. The word frame has been applied to a variety of slot-and-filler representation structures, mostly following the theory presented in Minsky[1975]. In other words, a frame consists of a collection of named "slot", that describe aspects of the object, each of which can be "filled" by values or by pointers to other frames describing other objects.

For example (adapted from Frost[1986]):

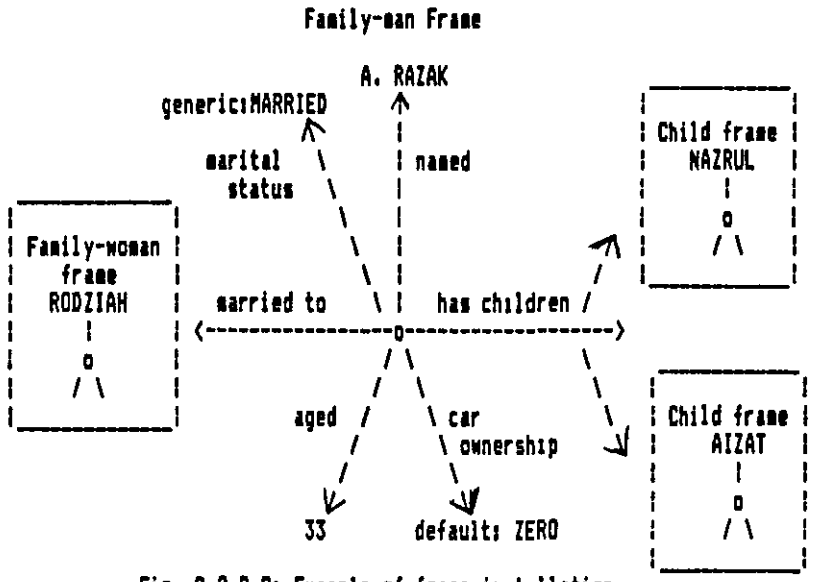


Fig. 2.2.2.2: Example of frame installation

In a frame, the slots can be filled by another frame such as child and family-woman frames in the family-man frame. The value of slots can be a pre-set value (generic value) in every frame instantiation such as the value MARRIED in the "marital status" slot in family-man frame The value of slot can be set to default value as in the "car ownership" slot where the default value is ZERO.

Frames, like semantic nets, are general purpose structures in which particular sets of domain-specific knowledge can be embedded. The details of the operation of a frame-based system vary with the sort of reasoning that the system will be called upon to perform and also with the specific kinds of knowledge the frames will contain.

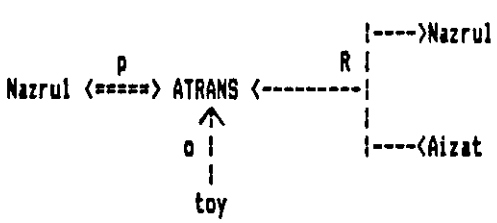
There are implemented frame languages which allow the user to build frame system such as KRL-0 and KRL-1 (Bobrow and Winograd [1977a], [1977b] and [1979]), FRL (Roberts and Goldstein [1977]), OWL (Szolovits et al. [1977]).

There are also many implemented frame-based systems such as UNITS (Smith and Friedland [1980]) which was developed for application in molecular biology, WHEEZE (Smith and Clayton [1980]) which performs medical pulmonary function diagnosis based on clinical test results, AM (Lenat [1982]) which was designed to discover concepts in mathematics, and GUS (Bobrow et al. [1977]) which is a frame-driven dialogue system.

2.2.2.3 Conceptual Dependency

Conceptual Dependency is a theory of how to represent the meaning of a natural language sentence in a way that facilitates drawing inferences from sentences and is independent of the language in which the sentences were originally stated. The theory was first described in Schank[1973] and was further developed in Schank[1975] and Schank & Abelson[1977].

The conceptual dependency (often nicknamed CD) is built not out of primitives corresponding to the words used in the sentence, but rather out of conceptual primitives that can combined to form the meanings of words in any particular language. For example, the sentence "Nazrul took the toy from Aizat" would be conceptualized as "The toy was ATTRANSed from Aizat (the original owner) to Nazrul (the new owner)" or is represented as follows:



where "o", "p" and "R" are objective case, past tense and recipient case. ATRANS is a one of the primitive ACTs used by the CD theory to indicate a transfer of possession from one owner to another, in this case, "toy", from Aizat to Nazrul. Arrows indicate direction of dependency. A double arrow indicates two way link between actor and action. The primitive ACTs describe everyday human actions.

In brief, CD is a special-purpose structure in which specific primitives to be used in building individual representations were defined, as were relationships that could occur between elements of a representation. And, CD requires that all knowledge be decomposed into fairly low-level primitives. This may be inefficient or perhaps even impossible in some situations (Rich [1983]). A lot of work must be done to convert each high-level fact into its primitive form and then the primitives may require a lot of storage. Another problem is that it is not at all clear what the primitives should be.

2.2.2.4 Script

A script is a special-purpose structure that exploits specific properties of their restricted domain. It is also a structure which describes a stereotyped sequence of events or a commonly occurring sequence of events in a particular context such as "going into a restaurant and ordering, eating, and paying for a meal" in a restaurant script (Schank and Abelson[1977]).

A script consists of a set of slots. Associated with each slot may be information about what kinds of values it may

contain, as well as a default value to be used if no other information is available. Typically, sets of slots are such as a set of entry conditions, a set of roles, a set of props, a set of scenes and a set of results.

A set of entry conditions must be satisfied before the script may be instantiated. People who would typically be involved in instances of the scripts are slots in a set of roles. Furthermore, objects which also would typically be involved in instances of the scripts are slots in a set of props. A set of events making up the sequence of events represented by the script is called a set of scenes. A set of results will be obtain after the sequence of events has been completed.

Scripts are useful because, in the real world, there are patterns to the occurrence of events. These patterns arise because of causal relationships between events. The events described in a script form a giant causal chain where the beginning and the end of the chain are the set of entry conditions and the set of results respectively. Some examples of script based systems are SAM (Cullingford [1981]) which has been used to understand newspaper stories, McSAM or Micro SAM (Sterling and Shapiro[1986]) which is a simplified version of SAM and was written in Prolog, and IPP (Lebowitz [1980]) which read and remembered newspaper stories concerning international terrorism.

2.3 Resolution

In this section we will discuss on how to conclude whether a new statement follows form the known statements. This proof procedure is based on the resolution principle which was proposed by Robinson [1965].

It would be useful from a computational point of view if we had a proof procedure that carried out in a single operation the variety of processes involved in reasoning with statements in predicate calculus. Resolution is such a procedure which gains its efficiency from the fact that it operates on statements that have been converted to a very convenient standard form, i.e a clausal (clause) form. Before we describe how the resolution operates, we will discuss the standard form in which statements will be represented and will be used in the resolution.

2.3.1. Conversion into Clausal Form

If the formula were in simpler form, the process of resolution would be easier. The formula would be easier to work with if it were flatter, i.e there was less embedding of components and the quantifiers were separated from the rest of the formula so that they did not need to be considered.

Conjunctive normal form (Davis and Putnam [1960]) has both properties. Since there exists an algorithm for converting any wff into conjunctive form, we lose no generality if we employ a resolution procedure that operates only on wff's in this form.

However, the resolution method requires formulas to be converted to a regular form called clausal form. A *clause* is defined as a wff consisting of a disjunction of literals. Formulas can be converted to clausal form once they have been transformed to conjunctive normal form. The resolution process, when it is applicable, is applied to a pair of of parent clauses to produce a derived clause. The following is a brief explanation of a sequence of steps to convert a wff into clausal form (the detailed explanation will be given in the next chapter):

[1]. Eliminate implication and equivalence signs (symbols). Replace a-->b with ~a\/b to eliminate implication signs. And, a<-->b can be replaced with (a-->b)/\(b-->a).

[2]. Reducing the scope of the negation signs, i.e moving inwards the negation signs by using De Morgan's law.

[3]. Standardize variables so that *each quantifier binds a unique variable*. Within the scope of any quantifier, a variable bound by that quantifier is a dummy variable and can be uniformly replaced by any other (non-occurring) variable throughout the scope of the quantifier without changing the the truth value of the wff. For example, the formula

$$\forall X\ p(X) \vee \forall X\ q(X)$$

would be converted to

$$\forall X\ p(X) \vee \forall Y\ q(Y)$$

[4]. Eliminate existential quantifiers by replacing each occurrence of its existential quantified variable with a Skolem function.

[5]. Convert to prenex form by moving all universal quantifiers to the front (left) of the wff. The resulting wff is said to be in *prenex form* consisting a *prefix* of quantifiers followed by a *matrix*, which is quantifier-free.

[6]. Eliminate all universal quantifiers (the prefix) as the remaining variables left are universally quantified. So, we are left now with a matrix only.

[7]. Converting a matrix into conjunction of disjuncts (conjunctive normal form) by applying a distribution law of logic.

[8]. Eliminate $\wedge$ (AND) symbols by replacing with the set of disjunction of literals. Any wff consisting solely of a disjunction of literals is called a clause. For example:

$$(p(X) \vee q(Z)) \wedge (r(X) \vee s(Z,a))$$

can be written as

$$\{ (p(X) \vee q(Z)) , (r(X) \vee s(Z,a)) \}$$

[9]. Standardize apart the variables *in the set of clauses* generated in step [8] such that no two clauses make reference to the same variable. For example, from step [8], the final clauses will be:

$$\{ p(X) \vee q(Z) , r(Y) \vee s(W,a) \}$$

As we can see that the literals may contain variables but these variables are always understood to be universally quantified. If variables are substituted with terms in an expression, we will obtain what is called a *ground instance* of the literal, for example, "s(b,a)" is a ground instance of "s(W,a)".

The final set of clauses, each of which is a disjunction of literals, can now be exploited by the resolution procedure to generate proofs. As we said above that each step of converting a wff into clauses will be fully explained in the next chapter.

2.3.2 Horn Clauses

A special important class of clauses, both because they arise so often in practice and because simplified theoretical results apply to them are the Horn clauses, named after Alfred Horn [1951], who originally isolated them. For many applications of logic, it is sufficient to restrict the form of clauses to those containing at most one conclusion (Kowalski [1979]). Clauses containing at most one conclusion (consequent) are called *Direct Horn clauses*. It can be shown, in fact, that any problem which can be expressed in predicate logic can be re-expressed by means of Horn clauses.

Obviously there are two types of Horn clauses, i.e *headed* and *headless* clauses. Headed and headless clauses are clauses with one unnegated literal (or one conclusion) and with no unnegated literal respectively. In other words, there are four forms of Horn clauses:

[1]. *Implication clause*: $a_1 \wedge \ldots \wedge a_n \rightarrow h$

[2]. *Goal clause*: $a_1 \wedge \ldots \wedge a_n \rightarrow$

[3]. *Assertion clause*: $\rightarrow h$

[4]. *Empty clause*: $\rightarrow$

So, implication and assertion clauses are headed Horn clauses. And, goal and empty clauses are headless Horn

clauses. For simplicity, we will denote assertion and empty clauses as h and [] respectively, i.e by taking out the implication signs. In fact, when we consider sets of Horn clauses, we need only to consider those sets where all but one of the clauses are headed. That is, any soluble problem (a theorem proving task) that can be expressed in such a way that:

(i) there is one headless clause.

(ii) all the rest of the clauses are headed.

Since it is an arbitrary how we decide which clauses are actually the goals, we can decide to view the headless clause as the goal and the other clauses as hypotheses or axioms (known statements) or premises. This has a certain naturalness.

2.3.3 The Basis of Resolution

The theoretical basis of the resolution procedure in predicate logic is Herbrand's theorem (Chang and Lee [1973]), which tells us the following:

[1] To see if a set of clauses S is unsatisfiable, it necessary to consider only interpretations over a particular set, called the Herbrand Universe of S.

[2] A set of clauses S unsatisfiable if and only if a finite subset of ground instances (in which all bound variables have had a value substituted for them) of S is unsatisfiable.

Herbrand Universe of a set of clauses S is a special domain such that S is unsatisfiable if and only if S is false under all the interpretations over this domain. Herbrand's theorem is a very important theorem as it is a base for most modern

proof procedures in mechanical theorem proving. The second part of the above theorem, [2], suggest a refutation procedure.

Gilmore was one the first men to implement the above idea (Gilmore [1960]). However the method used by Gilmore is inefficient. To overcome this inefficiency, Davis and Putnam [1960] introduced a more efficient method for testing the unsatisfiability of a set of ground clauses. Both methods which are based on Herbrand's theorem requires the generation of sets of ground instances of clauses, and for most cases, this sequence grows exponentially. For instance, for a small set of ten two-literal ground clauses, there are 1024 (2^{10}) conjunctions.

In order to avoid the generation of sets of ground instances as required in Herbrand's procedure, Robinson[1965] suggests a resolution principle which can be applied directly to any set of clauses, S, (not necessarily ground clauses) to test the unsatisfiability of S.

The essential idea of the resolution principles is to check whether a set of clauses, S, contains the empty clause, []. If S contains [], then S is unsatisfiable. If S does not contains [], then the next thing to do is to check whether [] can be derived form S. Indeed, the resolution principle can be viewed as an inference rule that can be used to generate new clauses from S.

The resolution procedure is a simple iterative process. It operates by taking two clauses that each contain the same

literal. These two clauses are called *parent clauses*. The literal must occur in positive form in one clause and in negative form in the other. The *resolvent* is obtained by combining all of the literals of the two parent clauses except the ones that cancel. For example:

(a) sunny $\vee$ raining
(b) ~sunny $\vee$ cold
(c) raining $\vee$ cold
(d) win
(e) ~win
(f) []

The literal "sunny" in parent clause (a) and "~sunny" in parent clause (b) will cancel each other to form a resolvent clause (c).

If the clause that is produced is empty clause, then a contradiction has been found, for example, two clauses (d) and (e) will resolve each other to produce an empty clause (f). If a contradiction exists, then eventually it will be found. Of course, if no contradiction exists, it is possible that the procedure will never terminate, although there are are often ways of detecting that no contradiction exists.

Another way of viewing the resolution process is that it takes a set of clauses all of which are assumed to be true. It generates new clauses that represent restrictions on the way each of those original clauses can be made true, based on information provided by the others. A contradiction occurs when a clause becomes so restricted that there is no way it can be true. This is indicated by the generation of the empty clause.

2.3.4 Unification

The most important part of applying the resolution principle is finding a literal in a clause that is complementary to a literal in another clause. In propositional logic, it is easy to determine two literals that are complementary each other, i.e by simply looking for ~*l* and *l*. However, for predicate logic (or for clauses containing variables), it is more complicated. For example:

(c1). p(X) $\vee$ q(X)
(c2). ~p(b) $\vee$ r(Y)

There is no literal in (c1) that is complementary to any literal in (c2). However, if we substitute b for X in (c1) we will obtain:

(c1)'. p(b) $\vee$ q(b)

We know that clauses (c1)' and (c2) can be resolved with each other to obtain "q(b) $\vee$ r(Y)". Thus in order to determine two complementary literals, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical (or complementary). The unification procedure will do just this.

The basic idea of unification is very simple. The matching rules are simple. Different constants, functions, or predicates cannot match; identical ones can. A variable can match another variable, any constant, or a function or predicate expression with the restriction that the function or predicate expression must not contain any instances of the variable being matched. The only complication in this procedure is that we must find a single, consistent substitution for the entire literal, not separate ones for each piece of it.

The single consistent substitution is called the *most general (or simplest) unifier*.We mentioned in the above paragraph that a variable can match a function or predicate expression with a restriction. This restriction is called *occur check*, For example, two clauses X of p(X) and f(X) of ~p(f(X)) cannot be matched or unified with each other as f(X) is a function which contains a variable being matched, i.e variable X.

2.3.5 The soundness and completeness

The soundness and completeness of resolution is a nice mathematical property. It means that if some statement or fact follows from known statements, we should be able to prove its truth using resolution. It is called *sound* because if the empty clause is ever produced, the original set must have been unsatisfiable. It is called *complete* because if the original set is unsatisfiable, the empty clause will eventually be produced.

The resolution process of deriving new clauses from old will eventually derive the empty clause if and only if the original clauses were unsatisfiable. The 'only if' part of this is the soundness theorem: that we cannot produce an unsatisfiable conjunction of clauses, and in particular, one containing the empty clause, from a satisfiable one. The 'if' part is the completeness theorem: that is, if we start with an unsatisfiable set and go on deriving new clauses by resolution, we will eventually derive the empty clause. The proof of both theorems can be found in Bundy [1983](pp233-234).

2.4 Control strategies for refutation process

The procedure in which the formula being tested is negated are called "refutation" procedure. In other words, to prove a statement, the resolution attempts to show that the negation of statement produces a contradiction with the known statements, i.e that it is unsatisfiable.

Resolution-based systems are designed to produced proofs by contradiction or refutation. In resolution refutation, we first negate the goal wff and then add the negation to the set,S. This expanded set is then converted to a set of clauses, and we use resolution in an attempt to derive a contradiction represented by the empty clause.

Although resolution tell us how to derive a consequence from two clauses, it does not tell us either how to decide which clauses to look at next or which literals to match. If the choice of clauses to resolve together at each step is made in certain systematic ways, then the resolution procedure will find a contradiction if one exists. However it may take a very long time. There exist control strategies for making the choice that can speed up the process considerably. Many refinements of the original resolution principle have been proposed, in order to control a resolution refutation principle such that it does not produce unnecessary clauses.

A control strategy for a refutation system is said to be complete if it use results in a procedure that will find a contradiction (eventually) whenever one exists. In AI applications, complete strategies are not so important as ones that find refutations efficiently.

Several control strategies for selecting clauses have been developed for resolution that not produce unnecessary clauses such as breadth-first strategy, the set-of-support strategy, the unit-preference strategy, the linear-input form strategy, the ancestry-filtered form strategy and combinations of the strategies.

Beside control strategies to avoid producing unnecessary clauses, there are strategies which are called simplification strategies. The aim of the simplification strategies is to reduce the rate of growth of new clauses. Such strategies involve the elimination of all tautologous clauses and clauses that are subsumed by others, and the resolving of pairs of clauses that contain complementary literals only.

We have mentioned some of control strategies for selecting clauses. Some authors (Chang and Lee [1973], Frost [1986]) use term resolution strategies instead of control strategies (Nilsson [1980], Rich [1983]). We will use both terms ("control strategies" and "resolution strategies") interchangeably thorough out in this thesis as both of them are synonymous. Now we will describe some of the control strategies.

2.4.1 Semantic Resolution

Semantic resolution was proposed by Slagle[1967]. In semantic resolution, an interpretation to divide clauses is used. For example, by using ordinary resolution to prove the unsatisfiability of the set S where S consists of the first four clauses as follows (Chang and Lee [1973]):

(1)	$\neg p \vee \neg q \vee r$	}
(2)	$p \vee r$	} S
(3)	$q \vee r$	}
(4)	$\neg r$	}
(5)	$\neg q \vee r$	from (1) and (2)
(6)	$\neg p \vee r$	from (1) and (3)
(7)	$\neg p \vee \neg q$	from (1) and (4)
(8)	p	from (2) and (4)
(9)	q	from (3) and (4)
(10)	$\neg q \vee r$	from (1) and (8)
(11)	$\neg p \vee r$	from (1) and (9)
(12)	r	from (2) and (6)
(13)	$\neg q$	from (4) and (5)
(14)	$\neg q$	from (4) and (6)
(15)	[]	from (4) and (12)

Example 2.4.0: Using unrefined resolution

Each segment shows the level of resolution where the new generated clauses from a new segment are resolved with the previously generated clauses. Among all these clauses generated, only (6) and (12) are actually used in the proof. All other clauses are irrelevant and redundant.

So, semantic resolution tries to avoid this. As said before, they use an interpretation to divide clauses such that some irrelevant and redundant clauses can be avoided. For example, suppose we divide the set S into two, i.e {(2),(3)} and {(1),(4)}. Thus, the resolution (1) and (4) will be avoided if we adopt the restriction that no two clauses from the same division (subset) are allowed.

Another restriction which is adopted by the semantic resolution is by ordering the predicate symbols, for example:

$$p > q > r$$

and when we resolve two clauses from two subsets, we only choose the largest predicate from, says, the first subset. With this restriction, we cannot resolve (2) with (4) because r is not the largest literal in (2) and (3).

By using the same set S as in the above example 2.4.1 and and let the ordering of predicate symbols be p>q>r and let the interpretation be {~p,~q,~r} such that the set S will be divided into S_1={(2),(3)} and S_2={(1),(4)}. Thus we will get the following:

(1)*	~p ∨ ~q ∨ r	}
(2)	p ∨ r	} S
(3)	q ∨ r	}
(4)*	~r	}
(5)*	~q ∨ r	from (1) and (2)
(6)*	~p ∨ r	from (1) and (3)
(7)	r	from (3) and (5)
(8)	r	from (2) and (6)
(9)	[]	from (4) and (7) or (8)

Example 2.4.1: Using semantic resolution

Clauses marked by "*" are members of subset S_2 which satisfy the adopted interpretation.

It can be seen that by adopting some restrictions on the resolution, the irrelevant and redundancy clauses can be avoided as shown by the semantic resolution. Chang and Lee [1973] showed that the semantic resolution is a complete resolution.

2.4.2 Hyper-resolution

Hyper-resolution which was introduced by Robinson [1965a] is a special kind of semantic resolution which uses a special kind of interpretation, i.e an interpretation in which every literal is the negation of the atom. For example, we would like to prove the unsatisfiability of set S consisting the first four clauses as in example 2.4.3 below. Let the ordering be p<q<r and certainly the interpretation, I, be {~p(a), ~q(a), ~r(a)}, thus:

(1)	q(a) ∨ p(X)	)	
(2)*	~q(X) ∨ p(X)	) S	
(3)*	~r(a) ∨ ~p(a)	)	
(4)	r(a)	)	
(5)	p(X) ∨ p(a)	from (1) and (2)	
(6)*	~p(a)	from (3) and (4)	
(7)	[]	from (5) and (6)	

Example 2.4.2 Using hypher-resolution

As before, clauses marked "*" are satisfied by the adopted interpretation.

2.4.3 Set-of-support strategy

The set-of-support strategy was proposed by Wos et al[1965]. This strategy is also a special kind of semantic resolution in the sense it divides the set to be proved unsatisfiable into two. A subset T of a set S of clauses is called a *set of support* of S if S-T is satisfiable (Chang and Lee[1973). So, a set-of-support resolution is a resolution of two clauses that are not both from S-T, i.e the two subsets are T and S-T.

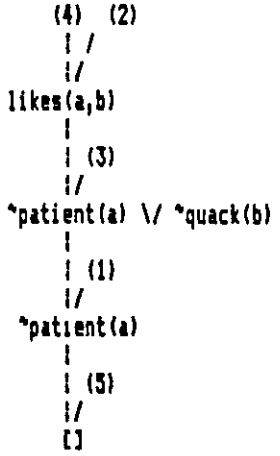
In other words, whenever possible, resolve either with one of the clauses that is part of the statement we are trying to refute or with a clause generated by a resolution with

such a clause. This is corresponds to the intuition that the contradiction we are looking for must involve the statement we are trying to prove. Any other contradiction would say that the previously believed statements were inconsistent. Thus, in this case, sets T and S-T are statements to be proved and known statements (knowledge) respectively.

For example, let S_1 be a set consisting the first three clauses and the negation of the statement to be proved from S_1 are clauses (4) and (5) as follows:

```
(1)   patient(a)                              )
(2)   ~doctor(Y)  \/ likes(a,Y)               ) S1
(3)   ~patient(X) \/ ~quack(Y) \/ ~likes(X,Y) )
(4)   doctor(b)       )
(5)   quack(b)        ) T
```

The set-of-support is a set T where T = {(4),(5)} and its complementary set is set S_1 where S_1=S-T={(1),(2),(3)}. The following is a refutation tree of the above proving:



Example 2.4.3 Using set-of-support strategy

We can see that no resolution is performed between clauses in the same set S_1.

2.4.4 Lock resolution

Lock resolution is a refinement of resolution which uses a concept similar to that of ordered clauses and was introduced by Boyer[1971]. The literals of clauses in a set S is ordered according to their indices. The index is arbitrarily given to each occurrence of a literal in S with an integer. Resolution is then permitted only on the literals of lowest index in each clause with all the literals inherit their indices from their parent. In other words, the index of each clause is maintained through out the resolution process. If there are more than one with the same literals, all the literals will be merged into one and the lowest index is assigned to it, i.e take only one literal with the lowest index, for example, the clause "$_1q \vee {}_2q$" will become "$_1q$". For example, let S = {p, q, r, w, ~p $\vee$ ~q $\vee$ ~r $\vee$ ~w} is a set to be proved unsatisfiable. We will give an index to each literal arbitrarily as follows:

(1)	$_1p$	
(2)	$_2q$	
(3)	$_3r$	
(4)	$_4w$	
(5)	$_5{\sim}p \vee {}_6{\sim}q \vee {}_7{\sim}r \vee {}_8{\sim}w$	
(6)	$_6{\sim}q \vee {}_7{\sim}r \vee {}_8{\sim}w$	from (1) and (5)
(7)	$_7{\sim}r \vee {}_8{\sim}w$	from (2) and (6)
(8)	$_8{\sim}w$	from (3) and (7)
(9)	[]	from (4) and (8)

Example 2.4.4: Using Lock resolution

From the above example, only three lock resolvents were generated. The lock resolution does not permit the resolution between others pair of parent clauses such as (2) and (5), (3) and (5) etc. If ordinary (unrefined) resolution were used, 40 clauses would be generated by the breadth-first method before [] could be generated. The lock resolution is a complete resolution (Chang and Lee [1973]).

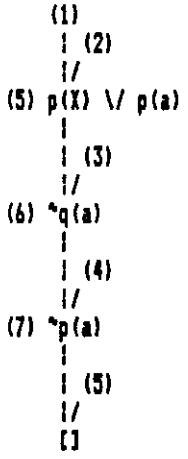
2.4.5 Linear Resolution

Linear resolution was independently proposed by Loveland[1970] and Luckham[1970]. The idea of linear resolution is similar to proving an identity in mathematics. In proving an identity, we often start with the left-hand side expression of the identity, apply an inference rule (an axiom) to obtain a new expression, then repeatedly apply some inference rule again to the new freshly obtained expression until the left-hand side expression is identical to the right-side expression of the identity.

Thus, a linear resolution starts with a clause, resolves it against a clause to obtain a resolvent, and resolves this resolvent against some clause until the empty clause [] is obtained. The two parents clauses are the new resolvent and a clause from the set S to be prove unsatisfiable or from one of the previous resolvents. For example, let set S to be proved unsatisfiable be:

(1) q(X) \/ p(a)
(2) ~q(X) \/ p(X)
(3) ~q(X) \/ ~p(X)
(4) q(X) \/ ~p(X)

then the following is its refutation graph:



Example 2.4.5: Using linear resolution

It can be seen from example 2.4.5 that in order to resolve (7), we use the previous resolvent, i.e (5). Following Chang and Lee [1973], clauses (1), (5), (6) and (7) are called centre clauses and clauses (2), (3), (4) and also (5) are called side clauses. So, linear resolution can use any previous centre clauses or from base set S. Nilsson's ancestry-filtered form strategy (Nilsson [1971]) is similar to this linear resolution. Clause (5) is called an ancestor clause. This linear resolution or ancestry-filtered form strategy is complete (see Chang and Lee[1973]).

2.4.6 Linear Input Resolution

Linear input resolution (or input resolution in Chang and Lee[1973] or vine form in Nilsson[1971]) is a special case of linear resolution where all side clauses must be from the base or input set S. The base or input set S is a set to be proved unsatisfiable.

To prove a statement, F, using this method, we commence by resolving ~F with one of the input clauses to form a resolvent, C. C is then resolved with also an input clause to form another resolvent, and so on until we get an empty resolvent or otherwise. For example, let an input set S be {(1),(2),(3),(4),(5),(6),(7)} and the negation of the statement, F, to be proved be {(8),(9)} where:

(1) ~wombat(X) ∨ ~lives(X,zoo) ∨ ~happy(X)

(2) happy(Z) ∨ ~animal(Z) ∨ ~meets(Z,Y) ∨ person(Y) ∨ kind(Y)

(3) kind(V) ∨ ~person(V) ∨ ~visits(V,zoo)

(4) meets(U,Y) ∨ ~animal(U) ∨ ~lives(U,zoo) ∨ ~person(Y) ∨ ~visits(Y,zoo)

(5) animal(A) ∨ ~wombat(A)

(6) person(naz)

(7) visits(naz,zoo)

(8) wombat(w)

(9) lives(w,zoo).

So, the following is an illustration to prove the statement F using a linear input resolution (we start with clause(8)):

(10) ~lives(w,zoo) ∨ ~happy(w) from (8) and (1)
(11) ~lives(w,zoo) ∨ ~animal(w) ∨ ~meets(w,Y) ∨ person(Y) ∨ kind(Y)
from (10) and (2)
(12) ~lives(w,zoo) ∨ ~animal(w) ∨ ~meets(w,Y) ∨ ~person(Y) ∨ ~visits(Y,zoo)
from (11) and (3) - after merging
(13) ~lives(w,zoo) ∨ ~animal(w) ∨ ~person(Y) ∨ ~visits(Y,zoo)
from (12) and (4) - after merging
(14) ~lives(w,zoo) ∨ ~wombat(w) ∨ ~person(Y) ∨ ~visits(Y,zoo)
from (13) and (5)
(15) ~lives(w,zoo) ∨ ~wombat(w) ∨ ~visits(naz,zoo)
from (14) and (6)
(16) ~lives(w,zoo) ∨ ~wombat(w) from (15) and (7)
(17) ~lives(w,zoo) from (16) and (8)
(18) [] from (17) and (9)

Example 2.4.6: Using linear input resolution

The above prove the statement F is a theorem of S. The new resolvent is always resolved with the input clause S in linear resolution. Thus, it lacks completeness, for example, we will not be able to prove the unsatisfiability of set S as in example 2.4.5 above. However, this linear resolution is efficient and furthermore it is complete for Direct Horn clauses (Frost[1986]).

2.4.7 Unit preference strategy

Unit preference strategy (or unit resolution in Chang and Lee[1973]) which was proposed by Wos et al[1964] is a special kind of linear resolution. This resolution, whenever possible, resolves with clauses with a single literal (called a unit). Such resolution generates new clauses with fewer literals than the larger of their parent clauses, and thus are probably closer to the goal of a resolvent with zero terms (empty clause). This method is essentially an extension of the one-literal rule of Davis and Putnam [1960].

However this method may take longer to reach the solution (empty clause) if it exists. The following example shows this shortfall. The following illustrates proving a statement using unit-preference strategy. We will use the same set of input clause and statements to be proved as in example 2.4.6 above (see section 2.4.6):

```
(10) ~lives(w,zoo) \/ ~happy(w)                    from (8) and (1)
(11) ~happy(w)                                     from (10) and (9)
(12) ~animal(w) \/ ~meets(w,Y) \/ person(Y) \/ kind(Y)
                                                   from (11) and (2)
(13) ~animal(w) \/ ~meets(w,naz) \/ kind(naz)
                                                   from (12) and (6)
(14) ~animal(w) \/ ~meets(w,naz) \/ ~person(naz) \/ ~visit(naz,zoo)
                                                   from (13) and (3)
(15) ~animal(w) \/ ~meets(w,naz) \/ ~visit(naz,zoo)
                                                   from (14) and (6)
(16) ~animal(w) \/ ~meets(w,naz)                   from (15) and (7)
(17) ~animal(w) \/  ~lives(w,zoo) \/ ~person(Y) \/ ~visits(X,zoo)
                                                   from (16) and (4)- after merging
(18) ~animal(w) \/  ~lives(w,zoo) \/ ~visits(naz,zoo)
                                                   from (17) and (6)
(19) ~animal(w) \/  ~lives(w,zoo)                  from (18) and (7)
(20) ~animal(w)                                    from (19) and (9)
(21) ~wombat(w)                                    from (20) and (5)
(22) []                                            from (21) and (9)
```

Example 2.4.7: Using unit-preference strategy

It can be seen from both examples 2.4.6 and 2.4.7 that a unit-preference strategy (13 resolvents) takes longer to reach the empty clause compared to a linear input resolution (9 resolvents). Chang[1970] prove that both linear input resolution and unit resolution (unit-preference strategy) are equivalent. That is, theorems that can be proved by linear input resolution can also be proved by unit-preference strategy, and vice versa.

2.4.8 LUSH Resolution

LUSH stands for "Linear resolution with Unrestricted Selection function for Horn clauses" (Hill [1974]). LUSH is same as a linear input resolution except that at each resolution step, the literal to be used as complement is selected from the last produced resolvent in a pre-defined order. The same order must be used throughout the resolution process once it was chosen. For example, suppose the order were "take the rightmost literal which has a complement in the input clause set S". The refutation search corresponding to example 2.4.6 above would proceed as follows:

```
(10) ~lives(w,zoo) ∨ ~happy(w)                          from (8) and (1)
(11) ~lives(w,zoo) ∨ ~animal(w) ∨ ~meets(w,Y) ∨ person(Y) ∨ kind(Y)
                                                        from (10) and (2)
(12) ~lives(w,zoo) ∨ ~animal(w) ∨ ~meets(w,Y) ∨ ~person(Y) ∨ ~visits(Y,zoo)
                                                        from (11) and (3) - after merging
(13) ~lives(w,zoo) ∨ ~animal(w) ∨ ~meets(w,naz) ∨ ~person(naz)
                                                        from (12) and (7)
(14) ~lives(w,zoo) ∨ ~animal(w) ∨ ~meets(w,naz)
                                                        from (13) and (6)
(15) ~lives(w,zoo) ∨ ~animal(w) ∨ ~person(naz) ∨ ~visits(naz,zoo)
                                                        from (14) and (4)
(16) ~lives(w,zoo) ∨ ~animal(w) ∨ ~person(naz)
                                                        from (15) and (7)
(17) ~lives(w,zoo) ∨ ~animal(w)                         from (16) and (6)
(18) ~lives(w,zoo) ∨ ~wombat(w)                         from (17) and (5)
(19) ~lives(w,zoo)                                      from (18) and (8)
(20) []                                                 from (19) and (9)
```

Example 2.4.8: Using LUSH resolution (selecting the rightmost literal)

This resolution is similar to a resolution known as SLD resolution which stands for "Linear resolution with selected function for Definite clauses" (Lloyd [1984]). LUSH resolution is complete for Horn clause (Frost [1986]). SLD is also a complete and sound resolution (Lloyd [1984]).

2.4.9 Other strategies and the combination of strategies

There are other variation of resolutions but most of them are basically the variation of semantic and linear resolution as discussed in sections 2.4.1 and 2.4.5 above. Such variations are selected literal (SL) resolution (Kowalski and Kuehner [1971]), Loveland [1969], Reiter [1971]), semantic resolution using ordered clauses (Chang and Lee [1973]), linear resolution using ordered clauses and the information of resolved literals (Chang and Lee[1973]).

It is also possible to combine the control strategies such as linear input resolution and set-of-support strategies. Some combination of the strategies preserve completeness, some don't.

As most resolutions produce refutation graphs, there are several ways of traversing the graphs. This traversing or also known as search strategies are usually depth-first strategy, breadth-first strategy and heuristic strategy. These strategies have also been combined with the resolution strategy to make the resolution more efficient. We will discussed a combined strategies implemented with breadth-first and depth-first search strategies in the next chapter (chapter 3).

2.5 Logic Programming

The key idea underlying logic programming is programming by description. The programmer describes the application area and lets the program choose specific operations. Logic programs are easier to create and enable machines to explain their results and actions. Logic programming differs fundamentally from conventional programming in requiring us to describe the logical structure of problem rather than making us prescribe how the computer is to go about solving them.

Logic programming which has begun in the early 1970's originated largely from advances in automatic theorem proving and artificial intelligence, and in particular from the development of the resolution principles. Constructing automated deduction systems is, of course, central to the aim of achieving artificial intelligence. Building on work of Herbrand[1930], there was much activity in theorem proving in early 1960's by Prawitz[1960], Gilmore[1960], Davis and Putnam[1960] and others. This effort culminated in 1965 with the publication of the landmark paper by Robinson [1965], which introduced an inference rule called the resolution rule which is particularly well-suited to automation on a computer.

Some of the earliest work relating resolution to computer programming was undertaken by Green[1969], who showed that the answer-extraction mechanism could be used for synthesizing conventional program by applying resolution to their specifications expressed in clausal-form logic, and also the works of Hayes[1973] and Sandwell[1973].

However, the credit for introduction of logic programming goes mainly to Kowalski[1974] and Colmerauer[1973]. In 1972, Kowalski and Colmerauer [1972] were led to the fundamental idea that logic can be used as a programming language. Before that (1972), logic had only ever been used as a specification or declarative language in computer science. However, what Kowalski[1974] showed is that logic has a procedural interpretation, which makes it very effective as a programming language.

One of the main ideas of logic programming , which is due to Kowalski [1979] and [1979a], is that an algorithm consists of two disjoint components, the logic and the control. The logic is the statement of what the problem is that has to be solved. The control is the statement of how it is to be solved. This relationship plays a central role in the philosophy of logic programming and can be expressed symbolically by the equation:

Algorithm = Logic + Control (A = L + C)

As we said in the beginning of this section, there are fundamental differences in logic and conventional programming. In traditional software engineering (conventional programming), one builds a program by specifying the operations to be performed in solving problem, that is, by saying how the problem is to be solved. The assumptions on which the program is based are usually left implicit. In logic programming, one constructs a program by describing its application are, that is, by saying what is true. The assumptions are explicit, but the choice of operations is implicit.

The difference can be expressed also in terms of Kowalski's idea of algorithm (A=L+C). So, conventional *algorithm* and programs expressed in conventional programming language combines the *logic* of the information to be used in solving problems with the *control* over the manner in which the information is put to use, i.e (following Hogger[1984]):

conventional program

= conventional algorithm (A)

= description of logic (L) and control (C).

On the other hand, logic programs express only the *logic component* L of algorithms . The *control component* C is exercised by the program executor, either following its own autonomously determined control decisions or else following control instructions provided by the programmer. In Hogger's [1984] formula:

logic program

= description of logic(L) + description of control(C)

There are several advantages to separate logic and control conceptually (Kowalski[1979]):

(1) Algorithms can be constructed by successive refinement, designing the logic component before the control component.

(2) Algorithms can be improved by improving their control component without changing the logic component at all.

(3) Algorithms can be generated from specifications, can be verified and can be transformed into more efficient ones, without considering the control component, by applying deductive inference rules to the logic component alone.

(4) Inexperienced programmers and database users can restrict their interaction with the computing system to the definition of the logic component, leaving the determination of the control component to the computer.

In a typical logic programming system, we can also view that the description of the control as an application-independent deductive inference procedure. Applying such a procedure to a description of an application are makes it possible for a machine to draw conclusions about the application are and to answer questions even though these answers are not explicitly recorded in the description. This capability is the basis for the technology of logic programming. The following figure illustrates a configuration of a typical logic programming system (Genesereth and Ginsberg [1985]):

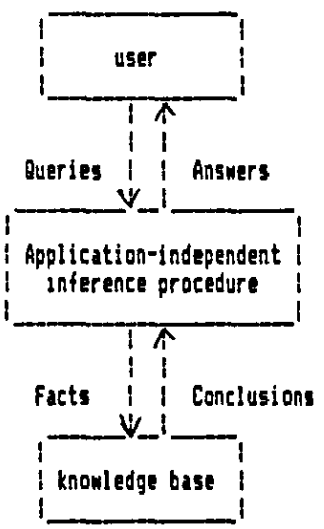


Fig 2.5.1: A Typical Logic Programming system

As shown in the above figure (fig 2.5.1), the application-independent inference procedure is independent of the knowledge base it access, thus it gives some advantages:

(1) Incremental development: as new information about an application is discovered (or just discovered), that information can added to the program's knowledge base and so incorporated into the program itself. There is no need for algorithm development or revision.

(2) Explanation:it is easy to save a record of the step taken in solving a problem due to the piecemeal nature of automated reasoning. This record can be presented to the user in some way such that the program explain how it solves each problem and therefore, why it believes the result to be correct. This is very valuable for debugging logic programming

Thus, from Kowalski's idea of algorithm, we can conclude that ideally, logic programming is that the programmer should only have to specify the logic component of an algorithm and the control should be exercised solely by the logic programming system. Unfortunately, this ideal has not yet been achieved with current logic programming systems (Lloyd[1984]). In order for this to be achieved there are two broad problemswhich have to be solved:

(1) Control problem: more control features for programmers should be the responsibility for the system itself. Currently the programmer has to provide a lot of control information such as clauses and atom ordering, looping checking etc.

(2) Negation problem: the logical negation is not implemented, but they implement the negation as a failure rule in logic programming languages. A negation as a failure rule means that the negation of a goal (statement) is true if the positive of the goal fails. Clark[1978] and Reiter[1978] have discussed

this problem and have regarded the negation as failure inference rule as deductions from the "completed data base"(CDB) and "closed world assumptions"(CWA) respectively. According to Shepherdson[1984] these deduction system are usually incomplete and CDB and CWA differs: one may be consistent and the other not, and when both are consistent they may be incompatible and both are compatable when the data base is Horn and definite clauses.

As discussed above and following Kowalski[1974], logic has a procedural interpretation which makes it very effective as a programming language and one of the most important practical outcomes of the research so far has been the language PROLOG, which is based on the Horn clause subset of logic. In addition to procedural interpretation, logic also has two other interpretation (Lloyd[1984]):

(b) Database interpretation: logic program is regarded as a database (Lloyd[1983]; Gallaire and Minker[1978] and [1981]) thus we obtain very natural and powerful generalisation of relational databases.

(c) Process interpretation: goal <-B1,...,Bn is regarded as a system of concurrent process. There are now several concurrent PROLOGs based on the process interpretation (Clark and Gregory[1981] and [1983]; Shapiro[1983]). This interpretation allows logic to be used for operating system applications and object-orientated programming (Shapiro and Takeuchi[1983]).

We have already discussed where we can use logic programming and its difference between conventional programming. Most logic programming system use clausal form. However, logic programming is by no means limited to PROLOG which is based

on Horn clauses. It is essential not only to find more appropriate computation rules, but also to find ways to program in larger subsets of logic, not just the clausal subset. In particular, such systems need not necessarily be based on clausal resolution or even resolution at all.

Various non-clausal resolution have been developed, for example, Storm [1974], Wilkins [1974], Bibel [1976], Nilsson [1979], Manna and Waldinger[1980], Bowen [1982], Murray [1982] and Stickel[1982]. Other methods of non-resolution theorem-proving are such as natural deduction (Bledsoe [1977], Hanson et. al.[1982], and, Haridi and Sahlin[1983]) and matrices and connections (Prawitz [1976], Andrews [1981] and Bibel[1983]).

It is clear that logic thus provides a single formalism for apparently diverse parts of computer science. Logic provides us with a general purpose problem-solving language, a foundation for database systems, and also a concurrent language suitable for operating systems and parallel algorithm. This range of applications assures that logical inference is about to become the fundamental unit of computation (Lloyd[1984]). This view is strongly supported by the Japanese fifth generation computer project where logic programming has been chosen to provide the core programming language for this very ambitious 10 years project (Moto-Oka[1982]).

2.6 A Logic Programming Language: PROLOG

The language PROLOG, stands for "PROgramming in LOGic", is one of the most practical outcomes of researches in logic programming. The early developers of this idea included Robert Kowalski at Edinburgh (on theoretical side - see Kowalski [1974]) and Alain Colmerauer at Marseilles (implementation- see Colmerauer et. al.[1973]). The present popularity of Prolog is largely due to David Warren's efficient implementation at Edinburgh in mid 1970s. Nowadays, various Prolog interpreters or compilers have been implemented, such as Quintus Prolog (Quintus Ref. Manual [1985]), C-Prolog (Pereira [1982]), POPLOG Prolog (Mellish and Hardy[1984]) etc (for example, see Clark and Tarnlund[1982], Campbell[1984]).

The research in this thesis was first carried out by using the UNIX Prolog in Edinburgh syntax (see Pereira et al.[1978]) and later by using POPLOG Prolog which has more facilities although similar syntax. Although there are some differences, they will not be explained here.

Now, we will describe briefly the syntax of Prolog (based on Edinburgh syntax) and also some problems encountered with the Prolog during the carrying out of the project. As said in the last section that Prolog is based on the Predicate calculus and takes the form of Horn clauses. The following are summaries of Prolog syntax's (for more details see Prolog books such as Clocksin and Mellish[1981] and Bratko[1986]):

- Prolog programming consists of defining relations and querying about relations.

- Prolog programs are built from *terms* which are either a *constant*, a *variable* or a *structure*. A constant consists of a number or an *atom* (a symbol which starts with a lower case letter). A variable looks like atom, except they begin with a capital letter or an underline sign "_". A structure which is a single object that have several components, are constructed by means of *functors*. Each functor is defined by its name (or *predicate* name) and arity.
- Prolog clauses are of the form:

  $$a :- b_1, \ldots, b_n$$

  where "a", and "$b_1, \ldots, b_n$" are called *head*, *body*. The sign ":-" and "," mean "*is implied by*" and "*and*"*(conjunction)*. Thus they can be classified to three types: *facts*, *rules* and *questions*. If n=0 then the clause is called a fact and can be written without implication sign. If n>0 then it is called a rule. If the head is an empty clause (i.e $[]:-b_1, \ldots, b_n$) then it is called a question or a *goal* clause.
- A *procedure* is a set of clauses about the same relation, i.e a set of clauses with the same predicate name.

Prolog's refutation is based on SLD resolution, i.e the literal to be matched (or unified) is always selected from the first one in the goal clause. Basically, Prolog adopts a depth-first strategy that is the new goals derived from the use of a clause are placed at the front of the (current) goal clause and Prolog finished satisfying a subgoal before it goes on to try anything else. Prolog will do *backtracking* in order to explore the alternative of the solution. Prolog also does not carry out the occurs check. Although it may

give wrong answers (see Lloyd[1984] pp40), it may also show a sign of programming error.

Prolog ,in executing the goal clause, will make a copy of a clause to be matched and all variables in it will be given new variables names. This copying clause principle in Prolog is equivalent to the process of standardizing variables names in converting a predicate calculus into clausal form.

Prolog program can be written such that it may become a reversible (inversible) program (Gries[1981]). A reversible program is a program of which we can get the input again from the output. Although there is not an easy task, Prolog in some cases, can do it, for example, the predicate *append(X,Y,Z)* which it will append list Y to the end of X and give the result list Z. The definition of *append(X,Y,Z)* is as follows:

```
append([],L,L).
append([X/L1],L2,[X/L3]):-append(L1,L2,L3).
```

If given X is an original list and Y is the input list to be appended to original list X, then we will get the output list, Z. The above program can also be used to get the input list Y if given the output list Z. The following session illustrates the feature of reversible program:

```
?- append([a,b,c],[d,e,f],Z),append([a,b,c],Y,Z).

Z = [a,b,c,d,e,f]
Y = [d,e,f]

yes
```

Session 2.6.1: Example of a reversible program

Another advantage of Prolog language that we can easily write a Prolog program in any languages which uses a Roman alphabet. For example:

```
eats(nazrul,apple).     /* English */
manger(nazrul,pomme).   /* German */
makan(nazrul,epal).     /* Malay */
```

All other built-in predicates can be defined in the equivalent non English language, for example, predicate *write(X)* can be defined in a non English language as follows:

```
tulis(X):-write(X).     /* Malay's definition of write */
```

Thus, Prolog can be written in any Roman-alphabet language without difficulty.

We will discuss in the following some of the problems we have faced in the carrying out of the project:

[1]. predicate *not/1*

As Prolog adopts a negation as a failure, so predicate *not/1* will not mean the real logical negation. In order to make Prolog's program more readable especially in testing a conditional predicate, i.e a predicate which is used to represent a certain condition. Suppose, we have a condition predicate *newsubsidiary* which indicates that new subsidiary clauses have been asserted into the database. Instead of using *not(newsubsidiary)* or *newsubsidiary* to show the non existence or the existence of predicate *newsubsidiary* respectively, we will use a new defined predicate *exists(X)* or *not_exists(X)* to show the existence or non existence of predicate (or fact) X. Predicates *exists/1* and *not_existence/1* are defined as follows:

```
exists(X) :- clause(X,true).

not_exists(X) :- clause(X,true),!,fail
not_exists(X) :- true.
```

So, *exists(newsubsidiary)* and *not_exists(newsubsidiary)*

will be used to test the existence or the non existence of fact *newsubsidiary*.

[2]. Hardcopy.

It would be easy to make a hardcopy of a session in POPLOG Prolog as there are built in predicates to cater for this, i.e predicate *log* and *nolog*. However it would be very difficult to make a hardcopy of a session by using UNIX Prolog as it will make a hardcopy but we will see nothing in terminal (VDU) or vice versa. To make a hardcopy in DEC-10 Prolog, we need to use predicates *tell(Filename)* to start of making a hardcopy and *told* when it is finish and to close file "Filename". For example:

```
write_answer(user,yes):-!,      /* 1st definition */
        nl,write('>>Answer: Yes').
write_answer(F):-               /* 2nd definition */
        write_answer(user,yes),
        tell(F),write_answer(F,yes).
```

The first definition of predicate *write_answer/2* will print the remark in the terminal and the second definition will print the remark in both terminal and the output file F provided F is not equal to "user". In order for the second definition to work, we must instruct the Prolog that the output file is F by command *tell(F)* at the beginning of a session and closed the file by issuing command *told* at the end of session.

Although there are problems of negation and the occur check, Prolog has been used in various fields such as expert systems, natural language understanding etc. (see Clark and Tarnlund[1982], Sterling and Shapiro[1986], for examples).

CHAPTER 3

A PROLOG-BASED RESOLUTION

3.1 Introduction

In this Prolog-based resolution, all the statements which are already in the form of a first order logic or predicate calculus, are transformed into Horn Clauses. The resolution is based mainly on how resolution in Prolog is implemented. The resolution strategy in Prolog is a kind of semantic linear (SL) input resolution. Basically, Prolog adopts a depth-first strategy. It means also that no occur check is implemented in the implemented resolution.

There are two major steps in Prolog-based resolution. These steps are

[1]. Converting a predicate calculus statement into Horn clauses.

[2]. The refutation process.

3.2 Converting a predicate calculus statement into Horn Clauses

First of all, the predicate calculus statements are converted into Horn Clauses in seven stages. The first six stages which are based on Clocksin and Mellish[1981] are to convert a predicate calculus formula into clausal form. So the associated Prolog program for the first six stages will not be explained here except if it is different or any modification is made, otherwise the programs can be found in the appendix. Before the conversion technique is described, let us define the syntax that will be used as follows:

Connective	PC syntax	The syntax used	Meaning
negation	~a	~a	not a
conjunction	a $\wedge$ b	a & b	a and b
disjunction	a $\vee$ b	a # b	a or b
implication	a --> b	a => b	a implies b
equivalence	a <--> b	a <=> b	a is equivalent to b

TABLE 3.2.1: Syntax definition (connective)

And for quantifiers , the following syntax will be used:

PC syntax	The syntax used	Meaning
$\forall$X,a	all(X,a)	For all X, a is true
$\exists$X,a	exists(X,a)	There exists X such that a is true

TABLE 3.2.2: Syntax definition (quantifier)

For the purpose of operator precedence, the following Prolog's operators declaration are defined for the connectives:

```
?-op(225,xfx,<=>).
?-op(225,xfx,=>).
?-op(200,xfx,&).
?-op(200,xfx,#).
?-op(30,fx,~).
```

Program 3.2.1: Operator Declaration

The seven stages of converting a predicate calculus statement into Horn clauses are as follows:

[1]. Removing all implication and equivalence signs.

[2]. Moving negation signs inward.

[3]. Skolemising existential quantifiers.

[4]. Moving outwards and eliminating universal quantifiers.

[5]. Distributing conjuction(&) over disjunction(#) signs.

[6]. Putting into a clausal form.

[7]. Converting into Horn clauses.

The following are the detail explaination of each stage.

STAGE 1: Removing all implication and equivalence signs

Replace all occurrences of "=>" and "<=>" signs by using the following rules:

Formula	The Replacement formula
a <=> b	(a => b)&(b => a)
a => b	(~a # b)

TABLE 3.2.3: The Replacement formula

e.g:

```
all(X,(man(X)#woman(X))=>human(X))
```

is transformed to

```
all(X,~(man(X) # woman(X)) # human(X)).
```

STAGE 2: Moving negation sign inwards

This stage is involved with cases where the negation sign, " ~ ", is applied to a formula that is not atomic. The end products are such that only the negation sign is applied to an atomic formula by reducing the scopes of "~" sign using the appropriate De Morgan's law as follows:

The formula	The reduction formula
~(a # b)	~a & ~b
~(a & b)	~a # ~b
~(~a)	a
~exists(X,a)	all(X,~a)
~all(X,a)	exists(X,~a)

TABLE 3.2.4: The reduction formula

For example,

```
~exists(X,wombat(X) & lives(X,zoo) & happy(X))
```

is transformed into

```
all(X, ~wombat(X) # ~lives(X,zoo) # ~happy(X))
```

STAGE 3: Skolemising

Skolemising is a stage to eliminate existential quantifiers by replacing all existentially quantified variables by a Skolem function with parameters of the universally quantified variables which bind it. In other words, instead of saying that there exists an object with a certain set of properties, one can create a name for one such object and simply say that it has the properties. Skolemisation has one important property, i.e there is an interpretation for the symbols of a formula that makes the formula true if and only if there is an interpretation for the Skolemised version of the formula.

The general rule for eliminating an existential quantifiers from a well form formula (wff) is to replace each occurrence of its existentially quantified variable by a Skolem function whose arguments are those universally quantified variables that are bound by universal quantifiers whose scope include the scope of the existential quantifiers being eliminated. For example,

```
all(X,man(X)&exists(Y,~woman(Y) # likes(X,Y)))
```

is Skolemised into

```
all(X,man(X)& (~woman(woman0(X)) # likes(X,woman0(X)))
```

Here, the Skolem function which replaces the existential quantifier ,Y, is "woman0(X)" where X is a universal quantifier which binds existential quantifier Y. The index 0 (zero) of "woman0" is to distinguish between other Skolem function which refers to a different woman, for instance, "woman0(X)" and "woman1(X)" refer to two

different women. Any function symbol used in Skolem functions must be new in the sense that they cannot be the one that already occurs in wffs.

If the existential quantifier being eliminated is not within the scope of any universal quantifiers, we use a Skolem function with no arguments, which in fact is a Skolem constant.

e.g.

exists(X,animal(X) & eats(X,human))

is Skolemised into

animal(animal0) &eats(animal0,human)

STAGE 4: Moving outwards and eliminating universal quantifiers

All universal quantifiers bind their own dummy variables, so move outwards all universal quantifiers to the front of the wff. The resulting wff is said to be in a prenex form. This does not effect the meaning. For example,

all(X, ~man(X)#all(Y, ~woman(Y)#likes(X,Y)))

is transformed into

all(X,all(Y, ~man(X)#(~woman(Y)#likes(X,Y))))

Since the order of universal quantifications is unimportant provided that all existential quantifiers have been Skolemised, and all universal quantifiers are at the front or outside of the wff, so we may eliminate the explicit occurrence of universal quantifiers without loss of meaning. Thus the above example becomes

~man(X)#(~woman(Y)#likes(X,Y)).

It should be noted here that no renaming universal quantifiers or variables is required as Prolog will do it automatically.

STAGE 5: Distributing "&" over "#"

In other words, at this stage, the wff is transformed into conjunctive normal form by using the following transformation rules:

Formula	The Distribution formula
a # (b & c)	(a # b) & (a # c)
(a & b) # c	(a # c) & (b # c)

TABLE 3.2.5: The Distribution Formula

e.g.

```
(~ man(X) & ~ woman(X)) # human(X)
```

is transformed into

```
(~man(X) # human(X)) & (~woman(X) # human(X))
```

STAGE 6: Putting into a clausal form

At the beginning of this stage, all wffs are in the conjunctive normal form. The "&" sign may be eliminated resulting a finite set of wffs where each of wffs is made up of literals joined by disjunctions ("#"). Any wff which consists of disjunction of literals is called a clause. Any clause which contains both negated and unnegated of the same literal are left out, since this clause is a tautology, therefore the clause is trivially true and contributes nothing.

Each clause will be written as cl(A,B) where A is a collection of unnegated literals and B is a collection of negated literals but without negation "~" sign. For example,

```
(~man(X)#human(X)) & (~woman(X)#human(X))
```

is written as

```
cl([human(X)],[man(X)]).
cl([human(X)],[woman(X)]).
```

STAGE 7: Converting into Horn Clauses

All clauses are already in the form of cl(A,B). We will use the convention to write cl(A,B) as described in Clocksin & Mellish [1981]. All literals in A and B will be written separated by semicolon ";" and comma "," respectively. Set of literals in A and B will be separated with ":-" sign. If A and B consist of unnegated literals a,b,... and negated literals k,l,... respectively, then the clause will be written as

```
a;b;... :- k,l,... .
```

In fact the signs ":-", ";" and "," are just like Prolog's syntax and their meaning are "is implied by", "or" and "and" respectively. For example, "a;b :- k,l" is read as "if k and l then a or b" or "a or b is implied by k and l". Set A and B are called the *head* (or the *conclusion*) and the *body* (or the *condition*) of the clause. For examples:

[S7.1]. cl([],[wombat(X),lives(X,zoo),happy(X)]).

is written as

```
[]:-wombat(X),lives(X,zoo),happy(X).
```

(i.e for all X, it is not the case that wombat(X) and lives(X,zoo) and happy(X) OR no wombat who lives in a zoo is happy).

[S7.2]. cl([human(X)],[man(X)])

is written as

```
human(X):-man(X).
```

(i.e human(X) is implied by man(X)).

[S7.3]. cl([alive(X),dead(X)],[])

is written as

```
alive(X);dead(X).
```

(i.e Either alive(X) or dead(X) is true)

[S7.4]. [cl([first_man(adam)],[]),cl([first_woman(eve),[])]

is written as

```
first_man(adam).
first_woman(eve).
```

(i.e adam and eve are the first man and woman respectively)

[S7.5]. cl([sad(joe),angry(joe)],
[offday(today),raining(today)])

is written as

```
sad(joe);angry(joe):- offday(today),raining(today)
```

(i.e if today is offday and it is raining then joe is sad or angry).

Since we are only interested in Horn clauses, the above clauses will be converted into Horn clauses. As described before that Horn clause is either a headless or a headed clause. Furthermore a headed clause consists only one unnegated literal. Where as A (i.e set of unnegated literals) or the head of clause may contains none, one or more than one unnegated literal. So examples [S7.3] and [S7.5] above are not Horn clauses, and example [S7.1] is a headless Horn clause while examples [S7.2] and [S7.4] are already headed Horn clauses . However all Prolog rules are in the form of headed Horn clauses, thus we will transform all clauses into headed Horn clauses.

In order to transform the clauses into headed Horn clauses, set A must contain only one literal. This can be achieved by transferring either the extra literal(s) of set A to B or one literal from set B to set A. By doing so, the sign of the transferred literal or literals must be changed i.e by putting a negation "~" sign in the front of transferred literal(s) since ~(~a) is equivalent with a. For instance, example [S7.1] above:

```
[]:-wombat(X),lives(X,zoo),happy(X)
```

is equivalent to (or can be written as)

```
~wombat(X):-lives(X,zoo),happy(X).
~lives(X,zoo):-happy(X),wombat(X).
~happy(X):-wombat(X),lives(X,zoo).
```

Since Horn clauses are like Prolog rules or facts, the negated sign will cause problem if it is straight fowardly implemented in Prolog. This is due to the fact that the operator "not" in Prolog is not exactly equivalent to the negation "~", i.e their meanings are not equivalent. For instance, the meaning of "not(man(mary))" in Prolog is different to "~man(mary)". The first one refers to the fact that either mary is not a man or an fact "man(mary)" does not exist in Prolog's database, while the latter only means that mary is not a man. This is due to the definition of operator "not" in Prolog which is defined as follows:

```
not(P):-call(P),!,fail.
not(P):-true.
```

To overcome this problem, the operator "~" which has already been defined, is used in order to make it as valid rules of Prolog, thus the above examples will be accepted as Prolog's rules (Note: for the purpose of tracing, use 'spy ~').

It should be noted here that by doing so, the system becomes an open world system. This is due to fact that in order to prove a negated fact, the negated fact should exist in the database, otherwise the system will tell that the questioned fact does not exist in the database or it cannot prove or deduce it from the database. This is a contrast to a closed world system whereby we prove the negation by trying to prove its counterpart, i.e. the nonexistence or nondeducability of its positive fact.

In addition, all possible combinations of Horn clauses i.e with a different head or conclusion literal, will be generated since we would like it to work just like Prolog's rules or facts. If altogether there are N literals in A and B, then N Horn clauses will be generated. For examples:

[S7.2']. human(X):-man(X)

is transformed into two Horn clauses:

(a). human(X):-man(X).
(b). ~man(X):-~human(X).

(i.e (b) means that if human(X) is false then man(X) is false or if X is not a human then X is not a man).

[S7.3']. alive(X);dead(X)

is transformed into two Horn clauses:

alive(X):-~dead(X).
dead(X):-~alive(X).

(i.e The first one means that if dead(X) is false then alive(X) is true. The second one means that if alive(X) is false then dead(X) is true).

[S7.5'] sad(joe);angry(joe):- offday(today),raining(today).

is transformed into four Horn clauses:

sad(joe):- offday(today),raining(today), ~angry(joe).
angry(joe):- offday(today),raining(today), ~sad(joe).
~offday(today):-raining(today),~sad(joe),~angry(joe).
~raining(today):-~sad(joe),~angry(joe),offday(today).

It should be noted here that clause [S7.2'](b) above is also called as the *contrapositive* form of clause [S7.2'](a) which is useful for backward production systems (Nilsson[1980]).

The top procedure to convert all clauses into headed Horn clauses is carried out by procedure HC or predicate *horn_clauses/2* as shown in the Program 3.2.2 below:

```
/* Procedure HC :converting a clausal into headed Horn clause */
horn_clauses([],[]):-!.
horn_clauses([Clausal1|Clausal2],Headedhorn):-!,
        horn_clauses1(Clausal1,Horn1),  /* procedure HC1 */
        horn_clauses(Clausal,Horn2),
        append(Horn1,Horn2,Headedhorn),!.
```

Program 3.2.2: Procedure HC

So, the above procedure HC will convert a set of of all clauses in the form of "Head:-Body" (variable "Clausal1") into headed Horn clauses. Each clause is transformed into equivalent headed Horn clauses by calling predicate *horn_clause1/2* or procedure HC1 which as shown in the following program (Program 3.2.3):

```
/* Procedure HC1 :creating H equivalence headed Horn clauses */
horn_clauses1([]:-Body,C):-
        /* procedure HC1.1 */
        horn_clauses2(Body,[],C),!.     /* procedure HC2 */
horn_clauses1(Head:-Body,C):-
        /* procedure HC1.2 */
        !,
        convert(Head,Body1),            /* procedure CONV */
        appendbody(Body1,Body,Body2),   /* procedure APPB */
        horn_clauses1([]:-Body2,C).
horn_clauses1((Head;Head1),C):-
        /* procedure HC1.3 */
        !,
        convert((Head;Head1),Body),
        horn_clauses1([]:-Body,C).
horn_clauses1(Literal,[Literal]).       /* procedure HC1.4 */
```

Program 3.2.3: Procedure HC1

As shown above (Program 3.2.3), procedure HC1 is classified into four subprocedures. Before a final conversion into headed Horn clauses is done, all clauses of the form "Head:-Body" will be converted into "[]:-Body1". Procedure HC.1 will test whether the set "Head" is already empty, i.e "[]:-Body", (see example [S7.1] above) and will do nothing.

Procedures HC1.2 and HC1.3 will first convert clauses of type "a;b;...:-k,l,..." (see examples [S7.2] and [S7.5] above) and "a;b;..." (see example [S7.3] above) respectively into the form of "[]:-m,n,...." before calling procedure HC2. If a clause is already in the form of "a" or it consists only one literal (see example [S7.4]), then nothing is done (procedure HC1.4).

After all clauses have been changed into headless clauses, then procedure HC2 will generate headed Horn clauses. Procedure HC2 which will generate N equivalent headed Horn clauses where N is a number of literals in each clause, is also subdivided into four subprocedures (see Program 3.2.4 below).

```
/* procedure HC2 */
horn_clauses2([],Body,[]):-
        /* procedure HC2.1 */
        !.
horn_clauses2((Body,Body1),Body2,[Hothead:-Body3|H]):-
        /* procedure HC2.2 */
        !,
        convert(Body,Hothead),
        appendbody(Body2,Body1,Body3),
        appendbody(Body2,Body,Body4),
        horn_clauses2(Body1,Body4,H).
horn_clauses2(Body,[],[Hothead]):-
        /* procedure HC2.3 */
        !,
        convert(Body,Hothead).
horn_clauses2(Body,Body2,[Hothead:-Body2]):-
        /* procedure HC2.4 */
        !,
        convert(Body,Hothead).
```

Program 3.2.4: Procedure HC2

As shown in the above Program 3.2.4, the first procedure HC2.1 is the ending procedure when there is no more literals to be transformed into headed Horn clauses. All clauses are type of "[]:-k,l,m,..." is passed from procedure HC1 to procedure HC2.

Procedure HC2.2 will first move literal k into the right hand side (RHS) of the rule and generate clause "~k:-l,m,...". This process is continued by moving one literal at a time into the RHS of the rule until N headed Horn clauses are generated.

Procedure HC2.4 is the ending of subprocess of procedure HC2.2. When there is only one literal in the clause, for example, "[]:-k", then procedure HC2.3 will convert it into clause "~k". Procedures HC2.3 and HC2.4 are in fact equivalent except that procedure HC2.3 is to form a clause of "~k" instead of "~k:-[]".

All literals of either the head or the body of clauses is moved from the left-hand side (LHS) or the right-hand side (RHS) to the opposite side of the rule by procedure CONV or predicate *convert/2* as shown in the following program (Program 3.2.5).

```
/* procedure CONV moving literals into the left hand side *
 *     or the right hand side of a rule as appropriate    */
convert((A;B),(A1,B1)):-
        /* procedure CONV.1 move literal(s) into the RHS of a rule */
        !,convert(A,A1),
        convert(B,B1).
convert((K,L),(K1;L1)):-
        /* procedure CONV.2 move literal(s) into the LHS of a rule */
        !,convert(K,K1),
        convert(L,L1).
convert(~A,A):-!.       /* procedure CONV.3 */
convert(A,~A).          /* procedure CONV.4 */
```

Program 3.2.5: Procedures CONV

Procedure CONV.3 (as shown above) will take away "~" sign from the literal if there already exists, otherwise the sign "~" will be added in the front of it if there does not already exist (procedure CONV.4) when moving it from one side of the rule into the other side.

3.3 The refutation process

As we said earlier that the implemented refutation system is almost entirely based on the refutation in Prolog as the program is written in Prolog. In the last section we have described how first order logic statements are transformed into Horn clauses. These clauses are actually facts which will be used to prove a hypothesis or to answer a question. In Prolog, a collection of facts is called a database. We can divide the refutation procedure into three stages, i.e

(1). Setting up a database (knowledge base).

(2). Formatting a goal.

(3). The refutation procedure.

3.3.1 Setting up a database (knowledge base).

A database consists a collection of Horn clauses which are generated from known knowledge as well as from questions and hypotheses. These clauses are called *knowledge base* clauses (KB clauses). KB clauses are made up from two types of clauses. The first one is a collection of clauses resulting from the negation of questions or hypotheses, i.e *query* clauses. And the other is a collection of axioms or known clauses, i.e *knowledge* clauses. Thus the definition of KB clauses is as follows:

```
knowledge_base(X):-
        query(X).       /* query clauses */
knowledge_base(X):-
        knowledge(X).   /* knowledge clauses */
```

Program 3.3.1.1: Definition of knowledge base (KB) clauses

KB clauses can be categorised into two classes, i.e *factual* and *ruled* KB clauses. Factual and ruled KB clauses are clauses without and with body respectively, for instance, the first following three are factual KB clauses and the rest are ruled KB clauses.

```
knowledge(person(nazrul)).          /* a factual KB clause */
knowledge(boy(aizat)).              /* a factual KB clause */
query(~girl(nazrul)).               /* a factual KB clause */
knowledge(human(X):-man(X)).        /* a ruled KB clause */
query(likes(X,Y):-man(X),woman(Y)). /* a ruled KB clause */
```

As we need these two different types of KB clauses (i.e query and knowledge clauses), we have to split the process of converting a predicate calculus (PC) statement into three subprocedures. There are procedures TRTOP, TRBOT and Skolemisation of existential quantifiers for knowledge statements (knowledge clauses) and questions (query clauses). As shown in Program 3.3.1.2, procedure TRTOP or predicate *translate_top* consists first two stages, i.e removing all implication and equivalence signs, and then, moving in negation signs.

```
translate_top(X,X2):-
        implout(X,X1),  /* STAGE 1 */
        negin(X1,X2).   /* STAGE 2 */
```

Program 3.3.1.2: Procedure TRTOP

Procedure TRBOT or predicate *translate_bottom/2* consists of Stage 4 up to stage 6 and also a part of Stage 7, i.e. the process of printing clauses in Clocksin format (see Program 3.3.1.3 below).

```
translate_bottom(X3,Clause):-
        univout(X3,X4),             /* STAGE 4 */
        conjn(X4,X5),               /* STAGE 5 */
        clausify(X5,X6,[])          /* STAGE 6 */
        buildclauses(X6,Clause),    /* STAGE 7a */
        printclauses(Clause).       /* printing Clause */
```

Program 3.3.1.3: Procedure TRBOT

The Skolemisation of existential quantifiers for each type of clauses (knowledge and query) will be explained accordingly in the following sections.

3.3.1.1 Knowledge clauses

Knowledge clauses result from the transformation of axioms or known knowledge. These Horn clauses are asserted in the Prolog system as facts by adopting them as variables or arguments of a predicate, namely *knowledge*. So predicate *knowledge/1* is a fact and its object is a generated Horn clause. For examples, the following generated Horn clauses:

```
human(X):-man(X)
~man(X):-~human(X)
animal(fs(animal0))
```

are asserted in a Prolog's database as follows:

```
knowledge(human(X):-man(X)).
knowledge(~man(X):-~human(X)).
knowledge(animal(fs(animal0))).
```

It should be noted that "fs" is taken as Skolem function indicator for all knowledge clauses. This is done by defining predicate *skolem/3* or procedure SKOLEM (see Appendix) to Skolemised existential quantifiers for all knowledge clauses. Then procedure PICKSK or predicate *pickskolem/3* (see Program 3.3.1.4) is called from procedure SKOLEM to pick a symbol's name for a Skolem function in replacing the existential quantifiers.

```
pickskolem(Name,Vars,Sk):-
        gensym(Name,F),          /* procedure GENSYM */
        append([F],Vars,Fandargs),
        Sk=..[fs|Fandargs].
```

Program 3.3.1.4: Procedure PICKSK

Procedure GENSYM or predicate *gensym/2* (see Appendix) will generate a symbol name for Skolemised existential quantifier and variable "Sk" is a Skolem function or variable. Thus "fs(animal0)" in the above example is a Skolem constant for an animal.

In general that if H is a Horn clause generated from axioms or knowledge statements, then *knowledge(H)* is asserted in the database as facts.

```
pcfact:-
        read(S),                /* read a PC statement */
        assert_knowledge(S).    /* procedure ASSKG */
```

Program 3.3.1.5: Procedure PCFACT

Procedure PCFACT or predicate *pcfact/0* (see the above Program 3.3.1.5) will read a predicate calculus (PC) statement, S, and assert its generated Horn clauses into the database by procedure ASSKG or predicate *assert_knowledge/1* as shown in the following program (Program 3.3.1.6).

```
/* Procedure ASSKG :to assert knowledge clauses into the database */
assert_knowledge(S):-
        translate(S,Clause),            /* procedure TRPCK */
        horn_clauses(Clause,Horn),      /* procedure HC */
        stassertz(Horn).

/* Procedure TRPCK: to translate PC, S, into Clocksin format, Clause */
translate(S,Clause):-
        translate_top(S,X2),
        skolem(X2,X3,[]),       /* Procedure SKOLEM :Stage 3 for knowledge clauses */
        translate_botton(X3,Clause).

/* Procedure STASZ :assert "knowledge(H)" in the database  */
stassertz([]):-!.
stassertz([H|T]):-
        assertz(knowledge(H)),
        stassertz(T).
```

Program 3.3.1.6: Asserting knowledge clauses into the database

Procedure ASSKG will firstly translate the PC statement S into clauses of Clocksin format by procedure TRPCK or predicate *translate/2* (see Program 3.3.1.6). After that the result (variable "Clause") is passed to procedure HC (see section 3.2 Stage 7) in order to transform them into headed Horn clauses. Finally the headed Horn clauses are asserted in the database (procedure STASZ or predicate *stassertz/1* - see also Program 3.3.1.6).

The following is an example of Prolog's session which shows how to assert knowledge clauses by using the predicate *pcfact/0* or procedure PCFACT.

```
?-pcfact.
  |: all(X,man(X)=>human(X)).

  The tanslated clauses:

  human(_1) :- man(_1),
  &&&&&&&&& nextnextnext &&&&&&&&&&

  yes

?-listing(knowledge).

  knowledge(human(_1):-man(_1)).
  knowledge(~man(_1):- ~human(_1)).

  yes
```

Session 3.3.1.1: Asserting knowledge clauses in the database.

After procedure PCFACT is successfully called, then we can see the listing of knowledge (KB) clauses in the database as shown in the above session (Session 3.3.1.1).

3.3.1.2 Query clauses.

Query clauses result from the transformation of the negation of questions or hypotheses to be proved. In order to distinguish between knowledge and query clauses, query clauses are asserted into the database with predicate *query/1*. This will also make it easier to retract all query clauses from the database as they will not be permanently stored in the database. For example, the following query clauses:

```
wombat(fq(wombat0))
lives(fq(wombat0),zoo))
```

are asserted into the database as follows:

```
query(wombat(fq(wombat0))).
query(lives(fq(wombat0),zoo)).
```

Here, "fq" is a Skolem function indicator for all query clauses. This is to differentiate with the knowledge's Skolem indicator "fs". Since knowledge clauses can be permanently kept in a file, and at any time the file can be consulted in Prolog session before proving a new statement, then the same index of Skolem variable will probably be generated. To overcome this problem, two different symbols are used to represent query and knowledge Skolem function indicators, i.e "fq" and "fs" respectively.

Although, it can be overcome by keeping the log of indices of Skolem functions used in the KB clauses, but this technique does not overcome the most important difference. The most important difference is that Skolem function of query clause is actually a result of the negation of universal quantifiers of a question. So it

does not only correspond to the meaning of replacing existentially quantified with a Skolem function, but it also serves a different purpose, i.e the query's Skolem function is only to match or unify with any variable or with the same Skolem function of KB clauses or otherwise it defeats the purpose of the negation of a question. Therefore the unification between query and knowledge Skolem functions are avoided. For instance, suppose the database contains the following clauses:

```
knowledge(wombat(fs(wombat0))).
knowledge(lives(fs(wombat0),zoo)).
```

which means that there exists a wombat who lives in a zoo, i.e *exists(X,wombat(X)&lives(X,zoo))*. And also suppose the question is "do all wombats live in a zoo?", thus the result of the negation of the question, i.e. *all(X,wombat(X)=>lives(X,zoo))*, becomes:

```
query(wombat(fq(wombat0))).
query(~lives(fq(wombat0),zoo)).
```

Indeed, "~lives(fq(wombat0),zoo)" of query clauses cannot be resolved with "lives(fs(wombat0),zoo)" of knowledge clauses to produce an empty clause due to the fact that "fq(wombat0))" cannot be unified with "fs(wombat0)". So the proof is an unsuccessful one, or we can say that not all wombats live in a zoo. In this case, "fs(wombat0)" refers to only one particular wombat who lives in the zoo (knowledge statement); on the contrary "fq(wombat0)" refers to all wombats whom we like to know whether all of them live in a zoo.

Nilsson[1980] pointed out that in the answer-extracting process, it is correct to replace any Skolem functions in the clauses coming from the negation of the goal wff by

new variables. This is not true in this case, as if we replace the Skolem function "fq(wombat0)" of query clauses with a variable, let say Y, to become:

```
query(wombat(Y)).
query(~lives(Y,zoo)).
```

So, "~lives(Y,zoo)" of query clauses will be resolved with "lives(fs(wombat0),zoo)" of knowledge clauses to produce an empty clause where variable Y is unified with "fs(wombat0)". This means that the proof is successful or we can say that all wombats live in a zoo. The irony lay in the fact that the refutation process produces a wrong answer, although it ends up with an empty clause. The wrong answer is that all wombats live in a zoo (except that if there is only one wombat in the world, i.e wombat "fs(wombat0)") as we know that the contrary is a true one. The arbitrary replacement of Skolem function with a variable certainly can not be adopted here as the refutation system which will be described later will not only be used with the answer-extraction process only, but it is also used to prove a hypothesis.

There are, however, certain cases where the Skolem function of questions causes a somewhat obtuse answer. For instance, let the database contains the following knowledge clauses which means that all men love Mary, i.e *all(X,man(X) => loves(X,mary))*:

```
knowledge(loves(X,Mary):-man(X)).
knowledge(~man(X):- ~loves(X,Mary)).
```

and, let the question is that "do all men love Mary?" i.e *all(Y,man(Y)=>loves(Y,mary))*. By saving the explanation of refutation procedure for the next section, let see the following session:

```
?-pcquest.
|: all(Y,man(Y)=>loves(Y,mary)).

The translation of its negation:

man(fq(man0)).
~loves(fq(man0),mary)).

>>Answer: Yes,
        all(fq(man0),man(fq(man0)=>loves(fq(man0),mary)).
=======================
        :
        |
```

Session 3.3.1.2: Example of an obtuse answer

We can see from the above session that the final answer is "*all(fq(man0), man(fq(man0) => loves(fq(man0),mary))*" where "Y" is instantiated with "fq(man0)" during the refutation process. In other words, the answer means that all men, namely "fq(man0)", loves mary . It should be noted here that "fq(man0)" is the questioned Skolem function which replaces the universal quantifier "Y". This answered PC is quite an obtuse answer and it does not clearly show that all men love mary due to the existent of "fq(man0)" in the final answer.

To prevent this obtuse answer, we will replace all questioned Skolem functions appearing in the final answer (after the proving has been done) with variables as opposed to the method suggest by Nilsson[1980] which replaces Skolem functions before the refutation process or immediately after the process of transforming the negation of question into clausal form. After substituting all questioned Skolem functions with variables, we will get "*all(Z,man(Z)=>loves(Z,mary))*" as a final answer which is more intelligible and meaningful. In order to replace all Skolem functions in the final

answer, we need to keep a list of all Skolem functions of questions during Skolemisation of existential quantifiers for query clauses.

The same technique as described before (section 3.3.1.1) is used to create a symbol for Skolem functions for all query clauses, i.e by defining predicates *skolemq/4* (procedure SKOLEMQ -see Appendix) and *pickskolemq/3* (or procedure PICKSKQ -see Program 3.3.1.7).

```
pickskolemq(Name,Vars,Sk):-
        gensym(Name,F),
        append([F],Vars,Fandargs),
        Sk=..[fq|Fandargs].
```

Program 3.3.1.7: Procedure PICKSKQ

In general , if H is a Horn clause generated from the negation of a question , then *query(H)* is asserted into the database as facts.

```
/* Procedure ASSQ :to assert query clauses into the database */
assert_query(Q,Clause,Sklist):-
        translateq(Q,Clause,Sklist),
        horn_clauses(Clause,Horn),
        qassertz(Horn).

/* Procedure TRQ :to translate PC, Q, into Clocksin format, Clause */
translateq(Q,Clause,Sklist):-
        translate_top(S,X2),
        skolemq(X2,X3,[],Sklist), /* Procedure SKOLEMQ :Stage 3 for query clauses */
        translate_bottom(X3,Clause).

/* Procedure QASS :assert "query(H)" in the database  */
qassertz([]):-!.
qassertz([H|T]):-
        assertz(query(H)),
        qassertz(T).
```

Program 3.3.1.8: Assert query clauses into the database

As shown in Program 3.3.1.8, procedure ASSQ (predicate *assert_query/3*) will convert a negation of question, Q, (in PC) and transform it into Horn clauses (procedures TRQ and HC) and assert the resulted headed Horn Clauses into the database (procedure QASS or predicate *qassertz/1*). This procedure is actually equivalent to

procedure ASSKG but both of them produce two different type of clauses (query and knowledge).

For example, if Q (in PC form) is a question, then query clauses will be asserted into the database by calling predicate *assert_query(~Q,C,Sklist)*, where variable "Sklist" contains a list of all questioned Skolem functions. It should be noted here that this predicate will not be used or executed independently, but will be used in conjunction with proving a goal or answering a question. However let see the following session:

```
?- assert_query( ~all(X,man(X)=>loves(X,mary)),C,Sklist).

  C = [man(fq(man0)), ~loves(fq(man0),mary))].

  Sklist = [[_1,fq(man0)]]

yes

?-listing(query).

query(man(fq(man0))).
query(~loves(fq(man0),mary))).

yes
```

Session 3.3.1.3i Asserting query clauses into a database

It can be seen from the above session (3.3.1.3) that by calling the predicate *assert_query/3*, we will get the the result of the converting the negation of the question into Horn clauses. The variable "C" refers to the resulting clauses in Clocksin format (as described before) and the variable "Sklist" consist a list of questioned Skolem functions and its original universal quantifiers, i.e in this case, the original universal quantifier of the question is "X" or "_1" (Prolog variable). The listing in the above session shows the query clauses as in the database.

3.3.2 Goal Formatting.

Once we have facts or have set up a database, i.e a collection of knowledge and query clauses, we can ask a question or prove whether a statement follows or can be deduced from the database (KB clauses). A goal clause in a set of Horn clauses is the headless one i.e cl([],B). These can be written in Prolog's syntax as

?- B

where B is a collection of literals separated by comma. For example, "cl([],[a,b,c])" or equivalently "[]:-a,b,c" , is written as a goal in Prolog as

?- a,b,c.

In order to format a goal in Prolog, one of query clauses must be a headless Horn clause, i.e *query([]:-B)* or cl([],B). As we have noticed that there does not necessarily exist a headless query clause in the database , therefore one of query clauses must be converted into a headless one. In other words, the head of the goal clause must be an empty set [].

It can also be noticed that the required format of the goal clause can be derived directly from the corresponding clause in Clocksin format. Thus we do not need to reconvert again from a headed Horn clause to a headless Horn clause. It is enough just by storing the corresponding clauses in Clocksin format and then transferring the head of the clause into the body of the clause. For examples:

[3.3.2.1]. If the query clause is

human(X):-man(X).

then the corresponding goal clause is

[]:-~human(X),man(X).

[3.3.2.2]. If the question is

```
exists(X,wombat(X)&happy(X))
```

then the corresponding goal clause is

```
[]:-wombat(X),happy(X).
```

or in Prolog is written as

```
?-wombat(X),happy(X).
```

(Note: This is a headless clause, so nothing is done)

[3.3.2.3] If the query clauses in Clocksin format are

```
wombat(fq(wombat0)).
lives(fq(wombat0),zoo).
```

then the corresponding goal clauses are

```
[]:-~wombat(fq(wombat0))
[]:-~lives(fq(wombat0),zoo).
```

The program to format a goal is similar to the program described in Stage 7 (section 3.2) with the exception that we do not need to create N equivalent goals as all N equivalent Horn clauses produce the same goal clause. For example, the following two equivalent Horn clauses of the query clause as in example [3.3.2.1]:

```
human(X):-man(X).
~man(X):- ~human(X).
```

will generate the same goal clause:

```
[]:- ~human(X),man(X).
```

The following procedure FG or predicate *format_goal/2* (see Program 3.3.2.1) will format a goal from the query clause in the form of Clocksin format.

```
/* Procedure FG :goal formatting */
format_goal([]:-Body,Body):-
        !. /* procedure FG.1 */
format_goal(H:-T,Goal):-
        /* procedure FG.2  */
        convert(H,Noth),                    /* procedure CONV */
        appendbody(Noth,T,Goal),!.          /* procedure APPB */
format_goal(H,Goal):-
        /* procedure FG.3 */
        convert(H,Goal),!.
```

Program 3.3.2.1: Procedure FG (goal formatting)

If the query clause is already in the form of a headless clause, then nothing is done (see example [3.3.2.1]) above), so procedure FG.1 will just take the body of the clause as a goal.

If there are literals in the head of ruled clause"H:-T", i.e set H is a nonempty one (see example [3.3.2.2] above), then all of them are moved into the RHS of the rule and appended to the current body to become a goal (procedure FG.2).

If the clause does not contain a body, i.e set T of "H:-T" is empty or the clause is a factual one (see example [3.3.2.3] above), then procedure FG.3 will carry across all the literals of set H into the RHS of the rule to become a headless clause and the new body is taken as a goal.

Procedures CONV and APPB have already been explained in Stage 7 of section 3.2 (see Program 3.2.5). The following session (Session 3.3.2.1) shows some examples of goal formatting by using procedure FG.

```
?-format_goal(([]:-wombat(X),happy(X)),Y).

  Y = wombat(_1), happy(_1).

yes

?-format_goal((human(X):-man(X)),Y).

  Y = *human(_1), man(_1).

yes
```

Session 3.3.2.1: Some examples of goal formatting

3.3.3 The Refutation Procedure

In the refutation or proving procedure, which will be described here, a combination of control strategies and resolution methods is adopted. The main strategy adopted is the linear input form. Although the linear input form strategy is incomplete in the sense that it does not produce all possible solutions, it is used because of its simplicity and efficiency. Furthermore Prolog adopts this strategy.

In addition to the linear input form strategy, we also adopt the set of support strategy (see chapter 2 for the definitions of all the strategies). By using the basis of the set of support strategy, we start with a headless clause from the set of query clauses. In other words, we take one of the query clauses and convert it into a headless clause and use this one as a goal clause or one of the starting parent clauses. The other parent clause is taken from the set of KB clauses.

The proving or refutation procedure is actually just like a Prolog interpreter written in Prolog. That is, we can define what is to run a Prolog program by something which is itself a Prolog program. This means that all the query and knowledge clauses are Prolog rules and can be executed directly.

Summarily, in our refutation or proving procedure, we start with the goal clause and resolve it with one of the KB clauses, to give a new clause. Then we resolve it with one of the KB clauses, and so on. At each stage, we resolve the

clause last obtained with one of the original knowledge clauses. At no point in the refutation procedures, do we either use a clause that has been derived previously or resolve together two KB clauses. In Prolog terms, the latest derived clause can be taken as the conjunction of goals yet to be satisfied. This starts off as the goal, and hopefully ends up as the empty clause.

At each stage , find a clause whose head matches the last literal of the goals, instantiate variables as necessary. Do the same process to the body of the instantiated clause which becomes yet another goal. The literal to be matched is always selected from the last one of the goal.

The instantiation of the variables or the unification procedure is based on Prolog instantiation. For example, if the goal clause is

```
[]:-drugpusher(X),officer(X).
```

then we will try to resolve officer(X) first and suppose that the following knowledge clause exists in the database:

```
knowledge(officer(fs(searched0,Y)):-entered(Y),~vip(Y)).
```

thus, the new resolvent will become

```
[]:-drugpusher(f(searched0,Y)),entered(Y),~vip(Y).
```

where X is instantiated with "fs(searched0,Y)". The new resolvent will form a new goal, i.e:

```
[]:-drugpusher(fs(searched,Y)),entered(Y),~vip(Y)).
```

So the proof continues with a new literal goal, "~vip(Y)", and so on taking new literal goal from right to the left of the goal. If the proof ends with the empty clause then the hypothesis is true, otherwise it fails.

The refutation or proving procedures are implemented by two different methods. One is a depth-first method and the other one is a breadth-first method. Before we describe both depth-first and breadth-first methods in sections 3.3.3.2 and 3.3.3.3 respectively, we will first explain the top level predicate which do the proving controlling and the answer printing.

3.3.3.1 The top level predicates

The most top level predicate which do the proving or refutation control is a predicate *pcquest/0* or procedure PCQUEST as shown in the Program 3.3.3.1 below. In other words, Program 3.3.3.1 shows how a question is proved or answered. Procedure PCQUEST will prompt a question in a predicate calculus form which will then be proved by procedure QUEST or predicate *question/1*

```
/* procedure PCQUEST :prompt a question and prove it */
pcquest:-
        read(Q),        /* read a question Q in PC form */
        question(Q).    /* procedure QUEST */

/* procedure QUEST :proving or answering a question Q */
question(Q):-
        clear_pc,                                  /* Step 1 :procedure CLEARPC */
        question_to_hornclause(Q,Clause,Sklist),   /* Step 2 :procedure QTHC */
        answer_search(Clause,Q,Y,Sklist),          /* Step 3 :procedure AS */
        print_answer(Q,Y,Clause).                  /* Step 4 :procedure PA */
```

Program 3.3.3.1: Proving a question

The procedure for proving or answering a question (as shown in the above program 3.3.3.1) is divided into four steps as follows:

[1]. Re-setting all control predicates.

[2]. Converting into Horn clauses.

[3]. Searching for answers.

[4]. Printing the answers.

3.3.3.1.1 Re-setting all control predicates (Step 1)

Before the refutation process is carried out, all control predicates will be reset by calling procedure CLEAR (or predicate *clear_pc/0*). Predicate *clear_pc/0* (see Appendix) will reset, by retracting or abolishing, all control predicates which will be used during proving the hypothesis or answering a question. Library predicates are used to retract (*retractall/1*) or to abolish (*abolish/2*) all control predicates such as *toptry0/1*, *toptry/1*, *proven/1* etc. All *query* clauses are also abolished from the database before we start the refutation process.

3.3.3.1.2 Converting into Horn clauses (Step 2)

The second step of procedure QUEST, i.e procedure QTHC or predicate *question_to_hornclause/3*, (see Program 3.3.3.2 below) is to convert a negation of the question ,Q, into *query* clauses (in the form of headed Horn Clause) by keeping its clauses in Clocksin format (variable "*Clause*") and also all questioned Skolem functions (variable "*Sklist*") for changing back into variables later, and then assert them (query clauses) into the database. This is done by calling predicate *assert_query(~Q,clause,Sklist)* or procedure ASSQ (see Program 3.3.8 of section 3.3.1.2 for its detail descriptions).

```
/* procedure QTHC */
question_to_hornclause(Q,Clause,Sklist):-
        assert_query(~Q,Clause,Sklist).        /* procedure ASSQ */
```

Program 3.3.3.2: Procedure QTHC

3.3.3.1.3 Searching for answers (Step 3)

The third step of procedure QUEST, i.e procedure AS (as shown in Program 3.3.3.3 below), is to search the answers for the question after its negation was transformed into Horn clauses.

At first, (the first procedure of predicate *answer_search/4*, i.e procedure AS.1), we will try to find all possible answers until the proving is exhausted. At this stage, the question is already in the form of Clocksin format (variable "*Clause*"). The prove is carried out by procedure ANS or predicate *answer/3*. This procedure (ANS) will return answer "yes" or "no" (variable "*Ans*") depending on the result of the proving.

For the purpose of printing and controlling, predicate *affirm(Ans)* will then be asserted in the database as in the procedure AS.1. Variable "X" which is actually the question in its original form (PC), contains all unified quantifier variables including questioned Skolem functions (if they exist).

As explained before (section 3.3.1.2) this questioned Skolem function gives unintelligent answers. Thus before this answered PC which may contain unified quantified variables is passed to other predicate for printing, all questioned Skolem functions, if they exist, will be changed back to variables (the last line of procedure AS.1) by calling procedure SSK or predicate *subst_skolem/3* (see Appendix for its definition). Thus, procedure SSK will return a new

answered PC ,"Y", which is free from any questioned Skolem functions.

```
/* procedure AS :searching answers for question Q */
answer_search(Clause,X,Y,Sklist):-
        /* procedure AS.1: find all possible answers until exhausted */
        answer(X,Clause,Ans),          /* procedure ANS */
        assertz_new(affirm(Ans)),
        subst_skolem(X,Y,Sklist).      /* procedure SSK */
answer_search(Clause,X,Y,Sklist):-
        /* procedure AS.2: no more answers (the proving ends) */
        Clause\==[],
        exists(toptry([])),!.
```

Program 3.3.3.3: Procedure AS

The second one of procedure AS, i.e procedure AS.2, will make itself successful after all possible proving paths are explored ,i.e until predicate *answer/3* fails, in order to show that the proof is finished. The ending of proving is indicated by the existence of predicate *toptry([])* (see Program 3.3.3.3). Eventually this procedure (AS.2) will return the uninstantiated value of an answered PC, i.e variable "Y".

It should be noted here that the procedure ANS or predicate *answer/3* is the core of refutation procedure and will be explained in the next sections 3.3.3.2 and 3.3.3.3.

3.3.3.1.4 Printing the answers (Step 4)

The fourth step of procedure QUEST is procedure PA (see Program 3.3.3.4). Procedure PA (or predicate *print_answer/4*) is called in order to print answers for the questions. The procedure PA will be divided into two subprocedures. The first one, procedure PA.1, is to print a remark about the inconsistency of the question. The remark is printed by calling procedure AF. The second one, procedure PA.2, is to

print the result of the proving of the question ,i.e either successful or failure one. This is done by calling predicate *print_answer0/2* or procedure PA0.

```
/* procedure PA printing the answer or solution */
print_answer(Q,Y,[]):-
        /* Procedure PA.1: the clause is an inconsistent one */
        affirm(Ans),
        answer_form(Y,Ans),     /* procedure AF */
        write(' The question clause is an inconsistent one '),
        !.
print_answer(Q,Y,Clause):-
        /* Procedure PA.2: print either successful or failure proving */
        Clause\==[],
        print_answer0(Q,Y).     /* procedure PA0 */

/* procedure AF */
answer_form(Y,Ans)
        write_answer(user,Ans),         /* procedure WA */
        writepc(user,Y),                /* procedure WPC */
```

Program 3.3.3.4: Procedures PA and AF

Procedure AF or predicate *answer_form/2* (see also Program 3.3.3.4) will print a remark of the result or answer (either "yes" or "no" depending on the value of "Ans") , and also the answered PC. The remark and the answer itself are printed by procedures WA and WPC respectively.

As shown in Program 3.3.3.5, procedure PA0 or predicate *print_answer0/2* is subdivided into two procedures, namely procedure PA0.1 and PA0.2, which will handle positive (yes) answer and negative (no) or end of proving remarks respectively. The positive answer or successful proving is shown by the existence of predicate *affirm(yes)* and the instantiated answered PC, "Y", (nonvar(Y) is true). As described before that the unsuccessful proving will return an uninstantiated value of "Y" (see procedure AS.2 of Program 3.3.3.3). The printing of successful result is done by predicate *more_answer/1* or procedure MORE-ANS.

```
/* procedure PAO */
print_answer0(Q,Y):-
        /* procedure PAO.1: to print "yes" answer */
        nonvar(Y),
        affirm(yes),
        answer_form(Y,yes),     /* procedure AF */
        !,more_answer(Y).       /* procedure MORE-ANS */
print_answer0(Q,Y):-
        /* procedure PAO.2: end of proving processing */
        exists(toptry([])),
        test_finish_fact(Q).    /* procedure TFF */
```

Program 3.3.3.5: Procedure PAO

After the printing of result, the enquirer is given a set of options. Those options are by selecting or typing one of the following: ;, Y , a, n, b, <return>, p or others characters. The following table gives the meaning of every options.

Option	Meaning
a	abort
b	break
p	printing solution's tree or graph
n or <return>	not to find other answers (satisfy with the given answer)
; or y	find if there is other answer
others	displaying help table (like this one)

TABLE 3.3.1: Options for procedure AR

The response from the enquirer will be diagnosed by predicate *answer_response/2* or procedure AR (see Appendix). This procedure, AR, is called from procedures MORE-ANS which is shown in Program 3.3.3.6. When option p is typed, predicate *print_solution* will print the solution's tree (see Appendix).

```
/* procedure MORE-ANS */
more_answer(Y):-
        write_proved(user,Y),
        get0(A),
        answer_response(A,Y).   /* procedure AR */
```

Program 3.3.3.6: Procedure MORE-ANS

The second subprocedure of *print_answer0*, i.e procedure PAO.2, deals with the end of proving sign. The proof ends when predicate *toptry([])* exists in the database. If this is

so, predicate *test_finish_fact/1* or procedure TFF will be called. As shown in Program 3.3.3.7, procedure TFF is sundered into two procedures, namely procedure TFF.1 and TFF.2, by the value of "Ans" of predicate *affirm(Ans)*.

Procedure TFF.1 is to print a remark that no more more answers are possible or the proof ends successfully provided that there exists the predicate *affirm(yes)*. On the hand, if the "Ans" is instantiated to "no", then procedure TFF.2 will print a remark that the proof is unsuccessful and then the enquirer or user will be ask a confirmation of asserting the question as a knowledge in the database. Procedure AAF or predicate *ask_assert_fact/2* (see Appendix) will diagnose the response given by the enquirer. There are only two options available ,i.e y and n for yes and not to assert the question into the database respectively. If the user agree to assert the question as a fact or knowledge, then procedure ASSKG (predicate *assert_knowledge*) will be called, otherwise procedure TFF will do nothing and eventually the whole question-answering process comes to end.

```
/* procedure TFF */
test_finish_fact(Q):-
        /* procedure tff.1: to print 'no more answers ' */
        affirm(yes),
        write_answer(user,finish),!.    /* procedure WA */
test_finish_fact(Q):-
        /* procedure tff.2: to ask  confirmation of asserting a question
                        as a fact into the database */
        affirm(no),
        answer_form(Y,no),      /* procedure AF */
        nl,nl,tab(5),write('Do you like to assert '),
        nl,tab(8),write_quote(Q),
        nl,tab(5),write(' as a  fact  in  the database (y/n) ? '),
        get0(X),
        ask_assert_fact(X,Q).   /* procedure AAF */
```

Program 3.3.3.7: Procedure TFF

3.3.3.2 The Depth First Method

Beside using the combination of the set of support strategy and linear input resolution as described earlier, we also adopt the unit preference strategy in the depth first method. This is to make sure that the resolvent has fewer literals than do their other parents i.e the goal. This process helps to focus the search towards producing an empty clause, thus typically increases efficiency, although it may take a longer path to reach the solution.

By using this method, we will prove only one goal clause at any time. So there is only one goal clause or headless Horn clause existing at any time of the proving. The proof is successful if one of the goal clauses can derive an empty clause, otherwise the proof fails if all goal clauses cannot derive an empty clause. Thus, for instance, if the following is a list of query clauses:

[wombat(fq(wombat0)),lives(fq(wombat0),zoo)]

then first we convert the first query clause or the head of the list into a goal or a headless clause, i.e.

[]:-~wombat(fq(wombat0)).

and we try to derive an empty clause from the above clause. If it is successful, then the hypothesis is true. Otherwise, we convert the second query clause of the above original list or the head of the current list, i.e [lives(fq(wombat0),zoo)], into a headless Horn clause and try to prove it, that is

[]:- ~lives(fq(wombat0),zoo).

If this one is still unprovable then the hypothesis is false or the question can not be proved, as there is no query clauses left in the current list, i.e [] (an empty list).

We will divide the refutation procedure based on a depth-first method into five different levels of major procedures as follows where the level 1 is the highest level and the level 5 is the lowest (deepest) one:

[1]. Procedure ANS

[2]. Procedure ASK

[3]. Procedure FACTPR

[4]. Procedure BASEPR

[5]. Procedure FACTCL

3.3.3.2.1 Level 1: Procedure ANS

The highest level predicate or procedure in a depth-first method of refutation procedure is procedure ANS (or predicate *answer/3*), as shown in Program 3.3.3.8, which will eventually return answer "yes" or "no" to the proving process of the question. This procedure which is called from procedure AS.1 (see Program 3.3.3.3 of section 3.3.3.1), is subdivided into two procedures, i.e subprocedures ANS.1 and ANS.2.

```
/* procedure ANS : answering the question */
answer(Pc,Q,yes):-
        /* procedure ANS.1: return remark "yes" for successful proving */
        asking(Pc,Q),           /* procedure ASK */
        assert_once(toptry0(Pc)).
answer(Pc,Q,no):-
        /* procedure ANS.2: return remark "no" for unsuccessful proving */
        not_exists(toptry0(Pc)).
```

Program 3.3.3.8: Procedure ANS

The first one, procedure ANS.1, will assert predicate *toptry0(Pc)* only once in the database if the proof is successful, and also it will return remark "yes". The proof is carried out by procedure ASK. In other words, the proof is successful if procedure ASK succeeds.

Otherwise, if the proof fails which is a consequent of the failure of predicate *asking/2*, the second procedure, ANS.2, will return remark "no". The predicate *asking/2* fails when all query clauses have been proved unsuccessful. This is indicated by the nonexistence of predicate *toptry0(Pc)* in the database; on the contrary the existence of predicate *toptry0(Pc)* (which is asserted by procedure ANS.1) will show that at least one of the query clauses has deduced an empty clause though predicate *asking/2* fails. The predicate *asking/2* also fails when the resulting Horn clause of the negated question is an empty one (*Clause*=[]). This means that the question clause is an inconsistent one because the negation of an inconsistent clause is an empty one (tautology).

3.3.3.2.2 Level 2: Procedure ASK

The major procedure in second level of refutation procedure is procedure ASK which is called from procedure ANS at level one.

Procedure ASK (predicate *asking/1*), as shown in Program 3.3.3.9, will prove the head clause in Clocksin format of the current list of query clauses one by one depending on the result of proving or the needs of other possible answers until the current list become empty. So, we will divide this procedure into three subprocedures.

The first suprocedure, i.e Procedure ASK.1, will first convert the head of the current list into a headless one by calling procedure TOPASK and then try to derive an empty

clause from it (the resulting headless clause) by calling procedure FACTPR. If the proof of the clause is unsuccessful (procedure FACTPR fails) or there are no more answers can be generated from the clause when it is asked by the enquirer (procedure FACTPR exhaustively fails) or the question clause is a tautology one (procedure SUCCESS fails), then backtracking will occur and the same process will repeat but by taking the next query clause in the current list i.e list "Quest", (procedure ASK.2).

```
/* Procedure ASK : to prove each clause of the question */
asking([Quest1|Quest]):-
        /* procedure ASK.1 */
        top_asking(Quest1,Goal),        /* procedure TOPASK */
        factprolog(Goal,[],[],Hp,[]),   /* procedure FACTPR */
        successful_action(Hp).          /* procedure SUCCESS */
asking([Quest1|Quest]):-
        /* procedure ASK.2 */
        print_comment(Quest1),          /* procedure PRCMT */
        asking(Quest).    /* prove the next query clause */
asking([]):-
        /* procedure ASK.3 */
        assertz(toptry([])),fail.
```

Program 3.3.3.9: Procedure ASK

The proof comes to an end when the current list become empty (procedure ASK.3). There are two possible conditions when this state is reached, either there are no more possible solutions or the proof is unsuccessful. In either case, procedure ASK.3 will assert predicate *toptry([])* into the database to mark the end of proving and return the value of false, i.e the predicate *asking/1* will fail at the end of proving. Consequently procedure ANS which calls it, will return with remark "no" (see procedure ANS.2 of Program 3.3.3.8).

3.3.3.2.3 Level 3

There are three major procedures in level three which are called from procedure ASK at level two previously. There are procedures TOPASK, SUCCESS and FACTPR which can be classified as before proving, after successful proving and the proving itself respectively.

Level 3.1: Procedure TOPASK

As we said earlier that before the proving process is carried out, procedure TOPASK or predicate *top_asking/2* (see also Program 3.3.3.10) is called first. We will keep a log of goals and parent clauses by predicates *goal/2* and *proving/2* respectively at each resolution node at which both of them are successfully resolved with each other. Furthermore, both predicates will be used during printing the solution's tree. So, predicate *node/1* denotes node number for each successful unified node. In other words, procedure TOPASK will initialise predicate *node/1* by asserting a new predicate *node(1)* (by calling predicate *assertz_new(node(1))*) and do the goal formatting (predicate *format_goal/2*).

Level 3.2: Procedure SUCCESS

As the needs of proving other clauses from the same question in order to find other possible answers or solution tree may or usually does arise, but still, during that process we do not want to prove the same goal clause twice or more. However, there is no harm by proving the same factual KB clause over and over again, but proving

the same ruled KB clause can be overtaxed. For instance, the following is a part of the solution tree:

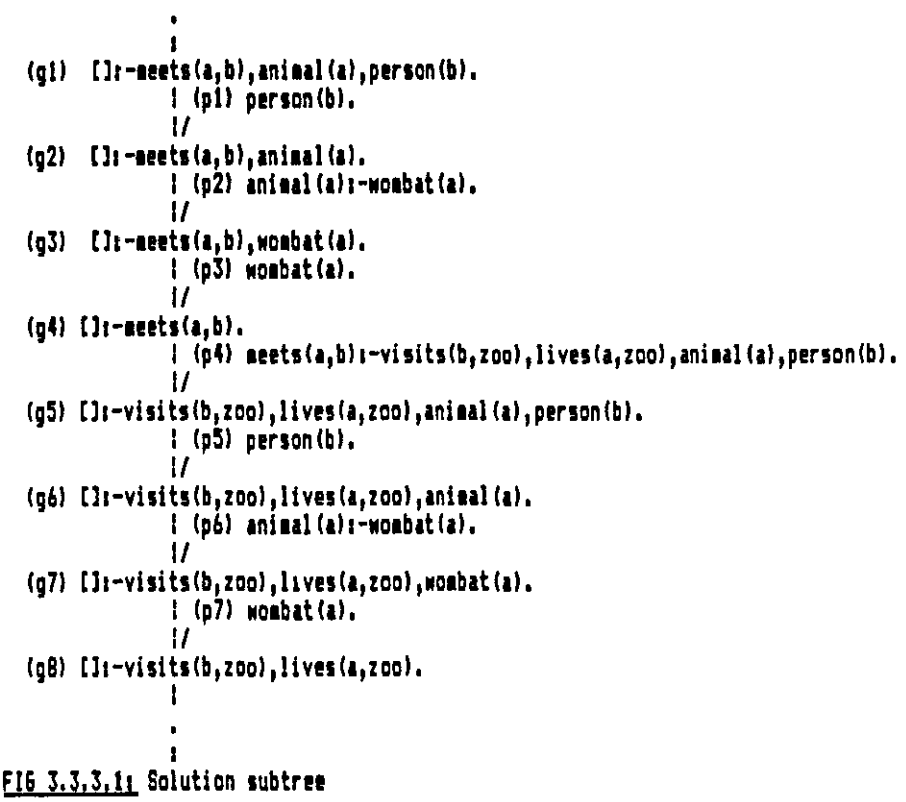


FIG 3.3.3.1 Solution subtree

It can be seen from the above subtree (Fig. 3.3.3.1) that three pairs of parent clauses are the same i.e clauses [(p1),(p5)], clauses [(p2),(p6)] and clauses [(p3),(p7)]. The first, [(p1),(p5)] and third [(p3),(p7)] pairs are factual KB clauses and the second one, [(p2),(p6)], is a ruled KB clause.

All the proven ruled KB clauses can be considered as lemmas to the other goal clauses from the same question. When the same ruled KB clauses are going to be proved again, we do not have to repeat the proving of the same ruled clauses once again, but we will used the proven ruled KB clause to show that we have already proved it successfully before by asserting them into the database (using predicate *proven/1*). So, from Fig 3.3.3.1 above, if

we assert clause (p2) (i.e *proven(animal(a):-wombat(a))*) into the database then we can save one step here, that is, we can go straight from goal (g6) to (g8) without the need to prove "wombat(a)" again (goal g7). However, we do not gain any advantages from asserting proven factual KB clauses.

Accordingly, after the proving ends successfully, procedure PROVEN (predicate *assertz_proven/1*) which is called from procedure SUCCESS (see Program 3.3.3.10), will assert into the database all the proven ruled clauses which was passed from procedure FACTPR. So if "Hp" is a proven ruled KB clause, then "proven(Hp)" is asserted in the database.

```
/* procedure TOPASK : top part of procedure ASK.1 */
top_asking(Quest1,Goal):-
        assertz_new(node(1)),          /* to initialise node(1) */
        format_goal(Quest1,Goal),!.    /* procedure FG: goal formatting */

/* procedure SUCCESS */
successful_action(Hp):-
        assertz_proven(Hp).     /* procedure PROVEN */

/* procedure  PROVEN */
assertz_proven([]).     /* the end of procedure */
assertz_proven([Hp|Hptail]):-
        /* procedure  PROVEN.1 */
        clause(proven(Hp),true),
        assertz_proven(Hptail),!.
assertz_proven([Hp|Hptail]):-
        /* procedure  PROVEN.2 */
        assertz(proven(Hp)),
        assertz_proven(Hptail).
```

Program 3.3.3.10: Procedures TOPASK, SUCCESS and PROVEN

If there already exists fact *proven(Hp)* in the database,then do not assert it in the database, but continue asserting other proven clauses (procedure PROVEN.1). Otherwise procedure PROVEN.2 will assert *proven(Hp)* in the database and also continue asserting other proven clauses if they exist.

Level 3.3: Procedure FACTPR

All the proving will be done by procedure FACTPR, as shown in Program 3.3.3.11). Procedure FACTPR is subdivided into two subprocedures, viz. procedures FACTPR.1 and FACTPR.2.

```
/* procedure FACTPR */
factprolog((Q1,Q2),Usedclauses,Hp,Hp1,Goalclause):-
        /* procedure FACTPR.1 */
        !,
        factprolog(Q2,Usedclauses,Hp,Hp2,[Q1|Goalclause]),
        factprolog(Q1,Usedclauses,Hp2,Hp1,Goalclause).
factprolog(Q,Usedclauses,Hp,Hp1,Goalclause):-
        /* procedure FACTPR.1 */
         assertagoal1([Q|Goalclause],N),                /* procedure ASSGL1 */
        baseprolog(Q,Usedclauses,Hp,Hp1,Goalclause,N).  /* procedure BASEPR */
```

Program 3.3.3.11: Procedure FACTPR

This procedure is actually a top level of refutation process. In other words, procedure FACTPR will steer the proving order of each literal of the goal clause. Hence, the first procedure, FACTPR.1, will eventually show the way of proving the goal which consists a conjunction of literals (in the form of "k,l,...").

In this case, the proving is done from right to the left of the goal, a contrast to the Prolog method where it does from left to the right of the goal. It can be easily modified to make it proving from left to right. The second one, procedure FACTPR.2, will call a refutation procedure, BASEPR, in order to match a literal of the goal with the head of KB clauses.

3.3.3.2.4 Level 4

There are two major procedures in level four, i.e procedures ASSGL1 and BASEPR which are called from procedure FACTPR at level three (Level 3.3).

Level 4.1: Procedure ASSGL1

Before a matching or unifying is done, procedure ASSGL1 or predicate *assertagoal1/2* is called. Procedure ASSGL1 (as shown in Program 3.3.3.12), firstly, will retract all predicates *goal/2* with the same node number N and assert a new *goal(Goal,N))*, then update node number (procedure UPNODE).

```
/* procedure ASSGL1 */
assertagoal1(Goal,N):-
        node(N),
        retractall(goal(_,N)),
        asserta(goal(Goal,N)),
        updating_node(N).       /* procedure UPNODE */

/* procedure UPNODE */
updating_node(N):-
        /* procedure UPNODE.1 */
        N1 is N+1,
        assertz_new(node(N1)).
updating_node(N):-
        /* procedure UPNODE.2 */
        assertz_new(node(N)).
```

Program 3.3.3.12 : Procedures ASSGL1 and UPNODE

The first procedure, UPNODE.1, (of procedure UPNODE of Program 3.3.3.12) will assert a new predicate *node(N1)* where N1 is an updated node number. On the other hand, the second procedure UPNODE.2 will assert the old node number (N), i.e decreasing by one (compared to N1). This is a consequent of the failure of procedure BASEPR (see procedure FACTPR.2 of Program 3.3.3.11), so backtracking will occur and the old node number must be restored in order to start a new solution tree.

Level 4.2 Procedure BASEPR

While procedure FACTPR do the proving steering, procedure BASEPR (Program 3.3.3.13) will do the actual matching with the current KB clauses. As there is a possibility that any literal of the goal clause is a Prolog or library predicates, procedure BASEPR.1 will match the goal literal with them first. If it is unsuccessful, then procedure BASEPR.2 will match or unify the goal literal with any factual KB clauses. This strategy is known as the unit preference strategy. If there is no factual KB clauses which can be matched with the goal literal, Q, then procedure BASEPR.3 will try to match it with any ruled KB clauses. If it is successful, then procedure FACTCL is called to do more test before the body of the matching ruled KB clause is considered or taken as a next goal.

```
/* procedure BASEPR */
baseprolog(Q,Usedclauses,Hp,Hp,Goalclause,N):-
        /* procedure BASEPR.1 */
        clause(Q,_),
        call(Q),
        assertaproving2(Q,N).          /* procedure ASSPRV2 */
baseprolog(Q,Usedclauses,Hp,Hp,Goalclause,N):-
        /* procedure BASEPR.2 */
        knowledge_base(Q),
        assertaproving2(Q,N).          /* procedure ASSPRV2 */
baseprolog(Q,Usedclauses,Hp,Hp1,Goalclause,N):-
        /* procedure BASEPR.3 */
        knowledge_base(Q1-A),
        factclause(Q1-A,Usedclauses,Hp,Hp1,Goalclause,N).  /* procedure FACTCL */
```

Program 3.3.3.13: Procedure BASEPR

At the end of successful matching of a goal literal with any KB clauses or system predicates (i.e at the end procedures BASEPR.1, BASEPR.2 and also in procedure FACTCL.2), the matching KB clauses or system predicates are asserted into the database by procedure ASSPRV2 (as shown in Program 3.3.3.14) in order to trace a solution

tree. So procedure ASSPRV2 will assert into the database a new predicate *proving(Q,N)* where Q and N are the matching clauses (parent clauses) and node number respectively after retract all predicate *proving/2* with the same node number.

```
assertaproving2(Q,N):-
        retractall(proving(_,N)),
        asserta(proving(Q,N)).
```

Program 3.3.3.14: Procedure ASSPRV2

3.3.3.2.5 Level 5: Procedure FACTCL

Although all KB clauses are Prolog rules and that they can be executed directly, cares must be taken to prevent cycling or looping in the refutation process. Cycling or looping in the refutation process can happen when the goal clause has itself as a subgoal. This may result from using the same clause as one of the parent clauses. This (cycling) is the main disadvantage of implementing the depth first method as the method does not guarantee that the solution will be reached though it exists. For example, if we have the following knowledge clauses (This is a famous monkey-banana problem):

```
(k1) at(X,Y,X,walk(Z,X,S)):-at(Z,Y,X,S).
(k2) at(X,X,X,carry(Y,X,S)):-at(Y,X,Y,S).
(k3) reach(climb(S)):-at(b,b,b,S).
(k4) at(a,b,c,s).
```

And if we try to prove []:-reach(B), then we will get the refutation tree as follows:

```
(g1) []:-reach(B)
              | (p1) reach(climb(S)):-at(b,b,b,S)
              |/
(g2) []:-at(b,b,b,S)
              | (p2) at(X,Y,X,walk(Z,Y,T)):-at(Z,Y,X,T)
              |/
(g3) []:-at(Y,b,b,Z)
     /|\      | (p3) at(X,Y,X,walk(Z,Y,T)):-at(Z,Y,X,T)
      |_____|/
```

FIG 3.3.3.7: Refutation subtree (looping)

It can be seen from the above subtree (Fig. 3.3.3.2) that a cycle occurs in the above refutation process because the parent clauses (p2) and (p3) are the same, i.e knowledge clause (k1), thus produce the same goal (g3). Clause (k1) is chosen in preference to clause k(2) because it is on the top of the list as Prolog adopts the selection from the top to the bottom. The cycling must be prevented from happening if we would like the refutation to reach a conclusion either positive or negative but not hanging around until the Prolog stack is exhausted.

As all KB clauses are Horn clauses and at any time we only delete one literal which is actually a head of one of the parent clauses, therefore we can make a log of the usage of all ruled KB clauses in order to prevent the cycling. The log will keep a record of all the ruled clauses which have been used as one of the parent clauses such that the same ruled clauses will not be used twice or more in the same path or subtree of the proving process. If the ruled clause has been successfully proved then this clause will be taken out from the record so that this clause can be used again in the proving process but in a different path (branch) of the proving tree or in the different subtree.

So, if we keep a log of the usage of all ruled parent clauses, by taking it as an argument of predicate '*try*'. For instance, from the above tree (Fig. 3.3.3.2), the side clauses (one of the parent clauses which are ruled clauses; the other one is a headless or goal clause) will be asserted in a database as follows:

```
try(reach(climb(S1)),[at(_,b,b,S1)]).
try(at(X,Y,X,walk(Z,X,S)),[at(Z,Y,X,S)]).
```

Then , we can continue the refutation process by taking knowledge clause (k2) as the third parent (p3'). In this case, we can not used knowledge clause (k1) again because the predicate *try(at(X,Y,X,walk(Z,X,S)):-at(Z,Y,X,S))* exists in the database. Thus, the new refutation subtree (starting from goal (g3)) is as follows:

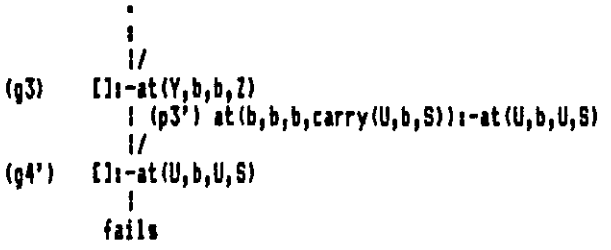


FIG 3.3.3.3: Refutation subtree (failure)

The subtree (Fig 3.3.3.3) in the above example incidentally does not derive an empty clause. However, by taking knowledge clause (k3) instead of clause (k2) as the second ruled (parent) clause , then we will be able to derive an empty clause; hence the proof is successful as shown in the following subtree (Fig 3.3.3.4). The variable "B" will be instantiated to "climb(carry(c,b,walk(a,c,s)))" or "walk from position a to position c then carry the box to position b and finally climb it to reach a banana"

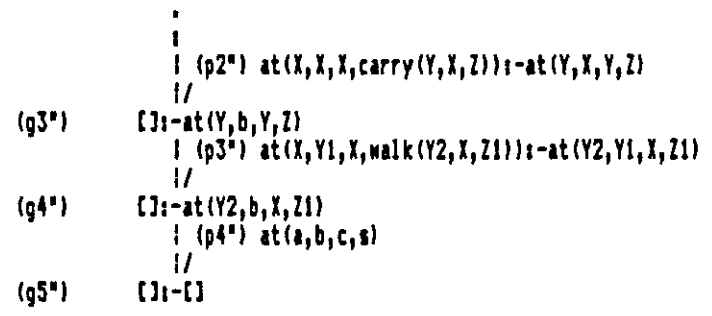


FIG 3.3.3.4: The refutation subtree (successful)

or all above subtrees (Fig. 3.3.3.2, Fig 3.3.3.3 and Fig 3.3.3.4) are combined, then the following graph (Fig. 3.3.3.5) will be produced:

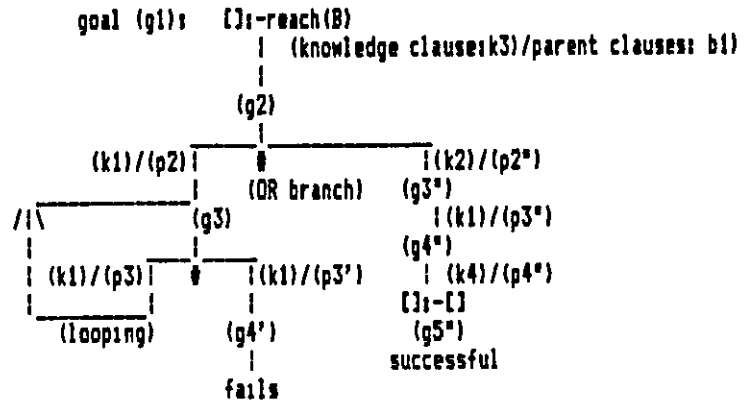


FIG 3.3.3.5: Refutation tree (combined)

From Fig. 3.3.3.4, the number associated with a goal, for instances g1 and g3", refers to node number. Thus the number with mark " is a current value of node number (see Program 3.3.3.12). Clauses referred by g (for instance g3") and p (for instance p3") are inserted into the database by predicates *goal/2* and *proving/2* respectively.

This cycling checking will be carried out after the literal goal is successfully matched with any ruled KB clauses in the procedure FACTCL. The following program, i.e Program 3.3.3.15, shows procedure FACTCL which is called after the literal goal is matched with any ruled KB clauses. This procedure is divided into two, i.e procedures FACTCL.1 and FACTCL.2.

```
/* procedure FACTCL */
factclause(Qs-A,Usedclauses,Hp,Hpl,Goalclause,N):-
        /* procedure FACTCL.1 */
        haveproved_top(Qs-A,Hp),        /* procedure HAVEPR */
        assertaproving2(Q,N).           /* procedure ASSPRV2 */
factclause(Qs-A,Usedclauses,Hp,[Qs-A|Hpl],Goalclause,N):-
        /* procedure FACTCL.2 */
        nottry1(Qs-A,Usedclauses),      /* procedure NOTTRY1 */
        check_failure(Qs-A),            /* procedure CHFAIL */
        assertaproving2(Qs-A,N),        /* procedure ASSPRV2 */
        factprolog(A,[Qs-A|Usedclauses],Hp,Hpl,Goalclause),   /* procedure FACTPR */
        retractall(failure(Qs-A)).
```

Program 3.3.3.15: Procedure FACTCL

The first one, procedure FACTCL.1, will check whether the matching ruled KB clause has been proved before. The checking is carried out by procedure HAVEPR by matching them with predicate *proven/1* or all ruled clauses in list "Hp". If it has already been proved before, then assert it into the database by procedure ASSPRV2. If it has not been proved before, the second procedure, FACTCL.2, will first check whether the matching ruled KB clause has been matched before in the same subtree or branch by calling procedure NOTTRY1.

Procedure NOTTRY1 will succeed if none has been matched before. Following that, the predicate *proving/2* is asserted into the database by procedure ASSPRV2 with the matched ruled KB clause is one of the objects (arguments). Then the body of the matched ruled clause is taken as a next goal and the proving is repeated until the empty clause is deduced or no more ruled KB clause can be matched with the literal goal.

3.3.3.3 The Breadth-First Method

By adopting this method, all possible resolvents will be generated at each level of the refutation tree until it encounters an empty clause or resolvent. The refutation tree generated is an OR graph. In order to generate a new resolvent, each literal in the goal clause will be resolved with one of the KB clauses from right to the left of the goal. This method is quite different to the above depth-first method in the respect of the way of a new resolvent is generated. In the depth first method, only one literal of the goal will be resolved upon. While using this method, all literals of the goal clause will be resolved from right to the left of the goal in order to reduce the number of possible resolvents generated, so the refutation graph will be much smaller. For example, if the goal clause is as follows:

```
[]:-strong(nazrul),intelligent(nazrul).
```

and given that the following are KB clauses:

```
intelligent(nazrul).
strong(X):-athlete(X).
```

then the resolvent will be

```
[]:-athlete(nazrul).
```

as opposed to the above depth first method where the resolvent will be:

```
[]:-strong(nazrul).
```

This is due to that in the breadth-first method, all literals of the goal clause will be resolved with the KB clauses, i.e both literals "*strong(nazrul)*" and "*intelligent(nazrul)*" of the goal clause will be resolved with the KB clauses. On the other hand, the depth-first method will only resolve one literal goal with one KB

clause, i.e "*intelligent(nazrul)*" of the goal clause will resolve with a KB clause ("*intelligent(nazrul)*").

As the depth-first method, the same combination of strategies is adopted, i.e a combination of the linear input, the set of support and the unit preference strategies. However, as the name of this (breadth-first) method suggests that all literals goal of query clauses will be proved in parallel.

The proof is successful if we encounter an empty resolvent, "[]:-[]". Otherwise, the proof is unsuccessful if we cannot resolve the new generated resolvents any more. In other words, we come to a dead end where all goal clauses cannot derive an empty clause any further.

One or more new goal clauses which are a result of resolution between the current goal clause and the KB clauses, are said at one level down (or up) to the current goal. So at every level there will be a set of goal clauses which is a result of previous resolution, and at level 1, all goal clauses are query clauses themselves. In other words, the goal clause is a father node and its resolvents with KB clauses are its son (daughter). For instance, Fig. 3.3.3.6 shows a solution tree of a question , (p&(q#r)). Thus its negation will produce two query clauses, i.e "[]:-p,q" and "[]:-p,r".

By using a breadth first method, we will prove both query clauses in parallel, that is we will find their sons (if they exist) and followed by finding their grandsons and

then followed by finding their great-grandson and son on until we encounter an empty clause. Their son is yet became another goal clause and so their grandson and grand-grandson. In order to find their sons, the goal clause will be unified or matched with KB clauses. The order of matching of each literal of the goal clause is from right to the left of the clause. From Fig. 3.3.3.6, the clause in a curly bracket, {}, is a KB clause which is a matching clause for the literal goal.

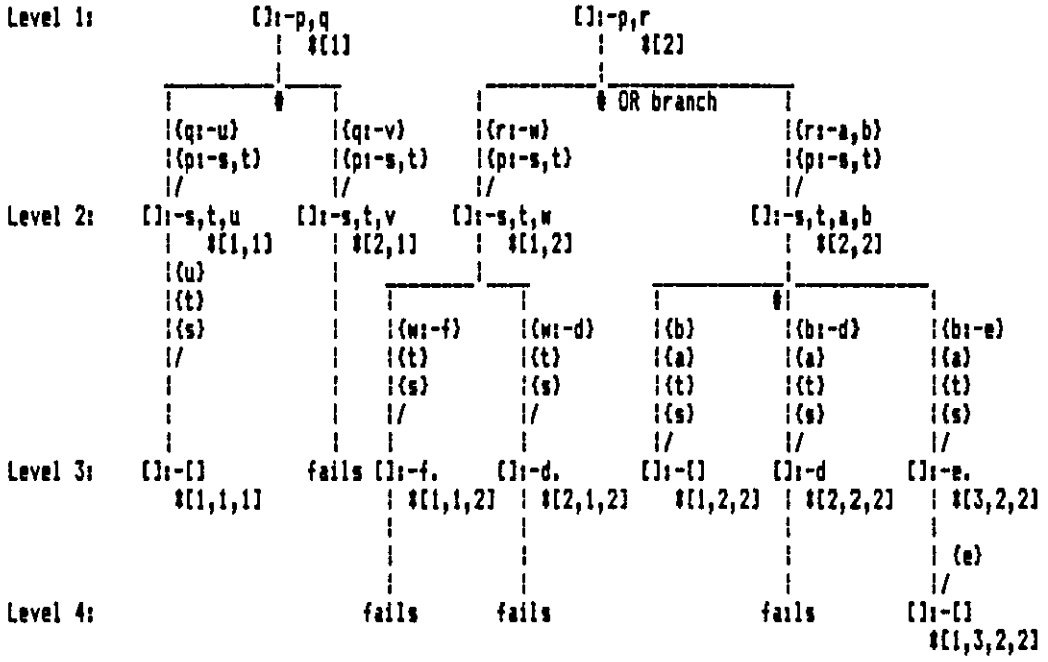


FIG 3.3.3.6: Solution tree using a breadth-first method

Each goal clause is given a node number in the form of a list in a square bracket, [], which is marked by a star, *, as in Fig. 3.3.3.6. The node number is given to a new son by prefixing the father's number with the son's number. For example, if the father's (goal clause) number is [2], thus its first son is given a number [1,2] by prefixing number 1 to the list [2] and the second son is given a number [2,2] and so on.

By using this notation, given any (goal) clause number, we can find its father, its grandfather and so on up to the query clause. So, for instance, if given a node number [1,2,2] (see Fig. 3.3.3.6), then its father is clause [2,2] (i.e the tail of the given node number, [1|[2,2]]) and its grandfather is clause [2] which in fact is a query clause. Accordingly we can find a solution path of the question from any empty clause. The length of list of the node number shows the level number of the node itself, for instance, node [1,3,2,2] is at level 4 (i.e *length([1,3,2,2],L)* where L is instantiated to 4).

Before we describe the program itself, let us define control predicates *c_node/4*, *node/4* and *node_no/1* which will be used in the program. Predicate *node_no/1* represents the number of sons for each goal clause such that its value will be reset to zero before unifying the new goal clause with any KB clauses. Both predicates *c_node/4* and *node/4* which represent goal clause and its son respectively, contain the information about the goal clause itself, i.e its literals, its node number, its level number and the list of a father-matching KB clauses which produces itself (a list in the curly bracket {} - see Fig. 3.3.3.6). Once the son become the goal clause, its representation changes from predicate *node/4* to predicate *c_node/4*. In other word, predicate *node/4* is a temporary representation of resolvent (or the son) node before the son becomes yet another goal. For example, predicate *c_node(a,[1,1,2],3,((w:-f),t,s))* (see Fig. 3.3.3.6) contains the information about the goal "[]:-a" where its node number is [1,1,2]; its level is 3 and father-matching KB clauses are {w-f},{t} and {s}.

As in the depth-first method, we will divide the main refutation procedures for a breadth-first method into six levels, i.e from the highest level 1 down to the lowest or the deepest level 5, as follows:

[1]. Procedure ANSBF.

[2]. Procedures ASKBF and BF.

[3]. Procedures BF0 and STOP-TEST.

[4]. Procedures BF1 and others.

[5]. Procedure BF2.

3.3.3.3.1. Level 1: Procedure ANSBF

The highest level procedure in a breadth-first method described here is a procedure ANSBF as shown in Program 3.3.3.16 below. This procedure is equivalent to procedure ANS (see Program 3.3.3.8 in Level 1 of section 3.3.3.2) and it is also called from procedure AS (see Program 3.3.3.3).

```
/* procedure ANSBF */
answer(Quest,yes):-
        /* procedure ANSBF.1 */
        assertz_new(node_no(0)),        /* initialise node_no(0) */
        asking_bf(Quest),               /* procedure ASKBF */
        breadth_first(1).               /* procedure BF */
answer(Quest,no):-
        /* procedure ANSBF.2 */
        write('no').
```

Program 3.3.3.16: Procedure ANSBF

Procedure ANSBF (see the above Program 3.3.3.16) is subdivided into two subprocedures. The first one, i.e procedure ANSBF.1, will first assert an initial value of node number, i.e predicate *node_no(0)*, in the database, then will set a starting proving tree (procedure ASKBF) and finally will find a solution to the question (procedure BF). Eventually this procedure, ANSBF.1, will return a remark "yes" if the proof is successful, i.e procedure BF is able

to find an empty resolvent. The second one, procedure ANSBF.2, will return a remark "no" if procedure BF is not able to find an empty resolvent or in simple words that this procedure (BF) fails.

3.3.3.3.2. Level 2: Procedures ASKBF and BF

There are two main procedures at this level (level 2). These are procedures ASKBF and BF where both of them are called from procedure ANSBF at level 1 above.

Level 2.1 Procedure ASKBF

This procedure ASKBF (see Program 3.3.3.17 below) is almost equivalent to procedure ASK (see Program 3.3.3.9 at Level 2 of section 3.3.3.2). Their differences are the order of goal formatting. Procedure ASK will format one goal clause and then prove it. On the other hand, procedure ASKBF will format all goal clauses and prove all of them at the same time.

```
/* procedure ASKBF */
asking_bf([Quest/Quest1]):-
        /* procedure ASKBF.1 */
        format_goal(Quest,Goal),        /* procedure FG */
        update_node_no(N),              /* procedure UPNOBNO */
        assertz(node(Goal,[N],1,[])),
        asking_bf(Quest1).
asking_bf([]):-
        /* procedure ASKBF.2 */
        !.
```

Program 3.3.3.17: Procedure ASKBF

For every literal goal of the question ,procedure ASKBF, as shown in Program 3.3.3.17, will format a goal from a given query clause (procedure FG); update node number *node_no(N)* and finally assert predicate *node(Goal,[N],1,[])* (i.e a temporary representation of the

goal node) into the database. The father-matching KB clauses, in this case, is an empty clause as the goal clause is not a resolvent one. However, predicate *node/4* is used here to represent every goal clause because that at the starting point of proving process, each goal clause can be considered as a son of the question.

The second procedure, ASKBF.2, does not *fail* as opposed to procedure ASK.3 (see Program 3.3.3.9). The adopted breadth-first method tries to resolve every query clause each time. Thus, at the end of setting up initial proving tree, the proving have not started yet, consequently procedure ASKBF must succeed or return *true* value. On the contrary, the adopted depth-first method resolves one query clause each time. So, procedure ASK.3 *fail* in order to show that the proving of each query clause has finished.

Referring back to Figure 3.3.3.6 at the beginning of section 3.3.3.3, this procedure ASKBF will create all first level nodes of the solution or proving tree, i.e nodes [1] and [2]. These nodes are actually goal clauses formatted by procedure FG (see procedure ASKBF.1 of Program 3.3.3.17).

Level 2.2: Procedure BF

After the temporary predicate *node/4* have been asserted into the database for every goal clause or the first level of proving tree have been set up, predicate *breadth_first(1)* (or procedure BF) is called from procedure ANSBF.1 (see Program 3.3.3.16). The argument of

predicate *breadth_first/1* is a level number of the solution tree, thus the initial value of level number is 1 (one). We split procedure BF into two subprocedures, i.e procedure BF.1 and BF.2 as shown in Program 3.3.3.18.

```
/* procedure BF */
breadth_first(Levelno):-
        /* procedure BF.1 */
        retract(node(Q,Nodeno,Levelno,Qfact)),
        assertz(c_node(Q,Nodeno,Levelno,Qfact)),
        assertz_new(node_no(0)),              /* initialise node number */
        Level1 is Levelno + 1,
        breadth_first0(Q,Nodeno,Level1).      /* procedure BF0 */
breadth_first(Levelno):-
        /* procedure BF.2 */
        Level1 is Levelno + 1,
        level_limit(Upperbound)
        stopping_test(Level1,Upperbound),     /* Procedure STOP-TEST */
        breadth_first(Level1).
```

Program 3.3.3.18: Procedure BF

The first one, procedure BF.1, will retract from the database one by one temporary predicate *node/4* (a son of the previous goal clause) and assert them back into the database but this time with predicate *c_node/4* to show that this son now will become a new (current) goal clause. Then an initial value of node number denoted by predicate *node_no(0)* is asserted into the database and the value of level number, "Levelno", is also temporarily updated as the new resolvent, if it exists, is at one level down. In other words, the new resolvent is a son of the goal clause.

Finally, in procedure BF.1, the goal clause is matched with others KB clauses by calling procedure BF0 or predicate *breadth_first/4*. The procedure BF0 will eventually fail if no solution is reached and backtracking will occur. So it (procedure BF.1) will try to generate new resolvents from new goal clauses until no more

predicate *node/4* exists at a current level, "Levelno". If this happens then the second procedure ,BF.2, will be called.

The second procedure, BF.2, will update the current level number and then a stopping test is carried out (procedure STOP-TEST). If this procedure STOP-TEST succeeds, then the proving process will continue to find the next generation of resolvents, otherwise the proving will stop and consequently procedure BF will fail and so too procedure ANSBF (see Program 3.3.3.16).

3.3.3.3.3. Level 3: Procedures BF0 and STOP-TEST

At this level 3, there two major procedures which are called from procedure BF at level 2 above. These procedures are BF0 and STOP-TEST.

Level 3.1: Procedure STOP-TEST

After all new resolvents have been generated by procedure BF.1, the stopping test (procedure STOP-TEST) is carried out (see procedure BF.2). The procedure STOP-TEST (see Program 3.3.3.19 below) are based on two stopping criteria. These are an upperbound limit of the level of the proving tree and the existence of any new resolvents (sons) of the goal clause at the current level, i.e the existence of predicate *node/4* at a new updated level, *node(_,_,Level1,_)*.

```
/* procedure STOP-TEST */
stopping_test(Level1,Upperbound):-
        /* procedure STOP-TEST.1 */
        Level1 =< Upperbound,
        exists(node(Q,N,Level1,Qf)),
        !.
stopping_test(Level1,Upperbound):-
        /* procedure STOP-TEST.2 */
        Level1 =< Upperbound,
        not_exists(node(Q,N,Level1,Qf)),
        assertz(toptry([])),
        !,fail.
stopping_test(Level1,Upperbound):-
        /* procedure STOP-TEST.3 */
        Level1 > Upperbound,
        exists(node(Q,N,Level1,Qf)),
        assertz(toptry([])),
        assertz(reach_limit(Upperbound)),
        fail.
stopping_test(Level1,Upperbound):-
        /* procedure STOP-TEST.4 */
        reach_limit(Upperbound),
        ask_new_limit,                    /* Procedure ASK-NEWLIMIT */
        level_limit(Upperbound1),
        Upperbound\=Upperbound1,
        stopping_test(Level1,Upperbound1). /* Procedure STOP-TEST */
```

Program 3.3.3.19: Procedure STOP-TEST

Although there are two stopping criteria, the procedure STOP-TEST is divided into four subprocedures. The first subprocedure STOP-TEST.1 will succeed when the level of proving tree has not yet reached its upper limit (upperbound) value and no solution has been found such that the proving should be continued (see procedure BF.2 of Program 3.3.3.18).

The second subprocedure STOP-TEST.2 will be set to fail when no solution has been found although the level of the proving tree has not yet reach its upperbound value. This happens when no new resolvent can be generated from all goal clauses. In this case the proving process should be stop immediately.

The third subprocedure STOP-TEST.3 will also be set to fail when the level of proving tree has reached its upperbound value and no solution has been found or more solutions are needed. This procedure is set to fail such that the fourth subprocedure can be carried out.

The fourth subprocedure, STOP-TEST.4 will be processed when the upperbound value of the level of proving tree has been reached (i.e subprocedure STOP-TEST.3 fails). In this case, the user will be asked whether to increase the upperbound value of the level or just stop there (procedure ASK-NEWLIMIT). In the case of increasing the upperbound value of the level, the proving process will continue as it has not yet reached its upperbound value. Otherwise, the proving will immediately stop. For example:

```
?-listing(knowledge).
knowledge(human(_1):-man(_1)).
knowledge(human(_1):-woman(_1)).
knowledge(man(aizat)).

?-pcquest.
|: human(aizat).
        .
        .
We cannot prove the question until level 1
Do you like to update upperbound level limit
from its current value (y/n) ? y

Type new value (followed by dot(.) and (return)):
4.

))Answer: Yes,
        " human(aizat) "
==============================

        .
        .
```

Session 3.3.3.1: Example on increasing the upperbound value of the level

The above session shows how the upperbound value of the level is increased from 1 (original value) to 4 (new value) such that the question can be proved or answered. The detail program of procedure ASK-NEWLIMIT can be found in the Appendix.

Level 3.2: Procedure BF0

Procedure BF0 (as shown in Program 3.3.3.20) is also called from procedure BF.1 at level 2. This procedure will resolve a goal clause to form a new resolvent and will be divided into two subprocedures

```
/* procedure BF0 */
breadth_first0(Q,Nodeno,Levelno):-
        /* procedure BF0.1 */
        breadth_first1(Q,Resolvent0,Qfact),          /* procedure BF1 */
        differences(Q,Resolvent0,Resolvent),         /* procedure DIFF */
        update_node(Resolvent,Nodeno,Levelno,Qfact),  /* procedure UPNODE */
        is_solution(Resolvent,Nodeno,Levelno).       /* procedure ISSOLN */
breadth_first0(Q,Nodeno,Levelno):-
        /* procedure BF0.2 */
        node_no(0),                     /* node number = 0 ? */
        graph_praning(Nodeno),          /* procedure GPRUNE */
        fail.
```

Program 3.3.3.20: Procedure BF0

The first subprocedure, BF0.1, is to find a set of new resolvents by matching the goal clause, Q, with the KB clauses (procedure BF1) and then delete any repeating literals which occur in the goal clause to produce a free repeating literal clause, *Resolvent* (procedure DIFF). The predicate *node/4* which contains information about the resultant resolvent, is asserted into the database (procedure UPNODE). Then procedure ISSOLN is called to find out if the new resolvent is an empty clause, thus the proof is successful, so too procedure BF0.

If no solution has been reached at this stage, the dead end goal checking is carried out in order to find out whether it generates any new resolvent (procedure BF0.2 of Program 3.3.3.20). In order to save space, the solution tree is pruned (procedure GPRUNE). Any dead node (branch or goal) will be pruned from the proving graph. The goal is said to be a dead end node (or branch or goal) if

predicate *node_no(0)* exists. In other words, a dead end node or goal is the one without sons. In the end, as the solution has not been reached yet, procedure BF0 will be set to fail such that backtracking will occur in procedure BF.1 (see Program 3.3.3.18).

3.3.3.3.4. Level 4: Procedures BF1 and others.

At level 3, procedures BF1, UPNODE, DIFF, ISSOLN and GPRUNE are called from procedure BF0. At this level we will explain all the procedures except procedure DIFF where we can find the definition in the Appendix.

Level 4.1: Procedures UPNODE and ISSOLN

These two procedures, UPNODE and ISSOLN, are shown in Program 3.3.3.21 below. Procedure UPNODE is to update node number and assert the temporary predicate *node/4* which contains information about the new resolvents, into the database. Procedure ISSOLN will detect whether the new resolvent is an empty clause, if so then the solution is reached and the associated predicate *c_node/4* will be asserted into the database.

```
/* procedure UPNODE */
update_node(Q,[N/Nodeno],Levelno,Qfact):-
        update_node_no(N),      /* procedure UPNODENO :updating node number */
        assertz(node(Q,[N/Nodeno],Levelno,Qfact)),
        !.

/* procedure ISSOLN */
is_solution([],Nodeno,Levelno):-
        node_no(N),
        retract(node([],[N/Nodeno],Levelno,Qfact)),
        assertz(c_node([],[N/Nodeno],Levelno,Qfact)),
        !.
```

Program 3.3.3.21: Procedures UPNODE and ISSOLN

Level 4.2: Procedure GPRUNE

As we recalled that procedure GPRUNE is meant to prune the proving graph in order to save space. This procedure, as shown in Program 3.3.3.22 below, is called from procedure BFO.2 (see Program 3.3.3.20).

```
graph_pruning([N/Nodeno]):-
        /* procedure GPRUNE.1 */
        retract(c_node(Q,[N/Nodeno],Levelno,Qfact)),
        (exists(node(_,[_/Nodeno],Levelno,_)) ;
                exists(c_node(_,[_/Nodeno],Levelno,_))),
        !.
graph_pruning([N/Nodeno]):-
        /* procedure GPRUNE.2 */
        graph_pruning(Nodeno),!.
graph_pruning([]).      /* procedure GPRUNE.3 */
```

Program 3.3.3.22: Procedure GPRUNE

The predicate *c_node/4* which associates with the dead end node goal, will be retracted from the database and so their dead end father goal and so on. The dead end father goal is a goal clause without any son after pruning their dead end son (clause). The dead fathers node will emerge after all their sons have been pruned, i.e all of them are dead nodes. This can be detected by the nonexistence of the predicate *node/4* or *c_node/4* with the same father's node number ,i.e node number [_|Nodeno]. In other words, the dead son does not have any brothers (sisters).

For instance, from Fig. 3.3.3.6, node [2,1] is a dead one, so this node is pruned from the solution tree but its father, node [1], is not a dead one as it has another son, i.e node [1,1]. Another dead end node, i.e node [1,1,2], is also pruned from the proving graph. It is also found out that node [2,1,2] is a dead one, so it will be pruned too. Now its father become a dead father (node [1,2]) as it has no more sons (both of them have been pruned), thus node [1,2] is also pruned from the proving graph.

Level 4.3: Procedure BF1

Procedure BF1, as shown in Program 3.3.3.23, is equivalent to procedure FACTPR (see Program 3.3.3.11). The purpose of this procedure which is called from procedure BF0.1 (see Program 3.3.3.20), is to steer a matching or unifying process from right to the left of the goal clause. Procedures BF1.1 and BF1.2 deal with a goal clause consisting of a conjunction of literals and one literal only respectively. The unifying is carried out by procedure BF2. Any repeated literal is then removed from the resulted resolvent of procedure BF1.1 by calling procedure DIFFERS. List "Qfact" contains the matching KB clauses of the literals goal.

```
breadth_first1((Qh,Qt),Resolvent,Qfact):-
        /* procedure BF1.1 */
        breadth_first1(Qt,Resolvent1,Qtfact),
        breadth_first2(Qh,Resolvent2,Qhfact),   /* procedure BF2 */
        differs(Qh,Resolvent1,Resolvent3),      /* procedure DIFFERS */
        merge(Resolvent2,Resolvent3,Resolvent), /* procedure MERGE */
        append1(Qhfact,Qtfact,Qfact).           /* procedure APPEND1 */
breadth_first1(Q,Resolvent,Qfact):-
        /* procedure BF1.2 */
        Q\==[],
        breadth_first2(Q,Resolvent).    /* procedure BF2 */
```

Program 3.3.3.23: Procedure BF1

3.3.3.3.5. Level 5: Procedure BF2

Procedure BF2 (see Program 3.3.3.24) is to unify the literal goal with any KB clauses. Procedures BF2.1 and BF2.2 will match a literal goal with factual and ruled KB clause respectively with a unification preference to factual KB clause.

```
/* procedure BF2 */
breadth_first2(Q,[],[Q]):-
        /* procedure BF2.1 */
        knowledge_base(Q).
breadth_first2(Q,A,[Q:-A]):-
        /* procedure BF2.2 */
        knowledge_base(Q:-A).
```

Program 3.3.3.24: Procedure BF

3.3.3.4 Comparison between The Depth-First and The Breadth-First Methods.

As it is known that there are some advantages and disadvantages between these two methods of implementation. For simplicity, we refers the depth first and the breadth first methods as algorithm DF and BF respectively in the remaining thesis. The following points are the differences between these two implemented methods in Prolog (see sections 3.3.3.2 and 3.3.3.3 above).

[1]. The termination condition.

As the algorithm DF having a cycling checker which spots cycle, thus it will terminate in most cases when the searching of new resolvents is exhausted or it encounters the empty resolvent. While algorithm BF will terminate when it first encounters the empty resolvent (in all cases) or the upperbound value of the specified level of its proving tree is reached (due to the space). Thus algorithm BF does not guarantee that all answers have been extracted when it terminates as we cannot unspecify the number of level of the proving graph due to the space problem. Although we can increase the upperbound value of the level of proving tree during the proving session, but the space and cycling problem are still there. Furthermore at the end of proving, algorithm DF guarantees that the searching is exhausted. On the other hand, algorithm BF does not guarantee the search is exhausted although it does not derive any conclusion.

[2]. The answer extraction procedure.

By using the algorithm DF, the answer can be extracted in natural due to the backward chaining of the Prolog implementation. Conversely, the algorithm BF cannot extract answers in a natural way, thus a special procedure has to be written in order to extract answers. This contradiction can be seen from the following examples:

```
?-pcquest.      /* example [3.3.3.4.1] */
|: exists(X,husband(mary,X)).
                .
                :
>>answer: Yes,
        exists(john,husband(mary,john)).
*********************************
                .
                :
yes

?-pcquest.      /* example [3.3.3.4.2] */
|: reach(X).
                .
                :
>>answer: Yes,
        reach(climb(carry(c,b,walk(a,c,s))))
*********************************
                .
                :
yes
```

Session 3.3.3.2: Solving by using the algorithm DF

In example [3.3.3.4.1] above, the answer by using algorithm DF can be interpreted as "mary has a husband whose name is John". In example [3.3.3.4.2], the answer of algorithm DF, i.e. "reach(climb(carry(c,b,walk(a,c,s)))", can be interpreted as that the monkey can reach the banana by walking from position a to position c, then carry the chair from position c to position b, and finally climb the chair and grab the banana.

```
?-pcquest.      /* example [3.3.3.4.3] */
|: exists(X,husband(mary,X)).
                .
                .
>>answer: Yes,
        exists(_1,husband(mary,_1)).
================================
                .
                .
yes

?-pcquest.      /* example [3.3.3.4.4] */
|: reach(X).
                .
                .
>>answer: Yes,
        reach(_1)
================================
                .
                .
yes
```

Session 3.3.3.3: Solving by using algorithm BF

On the other hand, by using algorithm BF, both answers of examples [3.3.3.4.3] and [3.3.3.4.4] which are equivalent to examples [3.3.3.4.1] and [3.3.3.4.2] respectively cannot be interpreted or extracted easily other than showing that mary has a husband and there exists a way of reaching the banana respectively.

[3] The cycle checking.

As algorithm BF guarantees, if the space permitted, that the empty clause can be derived if exists, then there is no need to implement cycle checking. Moreover, the cycle checking procedure is quite difficult to implement efficiently in algorithm BF. On the other hand, there is a possibility that a cycle exists if algorithm DF is adopted since the cycle (infinite) path may be explored first. Hence the cycle checking is implemented in algorithm DF to make sure that the empty clause can be derived, if it exists.

[4]. The space problem.

As algorithm BF will generate all possible resolvent (son) clauses, there is a possibility that the space is not enough. So, the failure resolvent (dead son) is pruned from the refutation tree. On the other hand, algorithm DF naturally will generate only one subtree at any time then there will not have any space problem.

[5] The average number of resolvents generated.

It is quite difficult to compare the average number of resolvents generated as both methods resolve the literal goal in different ways. For examples:

```
?-listing(knowledge).
knowledge(p:-a).        /* clause (k1) */
knowledge(p:-b).        /* clause (k2) */
knowledge(q:-c).        /* clause (k3) */
knowledge(q:-d).        /* clause (k4) */
knowledge(r:-e).        /* clause (k5) */
knowledge(r:-f).        /* clause (k6) */
knowledge(a:-g).        /* clause (k7) */
knowledge(a).           /* clause (k8) */
knowledge(c).           /* clause (k9) */
knowledge(f).           /* clause (k10) */

?-pcquest.
:: p & q & r
        .
        .
```

Session 3.3.3.4: The listing of KB clauses

And suppose we would like to prove "p & q & r" (as shown in the above session 3.3.3.4 by omitting the answer) assuming that the KB clauses are also as shown in the above session. By using algorithm DF, the refutation graph of proving "p & q & r" or equivalently "[]:-p,q,r" (the goal clause) is as follows:

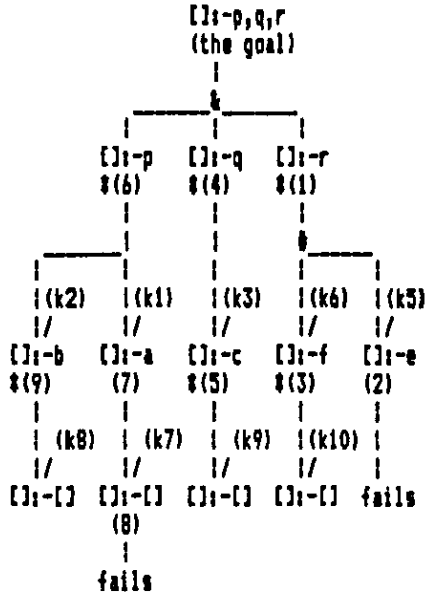


Fig. 3.3.3.1 A refutation graph of algorithm DF

The solution arrived after 9 resolvents have been generated and its path is marked by "*". The number at the goal is the order (node) number of the refutation process and the number at the branch, for example (k9), refers to KB clauses number (see session 3.3.3.4). The signs "&" and "#" at a branch node refer to AND and OR node respectively. Now let see how the proving graph generated by algorithm BF:

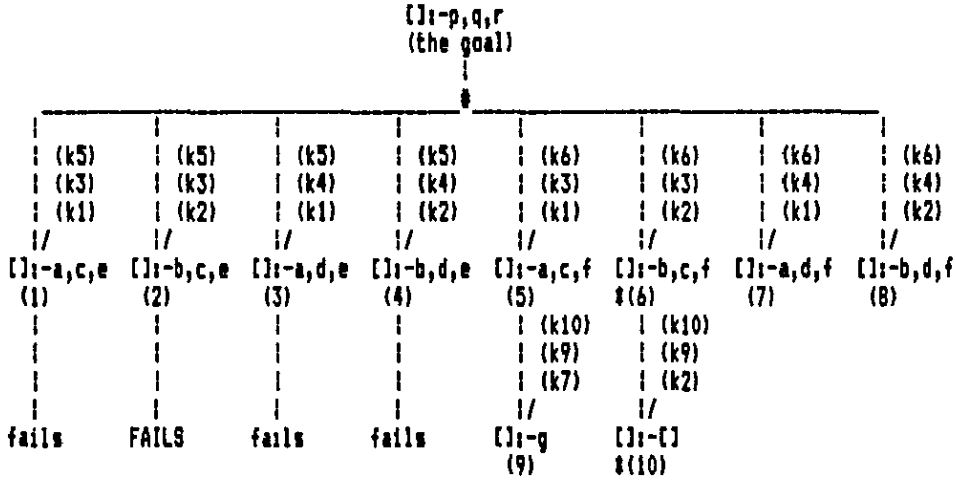


Fig 3.3.3.2: A refutation graph of algorithm BF

From the above figure, Fig. 3.3.3.2, the empty clause (solution) has been derived after generating 10 resolvents and at the second level of the refutation graph and its solution path is also marked by "*". As explained before algorithm BF will stop after it encounters the first empty resolvent, in this case, the resolvent (10). The notations used here are the same as in Fig 3.3.3.1 above.

From the above two figures (Fig. 3.3.3.1 and 3.3.3.2), algorithms DF and BF arrive at the solution after generating 9 and 10 resolvents respectively. This number cannot deduce anything relevant to the number of resolvent generated. Moreover, the number of resolvents generated during proving process depends on the arrangement of the knowledge base clauses. Both algorithm will match from top to the bottom of the list of KB clauses.

For example, if we rearrange the KB clauses in the order of (k2),(k1),(k3),(k4),(k6),(k5,(k7),(k8),(k9) and (k10), we will get a different number of resolvents generated before deriving the empty clause. Thus, algorithms DF and BF will generate 6 and 9 resolvents before arriving at the solution.

Another point is that algorithm DF will match one KB clause at a time to produce one resolvent. On the other hand, algorithm BF will match or unify with more than one KB clauses to produce one resolvent at a time.

Theoretically and also practically it is very difficult to compare these two algorithms DF and BF even we would like

to use the average number of resolvents generated as we have to rearranging the order of KB clauses. In proving "p & q & r", the average number of resolvents generated for algorithm DF and BF are 8 and 19.5 respectively.

In general, it can be seen that algorithm DF will give a lower value for the average number of resolvents generated (or matching clauses) compared to algorithm BF. And it is equal when there is only one alternative of the matching KB clause. This is due to the fact that algorithm BF will match more clauses in order to generate a new resolvent as shown in the above Fig 3.3.3.2.

[6]. The occur checking.
Both algorithm DF and BF cannot handle the occur checking as the unification of the variables is carried out automatically by the Prolog. The POPLOG Prolog which we use does not implement the occur checking.

[7]. The incompleteness.
Both algorithms DF and BF are incomplete as both adopt the linear input resolution strategy. They are incomplete in the sense that they do not derive an empty clause where they should do. For example we cannot derive an empty clause from the following premises: "q(X) # p(a)", "~q(X) # p(X)", "~q(X) # ~p(X)" and "q(X) # ~p(X)" by using both algorithms, although we know that these premises are inconsistent.

3.4 Comment and Conclusion.

The Prolog-based resolution which has been described above can be used in a deductive question answering system as well as in a problem solving system. It can answer all four classes of questions as classified by Chang & Lee[1973]. However there is some restriction on the use of these procedures. As described before that the instantiation or unification method is based on Prolog's unification, therefore there is no occur checking in the procedure.

In these procedures, the statements, either knowledge or query, are in the form of first order logic or predicate calculus. However the procedures itself are flexible in the sense that it can be modified to suit a second order predicate calculus. This is achieved by changing the representation of the predicate. Thus, for instance, predicate "man(X)" can be written as "f(man,X)". In fact, the procedures do not need any changes if the predicate is written by using predicate "f" as described except in the procedure of generating symbol names for Skolem function in replacing existential quantifiers.

Another interesting point is that algorithm DF can extract information easily as required. This can clearly be seen from the examples [3.3.3.4.1] and [3.3 3.4.2] above.

As there are cases in which a refutation exists but the Prolog-based refutation does not; therefore Prolog-based strategies are not complete. This is due to the adoption of a linear input form strategy in the Prolog-based refutation

procedures as it is known that the incompleteness property of the linear input strategy. However, as the procedure is used mainly for a deductive question-answering system and not for proving a set is unsatisfiable, the lack of completeness cases are quite seldom due to the fact that the set of knowledge clauses is usually unsatisfiable.

We have discussed how to assert facts or knowledge by using procedure PCFACT (see Program 3.3.1.5) and how to ask the question by using procedure PCQUEST (see Program 3.3.3.1). Another extra feature of procedure PCQUEST is that we can assert the question into the database as knowledge or fact (KB clauses) if it can be proved. This feature can be used to find out whether any statements can be deduced from the database before we assert them as knowledge into the database. Thus our database will not grow unnecessarily large if we use this extra feature of procedure PCQUEST. In this case, we actually do not need procedure PCFACT anymore.

Apart from incompleteness and occur check problems, the Prolog-based procedures seem quite efficient. We can also see that the translation into Horn clauses acts like a Prolog program generator while the proving procedures act like a Prolog interpreter.

CHAPTER 4

NATURAL LANGUAGE INTERFACING

4.1 Introduction

Pereira and Warren [1980] have proposed a method for expressing grammars in logic, which is a natural extension of a context free grammar and is called "Definite Clause Grammars" (DCGs). Winograd [1983] also pointed out that a context free grammar stated in this form (DCGs) can be used for parsing with a theorem prover. Following the work described in Pereira and Warren [1980] above on translating natural language into 1st order predicate calculus and from there into Horn clause format (Clocksin & Mellish [1981]), it is clear that this could form the basis of an intelligent front end (Bundy et al. [1983]).

Bennett et al.[1986] divided the question-answering system into two types, from the viewpoint of the relation between the natural language processor and the data stored, as follows:

[1]. Integrated systems: the data structures are conceived with the natural language (NL) processor in mind, and/or vice versa.

[2]. Front ends: which can be plugged on to the 'front' of a range of data retrievel systems to provide an NL interface.

The question-answering system which will be discussed in this chapter is classified as the second one, i.e the front ends type. These front ends will analyse the questions in a natural language (source language), and as it were 'translate' the source language firstly into a predicate calculus (PC) and later into Horn clauses to answer the

particular question and the resulting PC will then be synthesized back into the source language.

In this chapter, a study of this method (DCGs) as a base for a question-answering system is carried out. The question-answering system as described in the chapter 3 is based on a mechanical theorem prover, thus DCGs is chosen for a natural language interfacing. It should be noted here that as English is not my mother tongue language, a natural language interfacing discussed here is based on a subset of English language. The main aim here is to study how this grammar is used in analysing an English sentence (source language) into a PC and synthesizing from answered PC into an English sentence (target language). Much of the emphasise is placed on the technique for analysis into and synthesis from a PC. The source and target languages can be different but in this study both the source and target languages are the same, i.e English language. This can be summarised as follows:

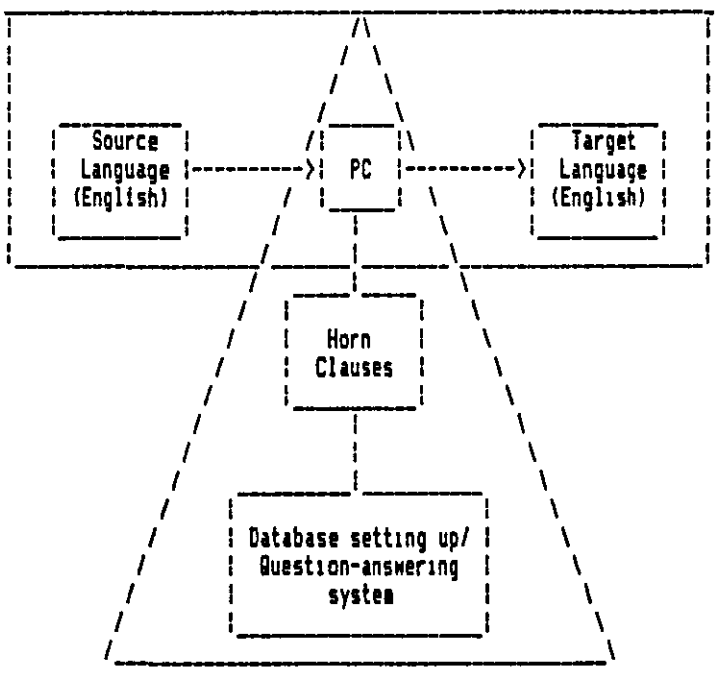


Figure 4.1.1: The described system.

From the previous figure, Fig. 4.1.1, the triangle represents the mechanical theorem prover described in chapter 3 where the input is a PC and the output is an instantiated (answered) PC. The rectangle represents a phase of analysing and synthesizing an English sentence which will be described in this chapter. The rectangle and triangles come together when the resulting PC of analysing an English sentence (source language) is passed into the triangle which will then pass back the answered PC back into the rectangle for synthesizing back into English or another target language.

The original DCGs system were described in Pereira and Warren [1980], and Clocksin and Mellish[1981]. Later Hinde[1983] made some modification to suit the problem in his Fuzzy Prolog. Then further modifications are made in order to make the system works from an English sentence to PC and back again to an English sentence. In other words, the grammar must work for translating a question (of course in English language) to PC and the question is also answered in English language.

4.2 Analysing an English sentence into Horn clauses

The question-answering system described in chapter 3 is based on Horn clauses. So, an English sentence is analysed into Horn clauses. We will divide the process of analysing English sentence into Horn clauses into two stages as follows:

[1]. Analysing an English sentence into PC.

[2]. Transforming PC into Horn clauses.

4.2.1 Analysing an English sentence into PC.

The program or grammar to analyse an English sentence is based on Hinde[1983]. The PC representation has been modified as he used in Fuzzy Prolog. It can parse a simple English sentence into PC. For examples:

```
[4.2.1] john loves mary.
        ==========>
        loves(john,mary).

[4.2.2] every man who loves every woman likes every fish.
        ==========>
        all(_1,man(_1)&all(_2,woman(_2)=>loves(_1,_2))=>all(_3,fish(_3)=>likes(_1,_3))
```

Originally it could parse or translate a subset of English sentences including the following syntactic relations:

(1). Noun phrase:

- -proper noun
- -determiner (a & an) and noun phrase
- -regular and irregular plural noun form.

(2). Verb phrase:

- -transitive verbs
- -intransitive verbs

(3). Relative clause:

- - who

Later, a few additions have been made to enhance the capability of the grammar itself. These additions can be made easily without any amendments to the original grammar. The following additions have been made:

(1a). Noun phrase:

-proper noun :

- any word starting with capital letters, e.g John, Aizat etc.
- word "who" as used in a question-answering system, e.g "who is kind".

-determiner:

- existential quantifier: the, some
- negation of existential quantifier: no

-quantified pronoun:

- existential quantifier: somebody, someone.
- universal quantifier: everybody, everyone,
- negation of existential quantifier: nobody.

(2a). Conjunction:

- and, or,
- neither...nor..
- either...or..
- both...and...

(3a). Relative pronouns:

- that, which.

In parallel to the above additions, the size of the dictionary has also been increased by adding new words according to their classes and also according to the type of the sentence which can be accepted in general. The classes of the words in the dictionary mostly are as described in McArthur [1981].

In general, there is no major problem at all in adding new grammar rules or words as long as the same representations of grammar rules and predicate calculus are maintained. The

above additions are made on the basis of the type of examples in the chapter 2. In other words, it can be said that the additions have been made on an ad hoc basis.

It should be noted here that, in order to accept any word starting with capital letters as a word of the sentence but not as a variable, the program which reads the sentence has to be modified such that it will translate any word starting with a capital letter as an atom but not as a variable in the sense of Prolog term (see example [4.2.4] below and compared with example [4.2.1] above). Another addition is to accept any word starting with "$" as a variable in order to distinguish the word which starting with capital letters as a proper noun (see example [4.2.8] below).

The program to read any sentence is called *read_in* and its detail can be found in the Appendix. It should be noted here that the equivalent program *read_in* has been made as a library program in POPLOG Prolog, but it does not have the capability of accepting either a proper noun starting with a capital letter or any word starting with "$" to denote a variable.

The following are some examples of the analysis of English sentence into PC:

```
[4.2.3]. No wombat who lives in a zoo is happy.
         ==========>
         *exists(_1,indefinite(_1),wombat(_1)&exists(_2,zoo(_2)&in(_1,_2)&lives(_1))&happy(_1)).

[4.2.4]. John loves Mary.
         ==========>
         loves(John,Mary).
```

```
[4.2.5]. everyone who is both strong and intelligent succeeds.
         ===========>
         all(_1,person(_1)&strong(_1)&intelligent(_1)=>succeeds(_1)).

[4.2.6]. Does Peter succeeds?
         ===========>
         succeeds(Peter).

[4.2.7]. who is kind?
         ===========>
         kind(_1).

[4.2.8]. $J loves Mary?
         ===========>
         loves(_1,Mary).
```

The grammar also accepts an English sentence in the form of a question as shown in example [4.2.6] above. This form of sentence (question) will be used in a question-answering system which will be explained in the next section.

As we need to differentiate between determiners "the" and "a" in the PC representation of a sentence, we will add another information in the PC, i.e "definite(X)" and "indefinite(X)" to denote definite and indefinite articles. These addition is quite important especially in translating PC into Horn clauses such that we can generate an appropriate symbol for each class of articles. We also add these two information in the PC representation of other quantifiers (e.g "all", "somebody" etc). For examples:

```
[4.2.9]. somebody visits a zoo.

         ==========>

         exists(_1,indefinite(_1),person(_1)&exists(_2,indefinite(_2),zoo(_2)&visits(_1,_2))).


[4.2.10]. somebody visits the zoo.

          ==========>

          exists(_1,indefinite(_1),person(_1)&exists(_2,definite(_2),zoo(_2)&visits(_1,_2))).
```

[4.2.11]. every man who loves every woman likes every fish.

==========>

all(_1,indefinite(_1),man(_1)&all(_2,indefinite(_2),woman(_2)=>loves(_1,_2))=>
all(_3,indefinite(_3),fish(_3)=>likes(_1,_3))

Examples [4.2.9] and [4.2.10] show the difference between the PC representation of determiners "a" and "the" respectively. Example [4.2.11] shows the new representation of determiner "all" (compared this with example [4.2.2]).

4.2.2 Transforming PC into Horn clauses

Why we need to transform PC into Horn clauses? As explained in the first paragraph of the section 4.2, the database is in the form of Horn clauses and the theorem proving mechanism as explained in the chapter 3 accepts an input (question) in the form of PC and subsequently transforms them into Horn clauses. So that is why we need to transform the PC into Horn clauses.

In section 3.2 of chapter 3, we have already explained how to convert PC statement into Horn clauses. In analysing an English sentence, the need to change the PC representation has arisen especially in dealing with definite and indefinite determiners.

The new rules to deal with this new PC representation are to be added to the programs which process the quantifiers. The affected programs are those in the first four stages, i.e. removing all implication and equivalence signs, moving

negation inwards, skolemization and finally moving outwards and eliminating universal quantifiers stages.

We need only to add new rules to deal with the new PC representation of the forms "all(X,D,P)" and "exists(X,D,P)" where D is a either "definite(X)" or "indefinite(X)" in all first four stages. Compared this with the old form of PC representations, i.e "all(X,P)" and "exists(X,P)" to denote universal and existential quantifiers respectively (see section 3.2 of chapter 3). The new rules are equivalent to those dealing with the old form of representing quantifiers, for examples in moving out all universal quantifiers (Stage 4):

```
anivout(all(X,P),P1):-
        !,              /* the old form (rule) */
        anivout(P,P1).
anivout(all(X,D,P),P1):-
        !,              /* the new form (rule) */
        anivout(P,P1).
```

We can see the similarity in those two rules above. The same also applies in the first three stages i.e removing all implication and equivalence signs, moving negation signs inwards (for universal quantifiers only) and the skolemization of universal quantifiers for either knowledge or query clauses (see section 3.3.1 of chapter 3).

The procedure to deal the indefinite and definite existential quantifiers in moving negation inwards are somewhat different. The method described in section 3.2 of chapter 2 still applies to moving negation inwards for indefinite existential quantifiers. As a definite existential quantifier or a definite determiner refers to a subject which is known before hand, thus its property of

existential quantifier must be retained as assuming that the subject is a proper noun. The following are some rules which are concerned with the indefinite and definite existential quantifiers in moving inwards the negation signs:

```
neg(exists(X,P),all(X,P1):-
        !,
        neg(P,P1).
neg(exists(X,definite(X),P),exists(X,definite(X),P1):-
        !,
        neg(P,P1).
neg(exists(X,indefinite(X),P),all(X,indefinite(X),P1):-
        !,
        neg(P,P1).
neg(all(X,B,P),exists(X,B,P1):-
        !,
        neg(P,P1).
```

The complete program of moving in negation signs can be found in the Appendix.

The other irregularity of dealing the definite and indefinite quantifiers is in generating Skolem variables for them. The method of generating the Skolem variables to replace existential quantifiers of the form of "exists(X,indefinite(X),P)" and "exists(X,definite(X),P)" are different for both query and knowledge clauses. For example, the following are two rules to generate Skolem variables for knowledge clauses of both forms.

```
skolem(exists(X,definite(X),P),P2,Vars):-
        !,      /* for definite determiners */
        picknane(X,P,Name),
        pickskolem(Name,the,Vars,Sk),  /*  picking a symbol name */
        substitute(Sk,X,P,P1),
        skolem(P1,P2,Vars).
skolem(exists(X,D,P),P2,Vars):-
        !,      /* for other (indefinite) determiners */
        picknane(X,P,Name),
        pickskolem(Name,others,Vars,Sk), /* picking a symbol name */
        substitute(Sk,X,P,P1),
        skolem(P1,P2,Vars).
```

As in the example before this, we can see from the above example that both rules are equivalent except in picking the

symbol names for Skolem variables (predicate *pickskolem/4*). Let us see in details how to pick up a symbol name for both query and knowledge base (KB) clauses:

4.2.2.1 Knowledge base (KB) clauses

As explained in section 3.3.1.1 of chapter 3, "fs" is an indicator to denote a Skolem function for knowledge clauses. Here, we need to change the definition of procedure PICKSK (program 3.3.4 of section 3.3.1.1) to accommodate the different method of generating symbols for Skolem variables. We will generate a symbol the same way as before (section 3.3.1.1) for other (indefinite) determiners, i.e a new number will be appended to a picked symbol.

However, for the definite determiners, a new symbol will not be generated, but instead the immediately previous one is taken to replace the definite existential quantifiers. This method is adopted because the determiner "the" usually refers to an immediately previous subject. However this method is used with some precautions from the semantic point of view. So the new procedure of picking a symbol for knowledge Skolem function (procedure PICKSK) is as follows:

```
pickskolem(Name,the,Vars,Sk):-
        /* procedure PICKSK.1 */
        skolem_the(Name,Sk),
        !.
pickskolem(Name,Others,Vars,Sk):-
        /* procedure PICKSK.2 */
        gensym(Name,F),
        append([F],Vars,Fandargs),
        Sk=..[fs|Fandargs],
        asserting(skolem_the(Name,Sk)),!.  /* procedure ASSERTING */
```

Program 4.2.1: The new procedure PICKSK

The first procedure (PICKSK.1) takes the latest symbol generated for the same existential quantifier or for the same symbol name "Name". The second one (PICKSK.2) generates the symbol for the existential quantifiers and the assert it into the database by procedure ASSERTING. The procedure ASSERTING will replace the old predicate "*skolem_the(Name,Sk)*" of the same symbol name "Name" with a new one. For examples (an English sentence into Horn clauses):

```
[4.2.12]. A wombat lives in a zoo.
          ==========>
          exists(X,indefinite(X),wombat(X)&exists(Z,indefinite(Z),zoo(Z)&in(X,Z)&lives(X)))
          ==========>
          wombat(fs(wombat0)).
          zoo(fs(zoo0)).
          in(fs(wombat0),fs(zoo0)).
          lives(fs(wombat0)).

[4.2.13]. Nazrul visits the zoo.
          ==========>
          exists(X,definite(X),zoo(X)&visits(Nazrul,X)).
          ==========>
          zoo(fs(zoo0)).
          visits(Nazrul,fs(zoo0)).

[4.2.14]. A wombat lives in a jungle.
          ==========>
          exists(X,indefinite(X),wombat(X)&exists(Z,indefinite(Z),jungle(Z)&in(X,Z)&lives(X)))
          ==========>
          wombat(fs(wombat1)).
          jungle(fs(jungle0)).
          in(fs(wombat0),fs(jungle0)).
          lives(fs(wombat0)).
```

"the zoo" in example [4.2.13] refers to the same "zoo" as in example [4.2.12], thus their Skolem functions are the same, i.e "*fs(zoo0)*". On the hand, "a wombat" in examples [4.2.12] and [4.2.14] are certainly different, thus their Skolem functions are different, i.e "*fs(wombat0)*" and "*fs(wombat1)*" respectively.

4.2.2.2 Query clauses

As in section 3.3.1.2 of chapter 3, "fq" is to indicate the Skolem function for query clauses. The method to generate a symbol for definite and indefinite determiners for query clauses is about the same as in the query clauses with one extra rule for definite determiners. The symbol for a definite determiner is taken from the latest symbol for knowledge clauses, i.e *skolem_the(Name,Sk)* (procedure PICKSKQ.1) or if it does not exists, the symbol is taken from the latest symbol for query clauses, i.e *skolemq_the(Name,Sk)* (procedure PICKSKQ.2) . Otherwise the same method of generating symbol is adopted (procedures PICKSKQ.3 and PICKSKQ.4 for definite and indefinite determiners respectively). The following procedure is to replace the old one (Program 3.3.7 of section 3.3.1.2) and will generate a symbol for query Skolem function of existential quantifiers.

```
pickskolemq(Name,the,Vars,Sk):-
        /* procedure PICKSKQ.1 */
        skolem_the(Name,Sk),
        !.
pickskolemq(Name,the,Vars,Sk):-
        /* procedure PICKSKQ.2 */
        skolemq_the(Name,Sk),
        !.
pickskolemq(Name,the,Vars,Sk):-
        /* procedure PICKSKQ.3 */
        gensym(Name,F),
        append([F],Vars,Fandargs),
        Sk=..[fs|Fandargs],
        asserting(skolem_the(Name,Sk)),!.   /* procedure ASSERTING */
pickskolemq(Name,Others,Vars,Sk):-
        /* procedure PICKSKQ.4 */
        gensym(Name,F),
        append([F],Vars,Fandargs),
        Sk=..[fq|Fandargs],
        asserting(skolemq_the(Name,Sk)),!.  /* procedure ASSERTING */
```

Program 4.2.2: The new procedure PICKSKQ

The following are examples of how English sentences are analysed into Horn clauses and suppose that we have already asserted the knowledge clauses as in the example [4.2.12] above in the database:

```
[4.2.15]. No wombat lives in the zoo ?
          ==========>
          ~exists(X,indefinite(X),wombat(X)&exists(Z,definite(Z),zoo(Z)&in(X,Z)&lives(X)).
          ==========>
          The transformation of the negation of the PC:

          wombat(fq(wombat0)).
          zoo(fs(zoo0)).
          in(fq(wombat0),fs(zoo0)).
          lives(fq(wombat0)).

[4.2.16]. All wombats are animals ?
          ==========>
          all(X,indefinite(X),wombat(X)=>animal(X))
          ==========>
          The transformation of the negation of the PC:

          wombat(fq(wombat1)).
          ~animal(fq(wombat1)).

[4.2.17]. The boy loves Kitkat?
          ==========>
          exists(X,definite(X),boy(X)&loves(X,Kitkat)).
          ==========>
          The transformation of the negation of the PC:

          []:-boy(fs(boy0)),loves(fs(boy0),Kitkat)).
```

From example [4.2.15], "the zoo" refers to the previous zoo, i.e the zoo as referred in the example [4.2.12], thus the questioned Skolem function for "the zoo" in example [4.2.15] is "*fs(zoo0)*". On the hand, the negation of the PC of example [4.2.16] gives an existential quantifier of a wombat, accordingly the questioned Skolem function which replaces it, is "*fq(wombat1)*" as this wombat "*fq(wombat1)*" has not been referred to before thus it takes a new symbol name and furthermore that "*fq(wombat1)*" is a result of the negation of a universal quantifier.

Example [4.2.17] shows that "the boy" is replaced with "*fs(boy0)*" which has been symbolised by procedure PICKSKQ.3 as both the first two procedures (i.e PICKSKQ.1 and PICKSKQ.2) fail. In this case "the boy" is not replaced by questioned Skolem function indicated by "fq" but instead it is replaced by knowledge Skolem function indicated by "fs". This is to show that "*fs(boy0)*" is not a result of the negation of the question so it cannot be replaced by "fq".

The details of programs for the first four stages after adding new rules and modifying some rules as explained above can be found in the Appendix.

4.3 Interfacing an English grammar into the question-answering system

As explained in chapter 3, the refutation process of the Prolog-based resolution is divided into three stages, i.e setting up a database (knowledge clauses), formatting a goal (in Horn clause form), and the refutation procedure itself. Basically, the method of converting a PC into Horn clauses and the refutation procedure will be the same as explained chapter 3 with some exceptions which will be explained in due course.

The Prolog-based resolution described in chapter 3 accepts input in the form of a predicate calculus. So in interfacing an English language grammar to the question-answering system (the resolution method), the input in English sentence will be accepted. However we would like to maintain that the input and output in a predicate calculus are still intact.

As we aim to maintain the question-answering system can accept both an English sentence and a predicate calculus as input, we need to assert in the database (or Prolog system) a control predicate to indicate both inputs such that the answer or output of the question is identical to the input. For this purpose, we create a control predicate *form_of_answer(X)* where X can be instantiated with either "eng" or "pc" to indicate the input and answer (output) are English sentence or a predicate calculus respectively.

The need to differentiate the input or output does not arise in the case of setting up a database or asserting knowledge clauses as we do not need any response at all; the Prolog system will response automatically with "yes" if everything is alright. So we need only to differentiate between those two question inputs.

4.3.1 A predicate calculus input

We need to change the definition of procedure PCQUEST as in the Program 3.3.3.1 of section 3.3.3 of chapter 3. The new definition is as follows:

```
pcquest:-
        read(Q),
        assertz_new(form_of_answer(pc)),
        question(Q).
```

Program 4.3.1: A new definition of procedure PCQUEST

The only difference with the old definition is the second line of the procedure where the new predicate *form_of_answer(pc)* is asserted into the database to replace any predicate *form_of_answer/1* in the database.

4.3.2 An English sentence (language) input

In section 4.2.1, we have described the features of the implemented English language grammar. The top level predicate of the English language grammar described is a predicate *stat_or_quest/2* or procedure SORQ. However procedure PHRASE is the highest level predicate which will called to read and to analyse an input sentence:

```
/* procedure PHRASE */
phrase:-
        read_phrase(S,Y),   /* read an input sentence */
        stat_or_quest(S,Y). /* procedure SORQ */
```

Program 4.3.2: Procedure PHRASE

There are two ways to distinguish whether the English sentence input is a question or not. The first one is either the sentence ends with a question mark ("?") or others ("." or "!"). It is clear if the input sentence ends with a question mark, this will indicate that the input is a question (procedures SORQ.1 and SOR1.3 below). However if the input ends with either a fullstop(".") or an exclamation("!"), then the structure of the sentence will determine whether it is a question to be answered (procedure SORQ.1) or a fact to be asserted into the database (procedure SORQ.2 below). The following is a definition of predicate *stat_or_quest/2* or procedure SORQ:

```
stat_or_quest(Y,Z):-
        /* procedure SORQ.1 */
        question_phrase(Q,Y,[]),        /* procedure QUEST-PH */
        assertz_new(form_of_answer(eng)),
        question(Q).
stat_or_quest(Y,Z):-
        /* procedure SORQ.2 */
        Z==\[?],
        statement_phrase(S,Y,[]),       /* procedure STATE-PH */
        assert_knowledge(S).
stat_or_quest(Y,[?]):-
        /* procedure SORQ.3 */
        assertz_new(form_of_answer(eng)),
        sentence(Q,Y,[]),               /* procedure SENTENCE */
        question(Q).
```

Program 4.3.3: Procedure SORQ

It can be seen from the above program that the predicate *form_of_answer(eng)* is asserted into the database if the input sentence is analysed as a question (procedures SORQ.1 and SORQ 3). All sentences (phrases) are classified into two types, i.e a statement sentence and a question sentence. Procedures STATE-PH and QUEST-PH will analyse statement and question sentences respectively. However, if the sentence input is a question (ended with a question mark), then there are two possibilities of its type, i.e either statement or question sentences, and this question will be processed by procedure SENTENCE as described below:

```
/* procedure SENTENCE */
sentence(Q,Y,[]):-
        /* procedure SENTENCE.1: to analyse a question sentence */
        question_phrase(Q,Y,[]).  /* Procedure QUEST-PH */
sentence(S,Y,[]):-
        /* procedure SENTENCE.2: to analyse a statement sentence */
        statement_phrase(S,Y,[]). /* Procedure STATE-PH */
```

Program 4.3.4: Procedure SENTENCE

The programs of procedures STATE-PH and QUEST-PH can be found in the Appendix.

4.3.3 The output procedure

After the proving has been carried out, the control predicate *form_of_answer(X)* has already be asserted into the database where X has been instantiated with either "pc" or "eng". As explained in chapter 3 that the procedure which prints the answer are procedures PA.1 and AF (see Program 3.3.3.4), procedure PA0.1 (see Program 3.3.3.5) and procedure MORE-ANS (see Program 3.3.3.6). So all these procedures have to be modified accordingly such that the control predicate can be fitted in. The following are the new procedures PA.1, PA0.1 and MORE-ANS:

```
print_answer(Q,Y,[]):-
        /* The new Procedure PA.1: the clause is an inconsistent one */
        affirm(Ans),
        form_of_answer(Formanswer),
        answer_form(Formanswer,Y,S,Ans),        /* procedure AF */
        write(' The question clause is an inconsistent one '),
        !.

print_answer0(Q,Y):-
        /* the new procedure PA0.1: to print "yes" answer */
        nonvar(Y),
        affirm(yes),
        form_of_answer(Formanswer),
        answer_form(Formanswer,Y,S,yes),        /* procedure AF */
        ',more_answer(Formanswer,Y).            /* procedure PA1 */

/* the new procedure MORE-ANS */
more_answer(Formanswer,Y):-
        print_answer1(Formanswer,Y),    /* procedure PRINT-ANS1 */
        get0(A),
        answer_response(A,Y).           /* procedure AR */
```

Program 4.3.5: The new procedures PA.1, PA0.1 and MORE-ANS

As can be seen from the above new procedure, the control predicate has been added to print the answer according the input ("pc" or "eng"). In doing so, there are two new procedures have been written accordingly, i.e procedure STATE-PH1 and PRINT-ANS1. The following are new procedures of AF and STATE-PH1

```
/* the new procedure AF */
answer_form(pc,Y,Y,Ans):-
        /* Procedure AF.1: if the input/output is PC */
        write_answer(user,Ans),         /* procedure WA */
        writepc(user,Y),                /* procedure WPC */
answer_form(eng,Y,Y,Ans):-
        /* Procedure AF.2: if the input/output is English */
        statement_phrase_one(Y,S,[]),   /* procedure STATE-PH1 */
        write_sent(user,S,Y),           /* procedure WENG */

/* procedure STATE-PH1 */
statement_phrase_one(Y,S,[]):-
        statement_phrase(Y,S,[]),
        !.
```

Program 4.3.6: Procedures AF and STATE-PH1

The procedure STATE-PH1 will generate one sentence only even when more answers are required, otherwise the program will give another equivalent sentence and not another answer for

the question. Procedure PRINT-ANS1, as shown below, will print the remark of asking whether another answer is required or not.

```
/* procedure PRINT-ANS1 */
print_answer1(pc,Y):-
        write_proved(user,Y).
print_answer1(eng,Y):-
        write_more(user).
```

Program 4.3.7: Procedure PRINT-ANS1

For examples, suppose that the question is "is nazrul happy?" (in English) or equivalently "happy(nazrul)" (in PC). The system will print the answer accordingly as follows:

```
(a) if the input is a PC:
        .
        :

>>answer: Yes,
        " happy(nazrul) "
************************************

PROVED: happy(nazrul) ? <answered-option>

(b) if the input is an English:
        .
        :

Yes,it is true that Nazrul is happy.

top(phrase): more answers ? <answer-option>
```

Session 4.3.1: The examples of output answers

The remark <answer-option> as shown in the above session (4.3.1) is a user option as explained in Table 3.3.1 of section 3.3.1 of chapter 3.

4.4 Analysing and synthesizing an English sentence into and from PC.

A few problems arise as we use the same set of grammar rules to analyse and synthesis a single sentence into and from PC. Those problems include:

(1) The univ operator "=.." of Prolog.

(2) Errors of using library predicate "*name(X,Y)*" as X or Y must be instantiated. This occurs particularly for plural grammar rules i.e to test whether a given word is a plural form of not.

(3) Taking more time to translate from PC to English than from English to PC. This is due to not using the information in the representation of PC itself.

(4) Giving a different generated English sentence to the original sentence.

These problems give a serious look at the representation of the PC itself as well as some of the grammar rules.

The problem of the univ operator "=.." arises during the synthesis phase as the rule proceeds extremely timidly having to search the whole dictionary of verbs before finding the one that "fits" a term S where "*S=..[V,X,Y]*", if the condition which uses the univ operator is placed at the end of a rule, for example:

```
trans_verb(plural,X,Y,S)-->
        [Z],
        { (verb(V,Z),
           plural(Z),
           S=..[V,X,Y]) }.
```

An alternative is the following where the last line is moved up two lines:

```
trans_verb(plural,X,Y,S)-->
        [Z],
        { (S=..[V,X,Y],
           verb(V,Z),
           plural(Z)) }.
```

However, this works admirably on the synthesis phase but fails immediately and consistently during the analysis. A "natural" replacement for both rules would be:

```
trans_verb(plural,X,Y,V(X,Y))-->
        [Z],
        { (verb(V,Z),
           plural(Z)) }.
```

This structure is not allowed in current Prolog interpreter used and also in most Prolog implementation. Hinde and Mawdsley[1984] have proposed then a standard fix "f(V,X,Y)"is applied representing structure V(X,Y) such that the rule works in both direction and does not require nearly duplicate rules for analysis and synthesis phases. Although this new structure suffers rather badly in readability and would be much clearer if variable functors were allowed (Warren[1980], and Hinde[1984] and also [1986]), this structure is adopted to overcome the problem concerning with the univ operator.

There are many methods or strategies to overcome those other problems (problem 2, 3 and 4 above) bearing in mind that these grammar rules will be used in a question-answering system and it may also be used in an inter-lingual question-answering system e.g from English to PC and back to Malay language or vice versa in future. The following sections are three strategies which try to overcome those remaining three problems.

4.4.1 A Tracing technique

By using this technique, we add an extra variable or argument to the existing grammar rules. We called this variable a *tracing variable*. A tracing variable is used to identify a particular grammar rule. For example:

sentence(X,T)-->statement_phrase(X,T)

will be converted into Prolog clause as:

sentence(X,T,Y,Z):-statement_phrase(X,T,Y,Z).

The variables X and T which refer to the resulting PC and the tracing variable respectively, will be instantiated by the end of translating process of an English sentence, Y. So, by knowing the value of X and T, we can synthesize an English sentence and furthermore the result of this synthesis of an English sentence is actually Y itself, i.e the same as the original (input) English sentence. Let see another example of grammar rules with the tracing variables:

```
/* the grammar rules for proper nouns */
gproper_noun(X,N,[gpn3,who])-->
        [who].
gproper_noun(plural,N,[gpn2,X])-->
        [X],
        ( (plural(P,X),
        proper_noun(P,N),
        not(noun(P,Y))) ).
gproper_noun(singular,N,[gpn1,X])-->
        [X],
        ( (proper_noun(X,N),
        not(noun(X,X1),
        not(plural(X2,X),
        not(pronoun(D,X,P))) ).

/* the grammar rules for class of names */
class_name(singular,X,P,[cn1/Gn])-->
        [a],
        gnoun(singular,X,P,Gn).
class_name(singular,X,P,[cn2/Gn])-->
        [an],
        gnoun(singular,X,P,Gn).
class_name(plural,X,P,[cn3/Gn])-->
        gnoun(plural,X,P,Gn).
class_name(PL,X,P,[cn4/Gadj])-->
        gadjective(Pl,X,P,Gadj).
```

The above example of the set of grammar rules are intended to analyse proper nouns and a class of names. Each grammar

rule has a unique tracing variable. For instance, the tracing variables for the grammar rules to analyse a class of names are "cn1", "cn2", "cn3" and "cn4" which are to indicate a singular noun which precedes with "a", a singular noun which precedes with "an", a plural noun and an adjective respectively.

Another by-product of implementing the tracing technique is that we may able to classify the class of each word of the input sentence provided that the symbol used for the tracing variable is unique. For example:

```
?- revsent.      /* example [4.4.1.1] */
|: John loves Mary.
==========>>>
f(loves,John,Mary)

The tracing variable=

[s3,[np7,[gpn1,John]],[cvp0,[vp1,[tv7,loves]],[cnp0,[np7,[gpn1,Mary]]]]]

<<<==========
John loves Mary.

yes

?- revsent.      /* example [4.4.1.2] */
|: somebody visits the zoo.
==========>>>
exists(_1,indefinite(_1),f(person,_1)&exists(_2,definite(_2),f(zoo,_2)&f(visits,_1,_2))).

The tracing variable=

[s3,[np5,[qpr1,somebody]],[cvp0,[vp1,[tv7,visits],[cnp0,[np2,[gdet3],[gn2,[gn02,zoo]]]]]]]

<<<==========
somebody visits the zoo.

yes.
```

Session 4.4.1.1: Examples of using tracing variables

From the above session, we can analyse and synthesize an English sentence by using the tracing technique. In the example [4.4.1.1] above, the tracing variable reveals that "John" and "Mary" are proper nouns and "loves" is a transitive verb. Furthermore the grammar rule which is used

to analyse "John" and "Mary" is the rule "gpn1" of proper nouns and the rule "tv7" of transitive verbs is used to analyse the word "loves". This information will be used to synthesize an English sentence from the resulting PC. As we intend to use the grammar in a question-answering system, we demonstrate the following examples in a question-answering session:

```
?-listing(knowledge).   /* a listing of KB clauses */
knowledge(f(kind,John))
knowledge(f(kind,Peter))

?-phrase.    /* example [4.4.1.3] */
|: who are kind?

NEXT SENTENCE:
       who are kind?
==============>

NEXT QUESTION:
       f(kind,_1).

The tracing variable=

[s3,[np7,[gpn3,who]],[cvp0,[vp0,[tv2,are]],[ccn0,[cn4,gadj,kind]]]]

The translation of the negation:
[]:-f(kind,_1).

>>answer: Yes,
       f(kind,John)
&&&&&&&&&&&&&&&&&&&&&&

======>
Yes, it is true that who are kind.

top(phrase)|: more answers ? n

yes
```

Session 4.4.1.2: A question-answering example using the tracing technique

It can be seen from the above session that although there exists John who is kind, but the answered sentence is still the same with the questioned one. At least the answer should be "Yes, it is true that John is kind". It should be noted here that the word "who" is parsed as a proper noun. The inability of the grammar system to return the correct sentence is due to the rigidity of the tracing variable

itself. In other words, the tracing variable contains fixed words and their classes such that it cannot produce different sentences other than the original (input) sentence itself.

It should be noted here that we use top level predicate "*phrase*" in the above question-answering system and also in the rest of this section. As explained in the last section (section 4.3) about procedure SORQ and at the beginning of this section about the need to add one more variable in the grammar rules for tracing variables, then we need to redefine the procedure SORQ to accommodate these tracing variables and the new procedure is called SORQTR which can be found in the Appendix.

Usually, in a question-answering system, the word "who" will be replaced by other proper noun such as "John" in the above session (4.4.1.2). So, the word "who" will not be recorded in the tracing variable itself but instead an uninstantiated variable is recorded. This is parallel to the uninstantiated variable assigned to word "who" for the corresponding PC, for instance as shown in the above session (4.4.1.2) where variable "_1" of "*f(kind,_1)*" represents "who" of the question "who are kind?". Thus the grammar rule which deals the special proper noun "who" is changed from:

```
gproper_noun(X,W,[gpn3,who])-->
        [who].
```

to:

```
gproper_noun(X,W,[Z,Y])-->
        [who],
        { (var(W)) }.
```

The above rule does not instantiate the tracing variables "Z" and "W" with "gpn3" and "who" respectively. The variable

"W" will be instantiated with any proper noun during a question-answering session and thus the condition "*var(W)*" will prevent the use of this rule during retranslation of PC to English or synthesizing phase. While the variable "Z" is not instantiated such that it can be instantiated with other rules of proper nouns. For example:

```
?- revsent.      /* example [4.4.1.4] */
|: who are kind?
==========>>>
f(kind,_1)

The tracing variable=

[s3,[np7,[_2,_3]],[cvp0,[vp0,[tv2,are]],[ccn0,[cn4,gadj,kind]]]]

<<<==========
who are kind.

yes
```

Session 4.4.1.3

Clearly it can be seen from the above examples that no word "who" and the tracing variable which indicated the grammar rule for analysing the proper noun "who", are recorded or instantiated i.e variables "_3" and "_2" respectively. However if this modified rule is used in a question-answering system, then the second question "who are kind?" did not generate an acceptable answered sentence (see example [4.4.1.6] of Session 4.4.1.4 below) by assuming the same KB clauses exist in the database (see the listing of KB clauses in the Session 4.4.1.2 above).

```
?-phrase. /* example [4.4.1.5] */
|: who is kind?

NEXT SENTENCE:
       who is kind?
==============>

NEXT QUESTION:
       f(kind,_1).

The tracing variable (before)=

[s3,[np7,[_2,_3]],[cvp0,[vp0,[tv1,is]],[ccn0,[cn4,gadj,kind]]]]
```

```
The translation of the negation:
[]:-f(kind,_1).

>>answer: Yes,
        f(kind,John)
&&&&&&&&&&&&&&&&&&&&&&

The tracing variable (after)=

[s3,[np7,[gpn1,John]],[cvp0,[vp0,[tv1,is]],[ccn0,[cn4,gadj,kind]]]]

=======>
Yes, it is true that John is kind.

top(phrase): more answers ? n

yes

?-phrase.  /* example [4.4.1.6] */
!: who are kind?

NEXT SENTENCE:
        who are kind?
==============>

NEXT QUESTION:
        f(kind,_1).

The tracing variable (before)=

[s3,[np7,[_2,_3]],[cvp0,[vp0,[tv2,are]],[ccn0,[cn4,gadj,kind]]]]

The translation of the negation:
[]:-f(kind,_1).

>>answer: Yes,
        f(kind,John)
&&&&&&&&&&&&&&&&&&&&&&

The tracing variable (after)=

[s3,[np7,[gpn1,John]],[cvp0,[vp0,[tv2,are]],[ccn0,[cn4,gadj,kind]]]]

=======>
Yes, it is true that who are kind.

top(phrase): more answers ? n

yes
```

Session 4.4.1.4: A question-answering examples using the tracing technique

Example [4.4.1.5] above give the correct answered sentence "...John is kind" where variables "_2" and "_3" in the tracing variable (before proving is carried out) are instantiated with "gpn1" and "John" respectively (see the

tracing variable after proving is carried out) and furthermore that the answered sentence is a singular sentence which is equivalent to the type of the input (questioned) sentence.

On the other hand, the example [4.4.1.6] above gives the same answered sentence "...who are kind" as the input sentence. This is due to different type of the input and the output (answered) sentences, i.e plural and singular types respectively. As "John are kind" is not a correct sentence (due to "[tv2,are]" in the tracing variable) then the output sentence is the same with the input sentence, i.e the system uses the same set of grammar rules during analysing and synthesizing. In this case, during synthesizing, the tracing variable for the special proper noun "who",i.e "[_2,_3]" (see the tracing variable (before) in the example [4.4.1.6] of Session 4.4.1.4 above), will be instantiated to "[gpn1,John]".

In order to overcome this problem, the tracing variables of the grammar rules for the singular and plural transitive verbs of "be" form are made equal i.e:

```
trans_verb(singular,X,Y,f(V,X,Y),[tv1,Z])-->
                [Z],
                { (verb_be(V,Z) ).
trans_verb(plural,X,Y,f(V,X,Y),[tv1,Z])-->
                [Z],
                { (verb(V,Z),
                   plural(Z)) ).
```

It should be noted here that the two above rules are still distinct in the sense of the type of the sentence, i.e a singular and a plural one. As a result of these modifications, the question "who are kind" can be answered properly.

Another deficiency of this technique is that the tracing variable must be instantiated in order to retranslate from PC to an English sentence. Otherwise it will cause unbounded recursion or a syntax error particularly in the usage of the library predicate "*name/2*". In other words, the path taken or the set of grammar rules used in translating from PC to English (synthesizing phase) is predetermined during translating English to PC (analysing phase) and its value is given by the tracing variable.

The time taken to translate English to PC or vice versa, therefore, is about the same. In fact, the time taken to translate PC to English is always less or equal to the time taken to translate English to PC since the path from PC to English has been determined. Another advantage of using the tracing variable technique is that the resulting tracing variable itself explains the class of each word of the input sentence, for example, see session 4.4.1.4.

No error of using predicate "*name(X,Y)*" occurs as X is always instantiated before using the predicate "*name/2*". The value of X is extracted from either the input sentence (from English to PC) or the tracing variable (from PC to English).

The details of the grammar rules incorporated with the tracing variable can be found in the Appendix.

4.4.2 The wording technique

Mawdsley [1984] has proposed a temporary data base which was called FLOATING_VOCAB(X) to speed up the process of translation from English to French or vice versa. The wording technique is adapted from the very same idea in order to study its suitability in English-PC-English system especially in a question-answering system. In fact, it does not restrict into a mono-lingual question-answering system but it also effect a interlingual question-answering system. So, the wording technique is actually keeping a record of each word of the input sentence in the database. For example, if the input sentence is "Nazrul is kind", then the record of each word of the sentence is kept as follows:

```
word_used(Nazrul).
word_used(is).
word_used(kind).
```

No repeating words will be recorded. For example, the words of the sentence "every man loves every woman" will be recorded as follows:

```
word_used(every).
word_used(man).
word_used(loves).
word_used(woman).
```

This database "word_used" will be used during the retranslation of PC to an English sentence. We will call this database "word_used" as a wording database. Although it will also be used during the translation from English to PC but as an extra or a redundant condition since the grammar parses the input sentence by using the input sentence itself. The extra condition "*word_used(X)*" is added to the grammar rules which contains "[X]" where X is a word of the input sentence. In other words, all grammar rules which classify the class of each word of the input sentence with a few exception are added with the extra condition. The

following rules are examples of using the condition "*word_used(X)*" and also not using it:

```
/* having the condition "word_used" */
trans_verb(singular,Z,Y,f(W,Z,Y))-->
                [W]
                { (word_used(W),
                   verb_be(W,W)) }.
/* not having the condition "word_used" */
verb_phrase(Z,Y,X)-->
                trans_verb(Z,Y,U,f(is,Y,U)),
                cp_class_name(Z,Y,X).
```

A group of grammar rules of containing the condition "[X]" but excepted from placing the condition "*word_used(X)*" is the group of the grammar rules which "X" has been determined (or has been instantiated), for example, in the grammar rules which classify a determiner, e.g "a", "the" and "an".

The wording database is set up before the sentence is parsed and after the whole input sentence is read. Basically the technique is equivalent to the previous technique, i.e the tracing technique (see section 4.4.1) The only different that the wording technique does not keep a track of the path from English to PC. Instead, it uses the information available from the wording database as well as the resulting PC in order to retranslate PC to English again.

The wording technique gives at least the same generated sentence as the input one. It generates also sometimes the equivalent sentence with the input one. For examples:

```
?- revsent. /* example [4.4.2.1] */
|: who is kind?

The listing of "word_used":
        word_used(who)
        word_used(is)
        word_used(kind).

==========>
f(kind,_1)

<==========
who is kind.

?- revsent. /* example [4.4.2.2] */
|: every man who loves every woman likes every food.

The listing of "word_used":
        word_used(every)
        word_used(man)
        word_used(who)
        word_used(loves)
        word_used(woman)
        word_used(likes)
        word_used(food)

==========>
all(_1,indefinite(_1),f(man(_1)=>all(_2,indefinite(_2),f(food,_2)=>f(likes(_1,_2)) &
                all(_3,indefinite(_3),f(woman,_3)=>f(loves,_1,_3)))

<==========
every man who loves every woman likes every food.

<==========
no man who loves every woman likes no food.
```

Session 4.4.2.1: The examples of using a wording database

In the example [4.4.2.2] above, two equivalent sentences are generated from one input sentence. The second sentence was generated due to the non-existence of the condition of "word_used(no)" in the rule of analysing determiner "no", thus permitting the second sentence to be generated.

If these grammar rules are used in a question-answering system, the generated answer sentence is not correct one. It is still the same as the input sentence particularly with regarding the questions such as "who is kind?" or "who are kind?". For example:

```
?-listing(knowledge).   /* listing of KB clauses */
knowledge(f(kind,John))
knowledge(f(kind,Peter))

?-phrase.    /* example [4.4.2.3] */
I: who is kind?

NEXT SENTENCE:
       who is kind?
***************>

The listing of "word_used":
       word_used(who)
       word_used(is)
       word_used(kind).

NEXT QUESTION:
       f(kind,_1).

The translation of the negation:
[]:-f(kind,_1).

>>answer: Yes,
       f(kind,John)
&&&&&&&&&&&&&&&&&&&&&&

*******>
Yes, it is true that who is kind.

top(phrase): more answers ? ;

>>answer: Yes,
       f(kind,Peter)
&&&&&&&&&&&&&&&&&&&&&&

*******>
Yes, it is true that who is kind.

top(phrase): more answers ? n

yes
```

Session 4.4.2.2: A question-answering example using the wording database

The example [4.4.2.3] above (see Session 4.4.2.2) shows that the answer sentence is "Yes, it is true that who is kind" where the word "who" should be replaced with "John" or "Peter" in this case. This is due to the non-existence of "*word_used(John)*" or "*word_used(Peter)*" in the wording database.

So the problem "who" encountered here is about the same one as in the tracing technique. The solution of this problem is a bit tricky as the wording database is set up before the sentence is parsed.

The "*word_used(who)*" will be asserted into the database if an input sentence contains any word "who". In this case, we also need a special rule to deal with proper noun "who" as it can be a singular or a plural one. The special rule for the proper noun "who" is as follows:

```
gproper_noun(X,Y)-->
        [who]
        { (var(Y)) }.
```

The above rule cannot be used if Y is instantiated especially as a result of a question-answering session. It prevents the retranslation of sentence (question) "who is kind?" in a question-answering system. So we need to define a rule for the "*word_used(Y)*" if Y is a proper noun and the "*word_used(who)*" exists in the database. The definition of the rule is shown as follows:

```
word_used(Y):-
        nonvar(Y),              /* Y is instantiated */
        proper_noun(Y,Y),       /* Y is a proper noun */
        exists(word_used(who)). /* it exists in the wording database */
```

Program 4.4.2.1: A special rule for dealing proper noun "who"

The above special rule for the proper noun "who" will only be used during the retranslation of PC to English (i.e "Y" is instantiated) and provided that the "*word_used(who)*" exists in the wording database and the instantiated "Y" is a proper noun. This special rule is asserted into the database before the whole input sentence is parsed provided that "*word_used(who)*" exists.

As in the tracing technique, the transitive verbs "is" and "are" will cause another problem in a question-answering system, therefore, the condition "*word_used(X)*" in the transitive verb rule which deals with verbs "is" and "are" are omitted. As a result, two sentences are generated from one input sentence when it involves "is" or "are". For example:

```
?- revsent.  /* example [4.4.2.4] */
|: who is kind?

The listing of "word_used":
        word_used(who)
        word_used(is)
        word_used(kind)
        word_used(_1):-
                nonvar(_1),proper_noun(_1,_1),word_used(who).

===============>
f(kind,_2)

<===============
who is kind.

<===============
who are kind.
```

Session 4.4.2.3:

The same sentences will also be generated from the input sentence "who are kind?". The listing of "word_used" include the extra rule as described before when the word "who" is part of an input sentence. Unfortunately, these modified grammar rules when used in a question-answering system, still does not generate the correct answers, i.e the answered sentence is still the same as in the above Session 4.4.2.2.

Although the equivalent modifications have been made to solve the same problem which occurred in the tracing technique, the resultant grammar rules still do not generate a correct answer in this case. Here, it seems that the

grammar rules did not use the information available in the resultant PC, i.e for instance the information "Peter" in the resultant PC "*f(kind,Peter)*" during the retranslation of PC to English. This information is gathered in the tracing technique such that no such problems are encountered.

This means that the formulation of PC should be revised especially the formulation of PC for a proper noun. Another reason we should revise the PC formulation for a proper noun can be seen in the following session:

```
?- revsent.  /* example [4.4.2.5] */
: a man loves Mary.

The listing of "word_used":
        word_used(a)
        word_used(man)
        word_used(loves)
        word_used(Mary)

===============>
exists(_1,indefinite(_1),f(man,_1)&f(loves,_1,Mary))

<===============
Mary loves a man.

<===============
a man loves Mary.
```

Session 4.4.2.4:

It can be seen from the above session (4.4.2.4) that the sentence "a man loves Mary" retranslates into two sentences, i.e "Mary loves a man" and "a man loves Mary". It is clear that the first retranslation sentence is a wrong one although it is grammatically correct and it also used all the words in the wording database. In this case, the variable "_1" has been instantiated with "Mary" as there exists "*word_used(Mary)*" in the wording database.

All the deficiencies explained as above are caused by the unsuitable method of PC formulation of a proper noun. So we need to change the formulation for a proper noun. For example, the PC which corresponds to the sentence "Peter is kind":

"*f(kind,Peter)*"

is changed to:

"*exists(Peter,proper_noun(Peter),f(kind,Peter))*"

These changes affect only the noun phrase rules which consist the rule "gproper_noun" as one of the conditions. The information "*proper_noun(Peter)*" is included in the PC to distinguish between a normal existential quantifier. For example:

```
?- revsent. /* example [4.4.2.6] */
|: John loves Mary.

==============>
exists(John,proper_noun(John),exists(Mary,proper_noun(Mary),f(loves,John,Mary)))

<==============
John loves Mary.

yes
```

Session 4.4.2.5: A new representation of PC

So the new representation of PC is different if we compare it with the old one, i.e *f(loves,John,Mary)*. Now let us see how this modification features in a question-answering system as shown in the following session assuming that the same KB clauses as shown in the Session 4.4.2.3 exist in the database:

```
?-phrase.    /* example [4.4.2.7] */
|: who is kind?

NEXT SENTENCE:
       who is kind?
==============>
       .
       :       (the listing of "word_used" is omitted)

NEXT QUESTION:
       f(kind,_1).

The translation of the negation:
[]:-f(kind,_1).

>>answer: Yes,
       f(kind,John)
&&&&&&&&&&&&&&&&&&&&&

======>
Yes, it is true that John is kind.

top(phrase): more answers ? ;

>>answer: Yes,
       f(kind,Peter)
&&&&&&&&&&&&&&&&&&&&&

======>
Yes, it is true that Peter is kind.

top(phrase): more answers ? n

yes
```

Session 4.4.2.6: A question-answering example using the new representation of PC

From the above session, it can be seen that the generated (answered) sentence is a correct one which represents the answered PC.

As the instantiation of tracing variable, the wording database should be created before the retranslation of PC to English sentence, otherwise an unbounded recursion or a syntax error will occur. So we cannot use them to translate from given PC to an English sentence straight forwardly. We need to define a procedure which will extract any possible word from the input PC before we can continue the synthesizing process. Here, we will not define the procedure

to extract the word from an input PC as we are only interested in the process of English->PC->English translation.

The wording technique does not determine the path taken but it will guide the grammar rules to retranslate the PC into an English sentence which corresponds to the wording database. It does prevent the unbounded recursion or the illegal used of predicate "*name/2*". On the other hand, it does not prevent other legitimate or grammatical sentences being generated. For example, in the above session 4.4.2.3, the input sentence "who is kind" is retranslated into two sentences, i.e "who is kind" and "who are kind" where both of them are correct.

This technique used the information either from the input sentence (question) or the PC during analysing or synthesizing respectively. So the time taken to go either way is about the same.

The detailed grammar rules adopting the wording technique can be found in the Appendix.

4.4.3 The conditioning technique: " var(X)" or "nonvar(X)"

The main reason for a syntax error of using predicate "*name(X,Y)*" is the uninstantiation of the variable "X". In order to prevent this error, an extra condition "nonvar(X)" is placed before the condition "*name(X,Y)*". The goal "nonvar(X)" succeeds if "X" is currently an instantiated variable. The predicate "*name(X,Y)*" is usually used to find out whether the input word is a singular or a plural one. Precisely, the predicate "*name(X,Y)*" will be used to convert the input word into a singular present tense word if it is not already one, i.e if it is a plural, past tense or others. The following grammar rule is used to check whether the word "W" is a plural verb ending with "s" or not.

```
trans_verb(plural,Z,Y,f(X,Z,Y))-->
        [W],
        { (name(W,V),
           append(V,"s",T),
           name(S,T),
           verb(S,X),
           transitive(S)) }.
```

However, the above rule cannot be used to translate PC to English as the word W is still uninstantiated. So an extra condition "*nonvar(X)*" is placed before the predicate "*name(W,V)*" and another rule is to be written in order to translate PC to English of a transitive verb ending with "s", i.e the above rule will be rewritten as:

```
trans_verb(plural,Z,Y,f(X,Z,Y))-->  /* rule (a) */
        [W],
        { (nonvar(W),
           name(W,V),              /* condition (i) */
           append(V,"s",T),        /* condition (ii) */
           name(S,T),              /* condition (iii) */
           verb(S,X),transitive(S)) }.   /* conditions (iv) */
trans_verb(plural,Z,Y,f(X,Z,Y))-->  /* rule (b) */
        [W],
        { (var(W),
           verb(S,X),transitive(S),  /* conditions (iv) */
           name(S,T),              /* condition (iii) */
           append(V,"s",T),        /* condition (ii) */
           name(W,V)) }.           /* condition (i) */
```

The rule (a) above is used to check whether the instantiated plural transitive verb "W" is ending with "s" or not and build a root word "X" without "s". On the other hand, the rule (b) is used to build a plural transitive verb "W" ending with "s" as a part of a whole sentence from a root word "X". Furthermore the order of the conditions in rule (a) is a reciprocal to the same conditions in rule (b), i.e the order is (i),(ii),(iii),(iv) in the rule (a) and a vice versa in the rule (b).

If we modify all the grammar rules which consist the predicate "*name/2*" and write an extra grammar rule to each of modification as above, then generally, the grammar rules can be used to translate either English to PC or a vice versa, but with some precautions especially regarding the used of the proper noun rules. The following are examples of translation from English to English via PC.

[4.4.3.1] John loves Mary.

```
==============>
exists(John,proper_noun(John),exists(Mary,proper_noun(Mary),f(loves,John,Mary)))

<==============
John loves Mary.
```

[4.4.3.2] Peter who loves Mary likes Plaice.

```
==============>
exists(Peter,proper_noun(Peter),exists(Plaice,proper_noun(Plaice),f(likes,Peter,Plaice)) &
       exists(Mary,proper_noun(Mary),f(loves,Peter,Mary)))

<==============

      |_that_|
Peter | who  | loves Mary likes Plaice.
      |_which_|
```

[4.4.3.3] somebody visits the zoo.

```
==============>
exists(_1,indefinite(_1),f(person,_1)&exists(_2,definite(_2),f(zoo,_2)&f(visits,_1,_2)))

<==============

a person    _|
an person    |  visits the zoo.
one person   |
somebody     |
someone     _|
```

It should be noted here that the new representation of PC for proper nouns, as described in the last section 4.4.2, has already been adopted here (see example [4.4.3.1] above) as the same problem of not using the information in the PC formulation during synthesizing the sentence as in the wording technique has arisen and furthermore to cut the retranslation time from PC to an English sentence.

The example [4.4.3.1] produces one-to-one sentences, i.e one input sentence is retranslated into one output sentence. Since there are three relative clause words defined in the dictionary, i.e "that", "who" and "which" in this order, then three sentences are generated in example [4.4.3.2]. This retranslation can be classified as a one-to-many translation. Another example of one-to-many translation is example [4.4.3.3] above where five equivalent sentences have been generated. So the number of generated sentences depends on the number of a certain class of words used in the input sentence and the number of the same class of words defined in the dictionary (see example [4.4.3.2]) or the number of different sentences which produce the same PC (see example [4.4.3.3]).

The same rule applies to the input sentence "who is/are kind?". Beside the sentence "who is/are kind" is retranslated, the other sentences "<proper_noun> is kind" are also retranslated where "<proper_noun>" is any defined proper noun words in the dictionary database, for instance, "Mary/John is kind", where "Mary" and "John" are defined as proper noun. However, if it is used in a question-answering system, the system will only generate the correct sentence provided that there is an answer to the question as shown in the following session, i.e the word "who" is replaced with a suitable proper noun:

```
?-listing(knowledge).   /* listing of KB clauses */
knowledge(f(kind,John))
knowledge(f(kind,Peter))

?-phrase.    /* example [4.4.3.4] */
|: who is kind?

NEXT SENTENCE:
       who is kind?
==============>

NEXT QUESTION:
       exists(_1,proper_noun(_1),f(kind,_1)).

The translation of the negation:
[]:-f(kind,_1).

>>answer: Yes,
       exists(John,proper_noun(John),f(kind,John))
&&&&&&&&&&&&&&&&&&&&&&

======>
Yes, it is true that John is kind.

top(phrase): more answers ? ;

>>answer: Yes,
       exists(Peter,proper_noun(Peter),f(kind,Peter))
&&&&&&&&&&&&&&&&&&&&&&

======>
Yes, it is true that Peter is kind.

top(phrase): more answers ? n

yes
```

Session 4.4.3.11 A question-answering example

On the other hand, if there is no answer to the question, then the generated answer sentence is wrong, i.e "Yes, it is true that who is kind". Only one sentence will be generated during answering the question as the procedure STATE-PH1 allows only one sentence to be generated (see Program 4.3.6 of the previous section 4.3).

Although no problem of the usage of "*name/2*", but this technique produced a different type of problem, i.e one-to-many translation which sometimes may cause an irritating or ungrammatical sentence, for example "an person visits the zoo" as in example [4.4.3.3] above.

In the previous two techniques, i.e tracing and wording techniques, no such problem occurs. One of the reasons, is that the PC representation does not contain enough information in order to produce a good sentence or the same input sentence when it matters. So, in order to get the results as the other two techniques, we need to put more information into the PC representation and it may make the PC more unreadable as it contains the information about the words of the sentence. For example, in representing the determiners, we add another extra information in the PC, i.e "det(X)" where X is a determiner to replace "indefinite(X)" or "definite(X)".

[4.4.3.4]. a man loves every woman
==========>
exists(X,det(a),f(man,X)&all(Y,det(every),f(woman,Y)=>f(loves,X,Y)))

[4.4.3.5]. the man visits the zoo.
==========>
exists(X,det(the),f(man,X)&exists(Y,det(the),f(zoo,Y)&f(visits,X,Y)))

Both sentences will be produced exactly as the inputs due to the completeness of the information contains in their PC

representation. It can be seen that this technique resembles the other two previous techniques and the results will expect to be the same. However, we need to write the exact PC if we would like to synthesize a sentence from a given PC as the above method work well if we are working from English->PC->English. The full grammar can be found in the Appendix.

4.4.4 Comments on the analysing and synthesizing techniques

In the last three subsections, we have discussed the three techniques which may be used in analysing and synthesizing an English sentence. All three techniques work admirably in the system of English->PC->English and also in the question-answering system of English->PC->English. This can be extended to inter-lingual systems (as described in Mawsdley[1984], Hinde and Mawsdley[1984], Baker[1985], Kok[1986]) but the PC representation has to be seriously looked and modified where necessary to suit the question-answering system especially in converting PC into Horn clauses as the inter-lingual system contains all the information for the translations which may be redundant or unsuitable for the question-answering system.

We need to be careful if we would like to use three techniques from PC->English (a synthesizing phase). The tracing technique can not be used in such a way. The wording technique needs extra help in order to extract the words from the PC representation in order to make the synthesizing phase successful. The conditioning technique (section 4.4.3)

needs the PC input to be written just like the sentence which defeats the purpose of writing them in PC form.

We have also introduced a new structure of PC, i.e "*exists(X,proper_noun(X),...)*" and "*exists(X,det(X),...)*". In case of "*exists(X,proper_noun(X),P)*", nothing is done during the skolemization process as this proper noun existential quantifier does not really mean an existential quantifier which should be replaced by a Skolem function but it is only for purpose of synthesizing of English sentences.

In the last three sections which describe all the three techniques, we have concentrated in a translation of English sentences which contain the word "who" as a proper noun. However, the grammars will act appropriately in translating word "who" as a relative pronoun, i.e a relative pronoun "who" will be translated into "&"(conjunction) and not into "_1" (a variable) as a meaning of a proper noun.

The first four stages of transforming a PC of the forms "*exists(X,det(the),P)*" and "*exists(X,det(X),P)*" where P is not instantiated to "the" are exactly the same as the stages for a PC of the form "definite(the)" and "indefinite(X)" respectively.

We will revisit these three techniques again when we will discuss the usage of them in rectifying and suggestion process later (chapter 6).

4.5 Comments

We have already discussed how an English sentence is analysed into PC which then is passed into the Prolog-based theorem prover for converting them into Horn clauses and either asserting the Horn clauses as a fact or answering the question (Horn clauses). The resulting PC is then is passed back to the English grammar for synthesizing them back into an ordinary (answered) English sentence.

We have seen also the problems in transforming a PC into Horn clauses for knowledge and query clauses especially regarding the definite determiner "the". In this case, we take the immediately previous reference to the subject as the symbol for both knowledge and questioned Skolem functions. This is one of the ways of referring of the definite determiner "the".

The question-answering system described is able to answer the question depending on the input, i.e either an English sentence (natural language) or PC, such that we can bypass the natural language grammar if we would like to and even assert the fact straight into the database using Prolog's consult command. We also can add any other natural language grammar to make the system inter-lingual on condition that the PC representation is the same.

Finally, I must admit once more that my knowledge of English grammar is limited thus the grammar which is discussed in this chapter may be small and may also not be satisfactory from the linguistic point of view. However most of the discussion is around the techniques of analysing and synthesizing an English sentence which are about the same regardless the size of the grammar apart from the grammar complexility.

CHAPTER 5

A FAULT DETECTING ALGORITHM

5.1 Introduction

In proving a theorem or making an enquiry to the database, we may get a negative answer due to some faults. Bundy et al[1985] classify these faults as follows:

[1] Factual Faults:

A rule false i.e the rules constitute a program which calculates incorrect answers.

[2] Control faults:

The rules are true, but have undesirable control behaviour when run as a program, e.g they do not terminate.

As described in chapter 3, the proving algorithm has been integrated with the cycling checking. So the control faults will not occur. In other words, the proving will terminate in all conditions except in the case of occur-check condition. However the factual faults will still occur. We will only concentrate on the required rule which simply does not exist. This non-existent rule will cause an error of omission in the context of Bundy et al.[1985]. Errors of omission occur when a rule failed to fire, either because it was incorrectly constrained, or the required rule simply does not exist.

In this thesis, we will not discuss how to find the rules which are insufficiently or incorrectly constrained. The insufficiently and incorrectly constrained rules will cause errors of commission and omission respectively. Error of commission is committed because the program fires a rule which is false or incorrectly constrained. It should be

noted here what we mean by the rule is the knowledge base (KB) clause in the database (see Program 3.3.1 of section 3.3.1 of chapter 3).

Why can we not get the result required? As explained above, this is due to the non-existent KB clause (rule). The other reason is that we may ask the wrong question such that we cannot get the expected or required result. So we will divide into two main parts:

[1] Fault detection.

[2] Fault rectification .

In this chapter (section 5.4), we will be only describing the first part, i.e a fault detection algorithm or a *critic* (following Mitchell et al.[1981]). The fault rectification algorithm will be explained in the next chapter (chapter 6). In the following next two sections, (sections 5.2 and 5.3), we will give an overview of software reliability and program debugging respectively.

5.2 Software Reliability

Due to expansion in software systems, the problem of software reliability has arisen. Many mathematical models such as the J-M model (Jelinski and Moranda[1972]), the probabilistic model (Shooman [1972]), the execution-time theory model (Musa[1979]), Fault Removal model (Littlewood [1981]) etc (see Sallih [1986] for a review of the said models and others] have been developed to describe the behaviour of software package errors and then to get some measures from which the reliability of these software packages can be calculated.

What is software? software includes the whole series of non-electronic support to computers, in other words, software is the non-physical part of the system (Ogdin[1979]). Software consists of an application program (a program is a unique way of communication (Prather[1984])) whose role is to solve user's problems, and system programs which handle the problems of computer service.

If we think carefully about the process of writing a software system we find it is not as easy a job as it might look, especially when we know that programmers are limited by time and cost specified by their own managers. Under these circumstances programs tend to have errors which make them unable to do what they should do, and this has led to the need to ensure reliability in software systems.

What is software reliability? Software reliability is the ability of a computer program to operate successfully

without failure, provided that its environment and time parameters have been specified. If a program is used in a different environment, the reliability may be different for each environment. There is no absolute measure of failure. Software could be accepted but on certain conditions. Putting in measures of successful program we can define failure as a deviation from these measures.

Of course, failures are caused by bugs. Basili and Perricone [1984] define a bug as something detected within the executable code that caused the module in which it occurred to perform in-correctly. However, in software reliability studies, the number of faults is not their concern but the performance of the program they are after. Sometimes a program containing many bugs performs closer to our requirements than a program containing fewer bugs, and in this matter Littlewood [1979], says, "software reliability means operational reliability who cares how many bugs are in a program? We should concerned with their occurrence on its operations". However and in spite of what we have said there are some cases where we might wish to know that the software is completely bug free; such a case could be a nuclear power station safety system.

Software reliability is as important as hardware reliability as the increasing usage of computer systems especially in very critical fields such as air traffic control etc. Computer software is also very costly in terms of money and labour. Boehm[1976], Brooks[1975], Myers[1978], and Yourdan and Constantine[1979] indicate that testing and debugging alone represent approximately half the cost of new system

development. As error detection and error correction are now considered to be the major cost factors in software development, it is worth spending effort to make sure that the programs we are writing are going to work.

To obtain increased reliability one should spot the cause of failure and remove it. If it is in the program coding the code should be corrected, or if it is a logical error the logic should be corrected. In the extreme case where the design of the software does not accept certain inputs the software should be modified. An important factor in achieving increased reliability is testing. Testing should cover every bit of a program.

Although there is similarity between software system testing and the hardware life-testing models, there are also significant differences, because (see Rault[1979]), software faults have a design origin while most hardware faults have a physical origin. Also the objectives of the tests are different. In hardware testing the statistical emphasis is often on estimating the failure rate of an item. In software testing the main statistical emphasis is on estimating the number of errors remaining in the system. Another reason for not using hardware methods is, as Littlewood [1980], says "a hardware device is certain to fail eventually, whereas a program if perfect is certain to remain failure free".

To detect errors we need first to classify the errors a piece of software may contain. These errors happen for a variety of reasons (Basili and Perricone [1984]):

[1]. misuse of the programming language

[2]. error in the logic of program.

[3]. error in the computational theory

[4]. error in the use of the data structures

[5]. error when correcting other errors.

and many other reasons which may cause other kinds of errors. Errors differ not only according to their type, but also in the way they present themselves. For example, Basili and Perricone [1984] mentioned that errors occurring in modified modules are detected earlier and at a slightly higher rate than those in new modules. The reason for this is that the causes of error in modified modules are due to the misinterpretation of the functional specifications.

Correcting an error could be a source of generating more errors (i.e either we fail to correct the error, or correcting a bug produces another bug or bugs). It has long been known that the debugging process is one of the sources of uncertainty, in software development, since correcting an error does not mean that our program has been freed from that error (an imperfect debugging). There is always a probability that this error will remain in the program, or even worse cause other errors to rise to the surface. There is another problem , and this is what Downs [1985], called the "obscure failures" problem. This means that a failure happens and because of a lack of information, no effort is spent to identify the error.

A lot of software failure models assume that all errors are detectable at all stages of testing; eventually this leads to the assumption of monotonically increasing software

reliability, characterised by convex upwards plot of cumulative number of errors versus time (see Fig 5.2.1 below).

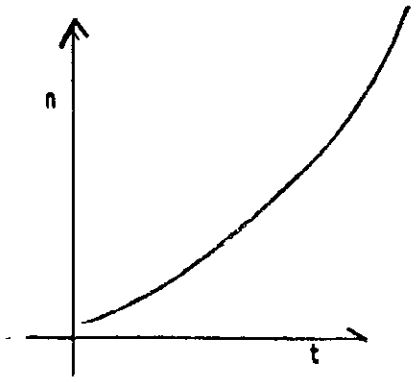


Fig. 5.2.1

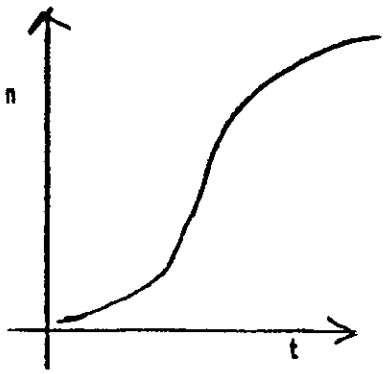


Fig. 5.2.2

In practice, however, many real software systems show a non-monotonic reliability profile (Angus et al. [1983]). This is shown by an "s-shaped" curve of cumulative number of errors versus time (see Fig 5.2.2 above). Studying this phenomenon reveals that in early debugging the bugs may take time to manifest themselves; once these errors have been detected other bugs may then become apparent. Various reasons for this behaviour come to mind- for example, some bugs may take an appreciable time to fix and during this period, no further bugs are detectable. Furthermore, a hierarchy of bugs may exists in which it is not possible to detect lower level bugs until the higher level ones have been fixed.

Many explanations could be given to the s-shaped curve produced by plotting errors detected against time. Some of these are (Schagen [1985] and Sallih [1986]):

(1) If the debugging process is not constant, i.e there is a change in the environment such as less effort being spent at the beginning of the debugging process, thus fewer errors being detected.

(2) If the correction of errors carried out during the debugging process generates further errors, then imperfect debugging could lead to a cluster of errors.

(3) The structure of a piece of software may consist of several modules and some of these modules which contain many bugs will only be exercised in a later stage of the debugging process.

(4) Software may contain two kinds of bugs: the first being major bugs which take some time to be fixed, during which time the system is down and no other errors can be detected. The second are minor bugs which take an insignificant time to be fixed.

(5) We may postulate a hierarchy of errors such that "secondary" bugs cannot be detected until all the "primary" bugs have been detected and removed. In other words, the debugging process does not "see" all possible errors from the beginning, and thus error detection rate is not monotonically decreasing.

Therefore the idea that the rate of finding errors is a monotonically decreasing function of the number of errors found should be abandoned as Angus et al.[1983] have failed to fit several different software failure models to a range of data that they had available relating to US defence software. Sallih [1986] has studied a few software reliability models which fit the s-shaped curve based on some of the reasons described above.

5.3 Program Debugging

In the last section, we have discussed the software reliability where many researchers have proposed different software reliability models in order to study the behaviour of the software failure profiles based on the assumption of how the debugging process is carried out. In this section, we will discuss the need of debugging, debugging processes and some debugging algorithms. The area of computer program debugging is also one of the key phases in the system software cycle. Debugging is the process of locating and correcting the error within the program once the existence of an error has been establishing by testing (Myers [1978]).

It is evident that a computer can neither construct nor debug a program without being told, in one way or another, what problem the program is supposed to solve, and some constraints on how to solve it. No matter what language of whatever generation we use to convey this information, we are bound to make mistakes. It is not because we are sloppy and undisciplined, as advocates of some program development methodologies may say, but because of a much more fundamental reason: as pointed out by Shapiro[1982] that we cannot know, at any finite point in time, all the consequences of our current assumptions.

A program is a collection of assumption, which can be arbitrarily complex; its behaviour is a consequence of these assumptions: therefore we cannot, in general anticipate all the possible behaviour of a given program (Shapiro[1982]). This principle manifests itself in the numerous undecidability results, that cover most interesting aspects

of program behaviour for any nontrivial programming system (Rogers[1967]). It follows from this argument that the problem of program debugging is present in any programming or specification language used to communicate with a computer.

As pointed out in the last section that debugging process accounted half of the cost of new system development. So if we can eliminate the debugging process, then we will save a lot of time and energy. It has also been suggested that one way to eliminate the need for debugging is to provide a correctness proof of the program. But, Goodenough and Gehart[1975] found seven bugs in a simple text formatter program described and informally proved by Naur [1969] who (together with Randell- Naur and Randell [1969]) suggested that we can dispense with testing altogether when we have given the proof of correctness of the program. So the informal or formal proofs of program correctness do not guarantee that the program is correct.

However, as pointed out by Goodenough and Gehart [1975], that the practise of proving a program correctness is useful for improving reliability, but suffers from the same types of errors as programming and testing, namely, failure to find and validate all special cases relevant to a design, its specification, the program and its proof. Gries [1981] also agreed that even though we can become more proficient in programming, we will still make errors, even if only of a syntactic nature. Hence some testing will always be necessary. But, he disagrees to call the testing process as a debugging, and suggests that the test is to increase our

confidence in a program we are quite sure is correct; finding an error should be the exception rather than the rule.

Manna and Waldinger [1978] also suggest that one can never be sure that specifications are correct, and agree that is is unlikely that program verification systems will ever completely eliminate the need for debugging. Debugging is an unavoidable component in the process of "model verification", in which the system verifies that it has the right idea of what the target program is (Balzer [1972]).

Traditionally, the efforts in program debugging were focused on how to bridge the gap between the programmer and the executable program. Core dumps and print statements were the common means of communication between the running program and the programmer. However, these are insufficient to solve the problems of software development. Another approach is the adaptation of structured programming in program development. This helps avoiding or detecting early many syntactic or shallow semantic errors.

The area of debugging crucial to software development and maintenance is semantic debugging. Syntactic errors are defined for the purposes of computer programming as errors that compilers recognise, and the use of high level programming with a strong-typing mechanism, such as Pascal, Algol-like languages will help toward finding syntactic errors. Semantic errors are those that compiler cannot recognise and the adaptation of structured programming techniques will help a little bit in removing such errors.

In formal sense, debugging can be understood from either a process viewpoint or a functional viewpoint (Vessey[1986]). The following figure (Fig 5.3.1) shows the basic model of debugging process (Vessey[1986]). This representation which she derived from medical diagnosis shows the different types of debugging behaviour as well as the relationships among them.

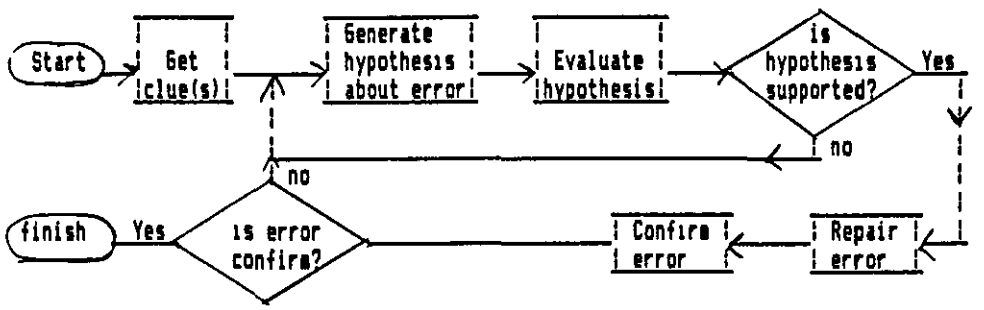


Fig. 5.3.1: Model of debugging process (Vessey[1986])

Fig. 5.3.2 below shows the understanding of debugging from a functional viewpoint i.e the model of debugging functions (Vessey [1986]) using structure chart conventions (Yourdan and Constantine[1979]; Weinberg[1979]).

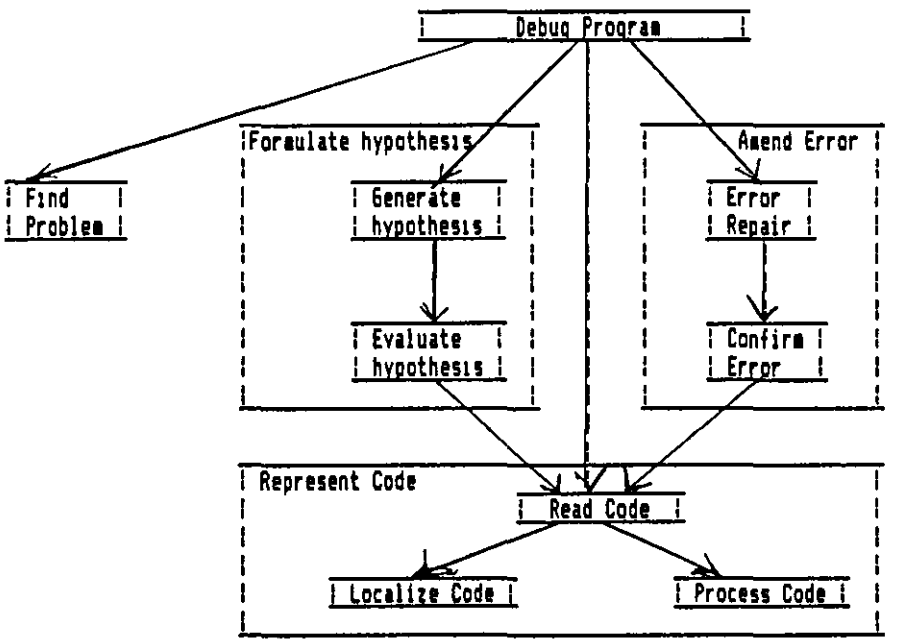


Fig. 5.3.2: Model of debugging functions (Vessey[1986])

The above Fig 5.3.2 shows top-down hierarchical relationship of functions. The functions at the top of the structure are responsible for control of the lower functions, which achieve most of the work, and invoke them as they required. The above chart defines the elements of procedural knowledge for debugging. Vessey[1986] gives a list of literatures which supports the debugging functions as shown above and which will be briefly described below,

DEBUG PROGRAM is the topmost function which exercises ultimate control over the debugging process. In FIND PROBLEM function (or get clue(s)), the search for clues is carried out to reveal the problem with the program. The FORMULATE HYPOTHESIS function is subdivided into GENERATE HYPOTHESIS (i.e to search for a possible cause of the problem) which invokes EVALUATE HYPOTHESIS (i.e to assess the validity of the suggested cause of the problem).

AMEND ERROR function which is invoked by DEBUG PROGRAM function, is further subdivided into REPAIR ERROR (i.e modification of the program in accordance with the perceived cause of the error), which invokes CONFIRM ERROR (i.e justification for introducing the suggested modification).

Another function invoked by the topmost function is REPRESENT CODE function. This function consists of the elementary functions: READ CODE (i.e sequential examination of program statements), LOCALIZE CODE (i.e search for a particular piece of code) and PROCESS CODE (i.e mentally process data through the program). Vessey [1986] pointed out that the inclusion of a REPRESENT CODE is essential that

when programmers are debugging programs with which they are not familiar.

Adam and Laurent [1980] discussed a debugging system called LAURA which have been designed not to prove the correctness of a program but to detect or localize the errors it may contain. LAURA uses a procedural description of the program task, under the form of a program model. Debugging is then viewed as a comparison of two graphs, built from the student program and from the program model (which is supposed to be a correct implementation of the algorithm). The system debugs program in various fields (e.g tax computation, perfect numbers, integration etc) and let the user himself make some difficult interpretations, i.e LAURA will localize a semantic error but it left to the user to solve it.

Shapiro[1982] also tried to lay theoretical foundations for program debugging, with the goal of partly mechanising this activity. In particular, Shapiro attempted to formalise and develop algorithmic solutions to the following two questions:

(1) How do we identify a bug in a program that behaves incorrectly?

(2) How do we fix a bug, once one is identified?

These questions (1) and (2) can be referred in term of Vessey's functions as FORMULATE hypothesis and AMEND ERROR functions respectively. However, Shapiro classified algorithms to solve the first problem as a diagnosis algorithm and the one that solves the second is a bug-correction algorithm.

To debug an incorrect program one needs to know the expected behaviour of the target program. Therefore Shapiro assumes the existence of an agent, typically the programmer , who knows the target program and may answer queries concerning its behaviour. The algorithms Shapiro developed are interactive, as they rely on the availability of answers to such queries. He integrated both diagnosis and bug-correction algorithms into a debugging algorithm. A debugging algorithm accepts as input a program (or empty one) to be debugged and a list of input/output samples, which partly define the behaviour of the target program.

Summarily, Shapiro's system can debug a PROLOG program using what is known as a *ground oracle*. The ground oracle is asked specific facts about the universe of discourse and must return true or false answers by the programmer. The oracle is not required to answer any universally quantified questions nor any containing free variables. In particular Shapiro's program can debug the empty program, or in other words synthesize a correct program from scratch using advice from the oracle. If a faulty (wrong) rule is fired and derived a contradiction, then the contradiction is backtracked with the oracle consulted at each stage to determine which branch of the tree the faulty rule lies. The rule may then be fixed or rectified by adding conditions to prevent the rule firing again in the erroneous context.

Rule learning techniques can also be considered as debugging program techniques. The task tackled by rule learning techniques is to modify a set of rules of the form hypothesis implies conclusion. This set of rules can be

considered as a program especially written in Prolog clauses. The basic rule learning technique is as follows:

Until the rules are satisfactory:

1. Identify a fault with a rule

2. Modify the rule to remove the fault.

Step 1 and 2 above can be considered in Vessey's terms as FORMULATE HYPOTHESIS and AMEND ERROR functions. Most identifying faults techniques is by comparing the ideal trace (graph) with a learning (program) trace (graph) (for eg. Bradzil[1981]). Shapiro's technique as briefly described above is also one of the technique to identify a fault.

A lot of techniques are used to modify the faulty rules such as reordering them (eg. Bradzil[1981]), adding extra condition(s) to them (eg. Bradzil [1981], Waterman [1970]), instantiating them (eg. Bradzil [1981], Shapiro [1982]), updating them (eg Waterman[1970], Mitchell et al.[1981] and [1983]) or asking a ground oracle to the user (Shapiro[1982]). Bundy et al.[1985] give an excellent review and comparison of rule learning techniques.

5.4 Faults Detection Algorithm

As we know Prolog implements backtracking in order to find other possible answers or paths. So we can append another rule (procedure FACTPR.3) to the procedure FACTPR (see Program 3.3.3.11 of chapter 3) in order to capture the failed facts or goals during the proving process.

```
/* PROCEDURE FACTPR.3: to capture the failed facts or goals */
factprolog(Q,Usedclauses,Hp,Hpl,Goalclause):-
        /* procedure FACTPR.3 */
        not(reason(Q)),
        assertz(reason(Q)),
        fail.
```

Program 5.4.1: An appended rule of procedure FACTPR

Program 5.4.1 above shows the definition of procedure FACTPR.3. In other words, if all KB clauses have been unsuccessfully tried, then assert the failed goal or fact into the database provided that the same failed goal or fact has not been asserted before. Then the rule is set to fail in order to make Prolog backtrack again to find other possible answers or failed goals. The failed goals or facts are asserted in the database with predicate *reason/1*. Accordingly we will call the failed goals or facts the reasons. However the above procedure FACTPR.3 has some defects. For example, suppose we have the following KB clauses in the database:

```
knowledge(f(a,X,Y):-f(b,X),f(c,Y)).
knowledge(f(c,y)).
```

And suppose we would like to prove f(a,x,y) by using the question-answering system described in Chapter 3 where procedure FACTPR which includes an appended procedure FACTPR.3 (see Program 5.4.1). The proving is shown in the following session (Session 5.4.1):

```
?- pcquest.
|: f(a,x,y).

NEXT QUESTION:
        f(a, x, y)

The translation of its negation:
[]:-f(a, x, y).
%%%%%%%%%%%%%%%%%%%%%%%%%

>>answer: No, it cannot prove :
        " f(a, x, y) "
*****************************************

        do you like to assert
                " f(a, x, y) "
        as a fact in the database (y/n)? n

yes

?- listing(reason).
reason(f(b, x)).
reason(f(c, y)).
reason(f(a, x, y)).

yes
```

Session 5.4.1: Proving "f(a,x,y)".

It clearly can be seen from the above session 5.4.1 that "f(a,x,y)" is undeducible as there are not enough knowledge in the database. However the listing of predicate *reason/1* produces an unacceptable list. From the following Fig. 5.4.1, the only true reason clause is predicate *reason(f(b,x))*, while the others two, i.e predicates *reason(f(c,y))* and *reason(f(a,x,y))*, are untrue reason clauses.

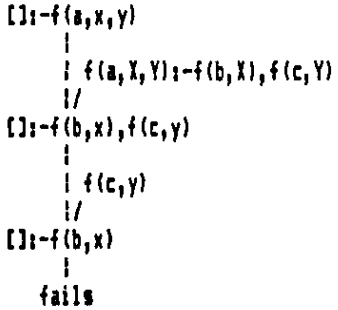


Fig. 5.4.1: Proving tree of f(a,x,y).

The untrue reason clauses are generated due to Prolog backtracking. What we mean by the true reason clause is the failed goal or fact which really does not exist in the database either in the form of a ruled KB clause with the failed goal as its head or a factual KB clause. The untrue reason clause is a reason clause which is not a true one. So we need to define predicate *knowledge_base_head/1* in order to prevent the generating of untrue reason clauses as follows:

```
knowledge_base_head(Head):-
                knowledge_base(Head).
knowledge_base_head(Head):-
                knowledge_base((Head:-Body)).
```

Program 5.4.2: The definition of predicate knowledge_base_head/1

And by putting this extra condition to procedure FACTPR.3, we will be able to produce true reason clauses only, i.e from the above example (Session 5.4.1), only predicate *reason(f(b,x))*. So, the new definition of procedure FACTPR.3 is as follows:

```
/* PROCEDURE FACTPR.3: to capture the failed facts or goals */
factprolog(Q,Usedclauses,Hp,Hp1,Goalclause):-
        /* procedure FACTPR.3 */
        not(reason(Q)),
        not(knowledge_base_head(Q)),
        assertz(reason(Q)),
        fail.
```

Program 5.4.3: The new definition of procedure FACTPR.3

The non-existent and undefined predicates are two different kind. The above procedure (Program 5.4.3) will detect the non-existent predicates. On the other hand, POPLOG Prolog will detect an undefined predicate. It is clear what is meant by an undefined predicate. However, the non-existent predicate is a predicate which does not exist in the database but it is not necessarily an undefined one. In other words, a non-existent predicate is more general than

an undefined predicate. The following session (5.4.2) shows an example of a non-existent predicate.

```
?- foo(a,b).
no

?- listing(foo).
foo(a,c).
foo(a,d).
foo(a,e):-foo(a,b).

yes
```

Session 5.4.2: An example of the non-existent predicate

From the above session 5.4.2, predicate *foo(a,b)* does not exist in the database although predicate *foo/2* has been defined. In other word, predicate *foo(a,b)* is a non-existent one. This cannot be detected by POPLOG Prolog.

As mentioned above, POPLOG Prolog has been integrated with a method to handle the undefined predicate when it is used as a goal. By default, when a goal is attempted for which there is no predicate, POPLOG Prolog will create one:

```
/* assume no prior definition of "pred/2" */
?- pred(1,2).
no

?- listing(pred).
pred(_1,_2):-
        fail,
        'UNDEFINED-PREDICATE'.
```

Session 5.4.3: the handling of an undefined predicate by POPLOG Prolog

The above session 5.4.3 shows an example how POPLOG Prolog handles an undefined predicate, i.e predicate *pred/2*. It should be noted here that the early development of procedure FACTPR.3 is done by using the Edinburgh Prolog version NU7 (see Clocksin &Mellish [1980a]) which does not have this facility (detecting undefined predicates). Even if the Edinburgh Prolog does have this facility, it still can detect non-existent predicates.

In order to see how the procedure works, suppose we have the following KB clauses in the database as shown in List 5.4.1.

```
knowledge((a(r):-b(r),c(r))).
knowledge((a(s):-b(s),c(s))).
knowledge((b(s):-k(s),j(s),h(s),d(X,s))).
knowledge(c(s)).
knowledge((d(X,s):-e(X,s))).
knowledge((e(s,s):-f(s,s))).
knowledge((e(s,s):-g(s,s))).
knowledge(e(r,s)).
knowledge((h(s):-i(s))).
knowledge(i(s)).
knowledge((k(s):-m(s),l(s))).
knowledge(m(s)).
```

List 5.4.1: a listing of KB clauses

And suppose we would like to deduce or prove "a(X)" from the above database (List 5.4.1). The following figure (Fig. 5.4.2) gives a full proving tree of "a(X)".

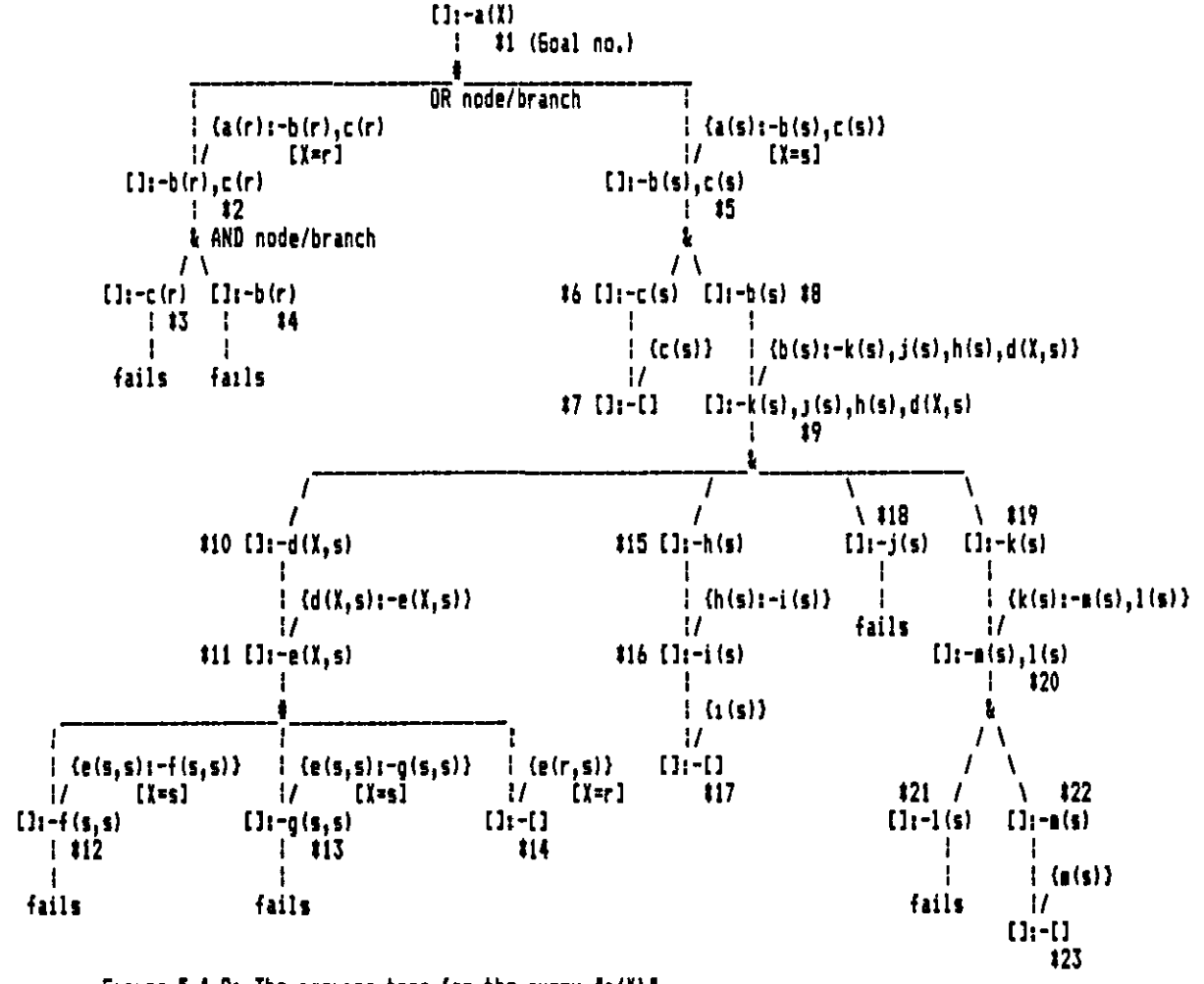


Figure 5.4.2: The proving tree for the query "a(X)"

As it can be seen from the above figure (Fig. 5.4.2) that there are two possible solutions to the query "a(X)", i.e "a(r)" (X=r) or "a(s)" (X=s). However both solutions cannot be derived from the database (List 5.4.1) because the database lacks the information or the knowledge. The following list (List 5.4.2) shows all reasons or non-existent clauses which can cause the non-deducibility of "a(X)":

```
(r1). reason(c(r))
(r2). reason(b(r))
(r3). reason(f(s,s))
(r4). reason(g(s,s))
(r5). reason(j(s))
(r6). reason(l(s))
```

List 5.4.2: The listing of non-existent clauses

The query "a(r)" cannot be deduced because both *knowledge(c(r))* and *knowledge(b(r))* do not exist in the database. On the other hand, the query "a(s)" cannot be deduced because of the non-existent of predicates *knowledge(j(s))*, *knowledge(l(s))* and either *knowledge(f(s,s))* or *knowledge(g(s,s))*. In other words, the sets of reason clauses for deducing "a(s)" are [(r3),(r5),(r6)] and [(r4),(r5),(r6)]. And set [(r1),(r2)] is the set of reason clauses for deducing "a(r)".

Now, let use again the question-answering system based on procedure FACTPR which consists a new procedure FACTPR.3 (Program 5.4.3) to deduce "a(X)" from the database as in List 5.4.1 above. Clearly the system cannot deduce or prove "a(X)". However the system has asserted predicate *reason/1* in the database to show the failed goals or facts. The following session (Session 5.4.4) shows the listing of predicate *reason/1*.

```
?- listing(reason).
reason(c(r)).
reason(f(s,s)).
reason(g(s,s)).
reason(j(s)).
yes
```

Session 5.4.4: The listing of predicate *reason/1*

However the system only asserted four reason clauses as shown in the above session 5.4.4 which correspond to (r1), (r3), (r4) and (r5) respectively as listed in List 5.4.3. This is a consequence of the way Prolog backtracking is implemented. Prolog does not try to prove the remaining goals, if they exist, when the current goal fails but instead Prolog will go back (backtrack) to the last successful goal and find its alternative. The following figure (Fig 5.4.3) shows the question-answering system's proving tree for deducing "a(X)" by using the same goal number as shown in Fig. 5.4.2 above.

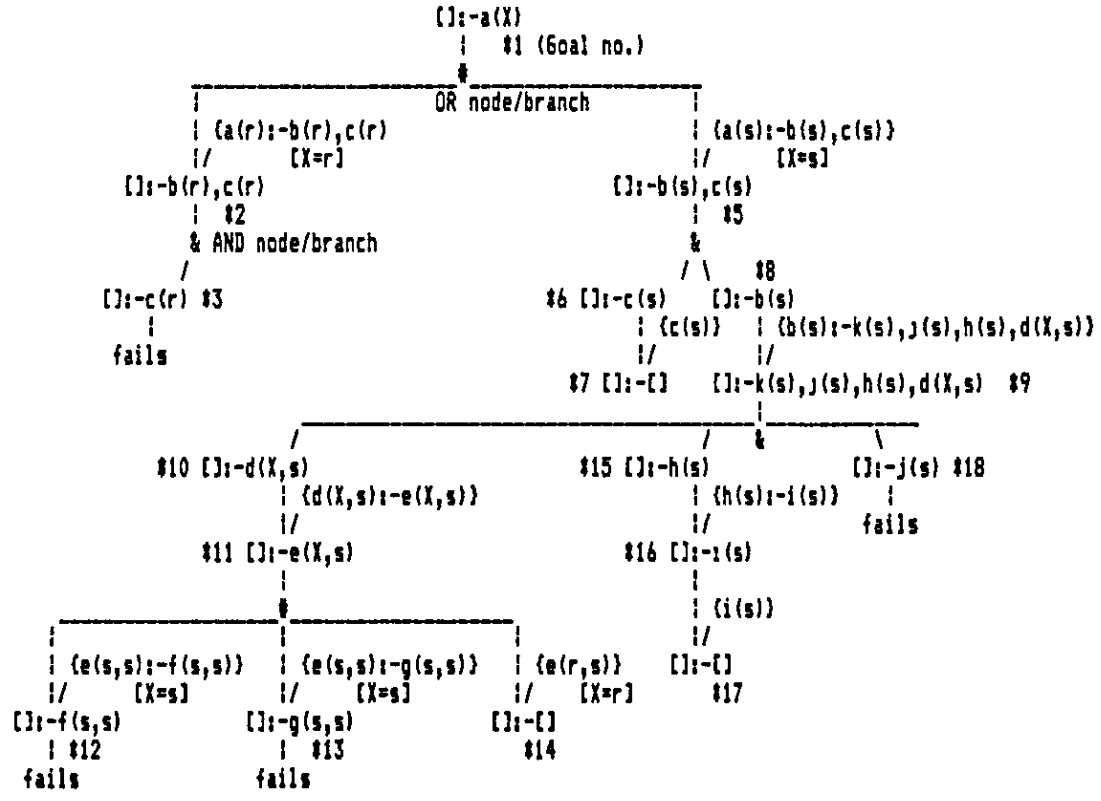


Figure 5.4.3: The system's proving tree for the query "a(X)"

It can be seen from the above figure 5.4.3, the system will not prove the clauses "b(r)" of the goal (*2) and "k(s)" of the goal (*9) as the goals "c(r)" (goal no. *3) and "j(s)" (goal no. *18) fail respectively. Consequently the system will not be able to assert clause "b(r)" and "l(s)" as reasons. Furthermore the listing of predicate *reason/1* as shown in Session 5.4.4 does not show the relationship among them. In other words, the listing does not show the membership relationship of the set of reason clauses among them, for instance *reason(f(s,s))* and *reason(j(s))* are members from the same set of reason clauses.

By saving the details of the rectifying process of the reason clauses for the next chapter, and by assuming that the reason clauses do not exist in the database (i.e it lacks knowledge), let see the following example to prove the goal "a(X)" as shown in the following session 5.4.5:

```
?- pcquest.
|: a(_1).

NEXT QUESTION:
        a(_1)

The translation of its negation:
[]:-a(_1).
&&&&&&&&&&&&&&&&&&&&&&&&

>>answer: No, it cannot prove :
        " a(_1) "
========================================

The reason why the goal:
        "a(r)"
fails is due to the non-existence of the the following fact:

        c(r)

However, we may able to prove the goal
after doing some corrections or additions

    --Do you like to continue ?y
```

```
If the following is true:
       c(r)

Do you like to try again by using the above assumption
   ---? y

RE-QUESTION:
       a(r)

The translation of its negation:
[]:-a(r).
&&&&&&&&&&&&&&&&&&&&&&&&

))answer: No, it cannot prove :
       " a(r) "
==========================================

The reason why the goal:
       "a(r)"
fails is due to the non-existence of the following knowledge:

       b(r)

However, we may able to prove the goal .
after doing some corrections or additions

    --Do you like to continue ?y

If the following is true:
       b(r)

Do you like to try again by using the above assumption
   ---? y

RE-QUESTION:
       a(r)

The translation of its negation:
[]:-a(r).
&&&&&&&&&&&&&&&&&&&&&&&&

))answer: Yes,
       a(r)
==================================

PROVED: a(r) ?

yes
```

Session 5.4.5: Proving "a(X)".

It can be seen from the above Session 5.4.5. that in order to prove "a(r)", the system successfully proves on the third attempt as in this case there are two reason clauses, i.e clauses "c(r)" and "b(r)". Furthermore the reason clause "c(r)" does not emerge on the first attempt but it only

emerges on the second attempt. This is the main disadvantage of the method of capturing reason clauses which does not show the relationships among them. So the number of attempts before the question is successfully proved, is more than the actual number of the reason clauses for a particular question provided that no repeating reason clauses occurred and in the case of non-existent clauses.

The reason why both reason clauses "c(r)" and "b(r)" could not be detected at the same time is that they are at a different hierarchy or level. This is one of the explanations of s-shaped curves discussed in the previous section 5.2. In this case, as "c(r)" and "b(r)" can be considered as a primary and secondary bugs (faults) respectively, so "b(r)" cannot be detected until all primary faults (bugs) have been detected and rectified.

In order to overcome the problem of the relationship of the reason clauses or the different hierarchy (level) of reason clauses, we now define a new predicate *set_of_reason/3* which will provide the information of the relationship of the reason clauses as well as their associated query goals. However the most important problem here is to link their relationship among the failed goals or reasons.

As shown in the Program 5.4.3 above, the last procedure of FACTPR, i.e procedure FACTPR.3, returns the failure value, in order to prevent the failed goals becoming successful ones and also to make Prolog backtrack. What we need here is a method of recording the failed goals without setting procedure FACTPR.3 fails, such that the question-answering

system can continue to prove the remaining goals until all of them have been "successfully" proved. In other words, the procedure FACTPR succeeds even though the answer may not be a correct one.

This can be achieved by creating another variable to record the failed goals and then testing its existence at the end of proving. For instance, if we refer back to Fig. 5.4.2, all the failed goals, i.e goals (*2), (*3), (*9), (*13), (*14), and (*17), will be set to succeed. All failed goals will also be recorded with their associated node numbers, for instance, the failed goal "c(r)" is recorded as the pair [(c(r),2].

Thus procedures FACTPR, BASEPR (see procedure 3.3.3.13 of chapter 3) and FACTCL (see Program 3.3.3.15 of chapter 3) need to be modified as well as procedure ASK especially procedure ASK.1 (see Program 3.3.3.9 of chapter 3). The following procedure (Program 5.4.4) is a final version of procedure FACTPR with a new predicate factprolog/7. The number of arguments of the predicate has been increased from five to seven in order to record the failed goals, i.e variable "Fc".

```
/* a new version of procedure FACTPR */
factprolog((Q1,Q2),Usedclauses,Hp,Hp1,Goalclause,Fc,Fc1):-
        /* procedure FACTPR.1 */
        !,
        factprolog(Q2,Usedclauses,Hp,Hp2,[Q1/Goalclause],Fc,Fc2),
        factprolog(Q1,Usedclauses,Hp2,Hp1,Goalclause,Fc2,Fc1).
factprolog(Q,Usedclauses,Hp,Hp1,Goalclause,Fc,Fc1):-
        /* procedure FACTPR.2 */
        assertagoal1([Q/Goalclause],N),                    /* procedure ASSGL1 */
        baseprolog(Q,Usedclauses,Hp,Hp1,Goalclause,N,Fc,Fc1).   /* procedure BASEPR */
factprolog(Q,Usedclauses,Hp,Hp1,Goalclause,Fc,[[Q,N]/Fc]):-
        /* procedure FACTPR.3: to capture the failed facts or goals */
        not(reason(Q)),
        not(knowledge_base_head(Q)).
```

Program 5.4.4: A final version of Procedure FACTPR

As shown in the procedure FACTPR.3 above (see Program 5.4.4), the variable "FC" is a list of pairs of the failed goal Q and its associated goal or node number. This variable "Fc" will be used to test the existence of failed goals. The Prolog will automatically record the set of the pairs of reasons and its associated number for each query goal when a backtracking occurs. As procedure FACTPR is modified, accordingly procedures BASEPR and FACTCL have to be modified by adding two variables as their arguments for recording the set of reason clauses exactly the same way as procedure FACTPR. The final version of both procedures can be seen in the Appendix.

The same modifications have also to be made in procedure ASK.1 which calls procedure FACTPR. Accordingly a new condition to check the existence of the failed goals is to be added into procedure ASK.1 as well. The new additional condition is a procedure REASONING. So the new version of procedure ASK is as shown in Program 5.4.5 below. The other procedures of ASK (procedures ASK.2 and ASK.3) remain unchanged.

```
/* Procedure ASK : to prove each clause of the question */
asking([Quest1/Quest]):-
        /* procedure ASK.1 */
        top_asking(Quest1,Goal),                    /* procedure TOPASK */
        factprolog(Goal,[],[],Hp,[],[],fc),         /* a new procedure FACTPR */
        reason_testing(Goal,Fc),                    /* procedure REASONING : a new condition */
        successful_action(Hp).                      /* procedure SUCCESS */
asking([Quest1/Quest]):-
        /* procedure ASK.2 */
        print_comment(Quest1),  /* procedure PRCMT */
        asking(Quest).          /* Prove a next query clause */
asking([]):-
        /* procedure ASK.3 */
        assertz(toptry([])),
        fail.
```

Program 5.4.5: a new version of procedure ASK

At the end of proving or procedure FACTPR has been successfully executed, procedure REASONING will check whether the answer given by procedure FACTPR is a successful answer or a failed one. This is achieved by testing the value of the list (variable) "Fc" which was returned by procedure FACTPR (see the arguments of a new procedure FACTPR in Program 5.4.5). If the list "Fc" is an empty one, then the answer is a successful one (procedure REASONING.1), otherwise the answer is a failed one and "Fc" consists of a list of the failed goals (procedure REASONING.2). The definition of procedure REASONING is given in Program 5.4.6 as follows:

```
/* Procedure REASONING: to test the answers */
reason_testing(Goal,Fc):-
        /* Procedure REASONING.1 */
        Fc==[],
        !.
reason_testing(Goal,Fc):-
        /* Procedure REASONING.2 */
        abolish(failure_son,2),
        split_reason(Fc,Reason),                          /* Procedure SPLIT-FC */
        not_exists(reject(Reason)),
        not_exists(consulted_sor(Goal,Reason)),
        not_exists(set_of_reason(_,goal(Goal),reason(Reason)),
        assert_reason_son_father(Goal),                   /* Procedure ASSERT-RSF */
        assert_set_of_reason(Goal,Reason),                /* Procedure ASSERT-SOR */
        !,fail.
```

Program 5.4.6: The definition of procedure REASONING

Procedure REASONING.1 will succeed when "Fc" is an empty list, i.e "Fc"=[], and so does procedure ASK which calls it and thus the proving is a successful one. The actions taken upon successful proving have already been described in Chapter 3. However if "Fc" is not an empty list, which means that the answer is not a successful one, then procedure REASONING.2 will be called.

The main purpose of procedure REASONING.2 is to assert a predicate *set_of_reason/3* which contains the information

about the failed goals of the query in the database. As the answer given is a failed one, this procedure (REASONING.2) is set to fail such that Prolog will automatically do a backtracking to find other answers until the whole proving tree is traversed or searched exhaustively. For instance, in deducing "a(X)" from given List 5.4.1 above, the backtracking will cause the whole proving tree as shown in Fig 5.4.2 above to be traversed or searched exhaustively.

However, before predicate *set_of_reason/3* is asserted into the database by procedure ASSERT-SOR, some actions and testing must be done first. As the variable "Fc" contains a list of pairs of failed goals and their associated (node) number, the actual reason clauses must be extracted from this list ("Fc") and this is achieved by calling procedure SPLIT-FC.

So procedure SPLIT-FC will return a variable "Reason" which contains a list of failed goals or reason clauses. Any repeating failed goals or reason clauses will also be deleted by the procedure SPLIT-FC. This procedure will also assert each pair of the reason clauses and its associated node number into the database as arguments of a predicate *failure_son/2*. This predicate *failure_son/2* is used to set up the predicates *reason_son_father/3* which will be asserted in the database after the various tests are successfully carried out.

The various tests which are carried out in the procedure REASONING.2 are to make sure that the list of failed predicates or reasons, i.e variable "Reason", has not been

rejected or has not been consulted or has not been asserted into the database before by checking the existence of predicates *reject(Reason)*, *consulted_sor(Goal,Reason)* and *set_of_reason(_, goal(Goal), reason(Reason))* respectively where variable "Goal" corresponds to the query goal itself. If all these three predicates do not exist in the database, then procedure ASSERT-RSF is called to assert predicates *reason_son_father/3* into the database by using the information in predicates *failure_son/2*.

Predicates *reason_son_father/3* which keep an information about the failed goals and their fathers are used to set up the failure path or tree for each failed goal or reason clause. The failure path or tree is a chain of the parent clauses (the unified KB clauses) starting from the failed goal leading up to the query goal itself. Thus the user will know exactly how each failed goal or fact is generated. For instance, referring back to Fig. 5.4.2, and by taking the failed goal "f(s,s)" (goal no. *13) as an example, the following failure tree or path can be produced from predicates *reason_son_father/3* at the end of an unsuccessful proving.

```
"f(s,s)" fails
        ===> "e(s,s) :- f(s,s)" fails
        ===> "d(s,s) :- e(s,s)" fails
        ===> "b(s) :- k(s),j(s),h(s),d(s,s)" fails
        ===> "a(s) :- b(s),c(s)" fails
        ===> "goal(a(X))" fails
```

Fig 5.4.4: The failure tree for the failure clause "f(s,s)"

The definition of procedures SPLIT-FC, ASSERT-RSF and ASSERT-SOR can be found in the Appendix. By using the same database given in List 5.4.1 above, we will try to prove an undeducible goal "a(X)" again. At the end of unsuccessful

proving, we will get the following list of predicates *set_of_reason/3* as given in Session 5.4.6 below. The variable "option(X)" is related to the option of predicates *set_of_reason/3* available in the database.

```
?-listing(set_of_reason).

set_of_reason(option(1),goal(a(r)),reason([c(r),b(r)])).
set_of_reason(option(2),goal(a(s)),reason([f(s,s),j(s),l(s)])).
set_of_reason(option(3),goal(a(s)),reason([g(s,s),j(s),l(s)])).

yes
```

Session 5.4.6: The listing of predicate set_of_reason/3

It can be seen from the above session 5.4.6 that there are three options for predicates *set_of_reason/3*. Each predicate *set_of_reason* shows the relationship among the failed goal and also the relationship with the query goal itself. For instance, for option(3) of *set_of_reason/3*, the query goal "a(s)" fails due to the failure of the goals "g(s,s)", "j(s)" and "l(s)". In other words, the reason why the query goal "a(s)" fails is the non-existence of predicates *knowledge(j(s))*, *knowledge(l(s))*, and *knowledge(g(s,s))*.

As before, by saving the details of rectifying the reason clauses for the next chapter, we use the same example to prove the goal "a(X)" as shown in the following session 5.4.7:

```
?- pcquest.
|:  a(_1).

NEXT QUESTION:
        a(_1)

The translation of its negation:
[]:-a(_1).
&&&&&&&&&&&&&&&&&&&&&&&&

>>answer: No, it cannot prove :
        " a(_1) "
========================================

OPTION: 1
The reason why the goal:
        "a(r)"
fails is due to the non-existence of the following facts:

        c(r)
        b(r)

However, we may able to prove the goal
after doing some corrections or additions

    --Do you like to continue ?y

If the following are true:
        c(r)
        b(r)

Do you like to try again by using the above assumption
   ---? y


RE-QUESTION:
        a(r)

The translation of its negation:
[]:-a(r).
&&&&&&&&&&&&&&&&&&&&&&&&

>>answer: Yes,
        a(r)
================================

PROVED: a(r) ?

yes
```

Session 5.4.7: Proving "a(X)".

It can be seen from the above session 5.4.7 that the new method of recording reason clauses is able to find the set of them, thus after one repeated proving, the proving is a successful one. This is better than the previous one as

shown in session 5.4.5 where the number of repeated provings is equal to the number of non-existent predicates for a particular goal.

In conclusion, the new procedure FACTPR (see Program 5.4.4) is able to produce a set of reason clauses for an unsuccessful query goal even though some of the reason clauses are at different hierarchy or level or branch of proving tree. This can clearly be seen from the results in the session 5.4.5 above.

5.5 Comments

In the section 5.4, we have discussed how to detect a faulty fact. This faulty fact can be assumed to be non-existent fact in the database. Without any modification to the original control of theorem prover program described in chapter 3, we will only able to find a first level of non-existent facts and need to repeat the proving to discover all the non-existent facts for a particular goal to be proved.

In order to discover all the non-existent facts for a particular goal to be proved, we have to traverse all branches of the proof tree, and this cannot be achieved without control modification of proving mechanism such that we can distinguish between the successful and unsuccessful proving of a goal.

The fault detecting algorithm described in section 5.4 is able to distinguish between the successful and unsuccessful proving, thus the algorithm is able to detect all the non-existent facts (or all the failed subgoals) for a goal to be successfully proved.

This algorithm is quite different to the contradiction backtracing algorithm by Shapiro although both use a backtracking method to discover faulty facts. Shapiro's algorithm will find one faulty rule or fact at a time by asking an oracle. The fault detecting algorithm described in section 5.4 will find a set of faulty facts which are inter-related in the sense that all of them must be true in order for a particular goal to be successfully proved.

The assumptions made by and the aims of Shapiro's technique and the faulty detecting algorithm are also different. Shapiro's technique is to find any faulty rules or facts used for an unexpected successfully proven goal. Our algorithm assumes that all the rules or facts are true and tries to find why we cannot prove a particular goal. In other words, our algorithm will find a reason behind an unexpected unsuccessful goal. Furthermore, our algorithm will not detect looping (control) faults as the looping test has been incorporated in the algorithm.

This technique can also be used to find the set of conditions for a particular fact or rule to be true. From the above session 5.4.6, the option(1) of predicate *set_of_reason/3* can be interpreted as that in order "*a(r)*" to be true, "*c(r)*" and "*b(r)*" must be true.

CHAPTER 6

A FAULT RECTIFICATION ALGORITHM

6.1 Introduction

In the last chapter (section 5.4), we have described an algorithm to detect the reasons behind the failure of the enquiries of the database. As promised in the last chapter, we will be describing an algorithm to rectify those reasons in this section. The reasons which have been detected are actually a set of conditions or predicates. There are two possibilities which cause those failures. These are as follows:

[1]. The wrong references in the enquiry.

[2]. The non-existence of the KB clauses in the database.

We will assume the first one above as the cause for the unsuccessful proving. If this is not the case then the second cause is assumed and the enquirer will be asked to choose whether to assert the non-existent KB clauses in the database or not. However before these assumptions are assumed, the enquirer will be presented with the reasons first and then will be given the choices of looking at the alternative suggestions to correct the reasons or looking at other sets of reason clauses if they exist.

The wrong references mean that the one or more arguments of the predicates of the query and the database do not match with each other. For instance, we would like to prove "man(rosie)" but instead in the database exists "man(nazrul)", so the argument "rosie" of "man(rosie)" does not match with the argument "nazrul" of "man(nazrul)". The wrong references do not only mean unmatching arguments but may also mean an unmatching predicate's name. For instance,

the predicate's name "man" of "man(rosie)" does not match with the predicate's name of "woman(rosie)".

The main problem here is to link between the proving procedure and the rectification procedure. We have to modify some of the top level procedures which link between the proving and the printing of result procedures as described in Chapter 3. We will described the link up between these two procedures in the section 6.3.

All the reasons detected are actually the failed goals (subgoals) and are not due to other reasons such as control problems or looping. Furthermore we assume that all the KB clauses are correct. However we may detect an incorrect KB clause in the matching process between the reason and the KB clauses. So all methods as reviewed in Bundy et al[1985] such as reordering rules, instantiating rules, updating rules or adding extra conditions to the rules are not applicable here.

Bourne[1977] examined the frequency of spelling errors in a sample drawn from 11 machine-readable bibliographic databases and concluded that errors are not only in the input queries, but also in the database itself. The failures may also caused by typing errors. Damerau[1964] indicates that 80% of typing errors are caused by transposition of two adjacent letters, one extra letter, one missing letter, or one wrong letter. These may also apply to predicates as their arguments can be considered as letters in the Damerau's finding. This supports the wrong references as one of the causes of the failures.

6.2 Fault Rectification Algorithm

In this section, we will describe an algorithm to rectify all the reason clauses detected by the fault detecting algorithm explained in the section 5.4 of chapter 5.

As explained in the chapter 3 that all questions will be negated first before being used to deduce an empty resolvent. In doing so, all relevant universal quantifiers will be changed to existential quantifiers by using the indicator "fq" to represent their equivalent questioned Skolem function. Accordingly if the unmatching arguments involve any questioned Skolem function then the wrong reference is referred to as an unmatching quantifier, for instance, the argument "fq(a)" (the questioned Skolem function) of "a(fq(a),s)" does not match with the argument "r" of "a(r,s)".

Each set of reason clauses ,if they exist, will be presented to the enquirer according to their increasing order of option or their occurrence provided that the set of reason clauses has not been rejected before by checking the existence of predicate *reject(R)* where R is the set of reason clauses itself. This is done by calling the procedure WHY-FAILS as shown in the following Program 6.2.1. This procedure is called from the answer printing procedure which forms the link up between the finding and rectification fault procedures (see the section 6.3.4).

```
/* to find out why it fails */
why_it_fails(Originalgoal,Listquant):-
        retract(set_of_reason(option(Option),goal(G),reason(R))),
        not_exists(reject(R)),
        assert(consulted_sor(G,R)),
        equatevars(G,Originalgoal),        /* procedure EQUATEVARS */
        list_reason(Option,G,R),           /* procedure LISTREASON */
        find_reason(Option,G,R,Listquant). /* procedure FINDREASON */
```

Program 6.2.11 The procedure WHY-FAILS

As explained in chapter 2 that Prolog adopts a kind of copying clause from the database, thus when we retract a predicate *set_of_reason/3*, different variable names were given to any universal quantifier which occurred in the said predicate. To overcome this different instantiatians of variable names, procedure EQUATEVARS or predicate *equatevars/2* is called from the above procedure WHY-FAILS such that any universal quantifier in the retracted goal "G" and in the original goal (question), "Originalgoal", will be instantiated with the same variables. The definition of procedure EQUATEVARS can be found in the Appendix.

Before the matching process between the reasons and the KB clauses in the database are carried out, the enquirer will be presented with the first set of reason clauses, then a choice of options as shown in the Table 6.2.1 must be chosen by the enquirer in the order to continue to the next step. All of these options and the set of reason presentation are carried out by procedure LISTREASON which is called from the above Program 6.2.1. So referring back the last session 5.4.7 of chapter 5, the printing starting from "*OPTION: 1*" until "*---Do you like to continue ?*" is done by this procedure LISTREASON. And the chosen option is typed after the remarks "*---Do you like to continue ?*".

Option	Meaning
a	abort
b	break
n	not accepting the given set of reason clauses so see other alternative.
l	list other sets of reason clauses.
s	show the failure tree of the given set of reason clauses.
t	type again the introduction.
y or <return>	yes to continue with a matching process.
others	displaying help table (like this one)

TABLE 6.2.1: Options for procedure LISTREASON

In the end, we have to choose either option "n" or "y" only. Other choices except "a" and "b" will lead to a repetition of asking whether to continue with the rectification process. If we type "n" then the procedure LISTREASON fails, so procedure WHY-FAILS will backtrack to retract another set of reason clauses, if it exists. Otherwise procedure WHY-FAILS fails and the actions taken afterwards will be explained in the next section 6.4. If we type "y" or press <return> then a matching process will be carried out by the predicate *find_reason/4* or procedure FINDREASON. The program of procedure LISTREASON is given in the Appendix.

In the matching or rectification process, the procedure FINDREASON will be divided into two subprocedures as shown in the following program 6.2.2:

```
/* Procedure FINDREASON */
find_reason(Option,goal,Reason,Listquant):-
        /* Procedure FINDREASON.1 */
        find_mismatch_clause(Reason),                  /* procedure FINDMATCH */
        suggestion(Option,Goal,Reason,Listquant).      /* procedure SUGGESTION */
find_reason(Option,Goal,Reason,Listquant):-
        /* Procedure FINDREASON.2 */
        assert(reject(Reason)),
        fail.
```

Program 6.2.2: Program FINDREASON

The first one of the subprocedures, procedure FINDREASON.1 is the main part of the procedure. Procedure FINDREASON.1 is divided into two parts, the first one is a matching process (procedure FINDMATCH) and the second one is a suggestion process (procedure SUGGESTION). If we reject all the suggestions to rectify the question based on that particular set of reason clauses, then the second procedure FINDREASON.2 is called to assert predicate *reject(Reason)* into the database to denote that the particular set of reason clauses, Reason, has been rejected and will not be consulted later again.

6.2.1 The Matching Process

The matching process (procedure FINDMATCH) is divided into three subprocedures as shown in Program 6.2.3 below. The first subprocedure, FINDMATCH.1 will find any KB clauses which differs with a member of the set of reason clauses in the sense of the wrong references as explained in the previous section (6.1) and then records its difference. If there is none of the said KB clauses, i.e procedure FINDMATCH.1 fails, then the second procedure, FINDMATCH.2 is called to continue the same process for the next member of the set of reason clauses until all members have been matched with all KB clauses (procedure FINDMATCH.3).

```
/* Procedure FINDMATCH */
find_mismatch_clause([R/T]):-
        /* Procedure FINDMATCH.1 */
        setup_predicate(R,Q,N),                          /* Step 1 */
        fast_setof(D,mismatch_clause(R,Q,D),List),       /* Step 2 */
        find_difference(R,List),                         /* Step 3: procedure FINDDIFFER */
        find_mismatch_clause(T),!.
find_mismatch_clause([R/T]):-
        /* Procedure FINDMATCH.2 */
        find_mismatch_clause(T),!.
find_mismatch_clause([]).       /* Procedure FINDMATCH.3 */
```

Program 6.2.3: Program FINDMATCH

The main part of the matching process as shown as procedure FINDMATCH.1 in the above Program 6.2.3 can be divided into three steps. These steps are setting up a potential predicate to be matched (Step 1), finding the mismatch clauses (Step 2) and finally finding the differences between the mismatch and the reason clauses (Step 3).

6.2.1.1 Setting up a predicate (Step 1)

In order to find all KB clauses which have potentially wrong references with the reason clause, R, a predicate *setup_predicate/3* will be called to create a clause "Q" which has the same predicate name and the number of arguments, N, with the reason clause, R, (a member of the set of reason) but its arguments are uninstantiated. For example, suppose that the reason clause is the clause "*g(s,s)*", then the following session will show the created clause, "Q".

```
?- setup_predicate(g(s,s),Q,N).

Q = g(_1,_2)
N = 2

yes
```

Session 6.2.1: An example of setting up a clause

6.2.1.2 Finding a set of mismatch clauses (Step 2)

In the second step, a library predicate *fast_setof/3* is called in order to produce a list of all variables "D" which satisfy the conditions set in the predicate *mismatch_clause/3* or procedure MISMATCHCL. So the variable "List" will consist of a list of KB clauses whose either some of its arguments or its predicate name differ or do not match with the reason clause "R". Then a difference list

will be produced by procedure FINDDIFFER (Step 3) before the next member of the set of reasons is to be matched with the same process again. Each member of "List" contains the mismatch clause itself and its type (either a query clause (q) or a knowledge clause (k)).

Two considerations are taken in finding a set of mismatch clauses. A mismatch clause is a KB clause whose arguments or predicate name differs from the reason clause. It should be noted here that the reason clause certainly cannot exist in the database.

The first consideration is that the reason clause, R, is matched with the query clauses, i.e the mismatch clause is a query clause. The second consideration is that the reason clause, R, is a query clause and it will be matched with any non-query clauses, i.e any KB clause which is not a query clause. These considerations are taken on the basis that the enquirer has typed the wrong question and the knowledge base does not contain any wrong fact.

The program of finding the mismatch clauses or procedure FINDMATCH is as follow (Program 6.2 4):

```
mismatch_clause(R,Q,[q,(Q:-T)]):-
        /* procedure MISMATCH.1: the first consideration */
        query_clause(Q:-T),
        Q\=R.
mismatch_clause(R,K,Rule):-
        /* procedure MISMATCH.2: the second consideration */
        convert(R,NotR),                /* line 1 */
        queryhead(NotR),                /* line 2 */
        mismatch_clause1(R,K,Rule).     /* procedure MISMATCH1 */
```

Program 6.2.4: Finding mismatch clauses: procedure MISMATCH

6.2.1.2.1 The first consideration (procedure MISMATCH.1)

The program for the first consideration, i.e when the reason clause is matched with the query clause, is shown as procedure MISMATCH.1 in the above Program 6.2.4.

If the mismatch clause is a query clause, then only the arguments of the mismatch and the reason clauses will be matched. In this case we do not match the predicate names of the mismatch and the reason clauses because the reason clause does not exist in the database and also the query clauses for a particular question are only temporarily asserted into the database. The new question will retract the old query clauses and will assert a new set of query clauses into the database. So that if we permit the predicate names to be compared and the suggestion to replace them is accepted, then after replacing predicate names is carried out, the old query clauses will be retracted from the database and will be replaced by the new set of query clauses resulting from the new question. For example, suppose the following are KB clauses in the database:

```
knowledge(human(X):-man(X)).
knowledge(~man(X):- ~human(X)).
knowledge(human(X):-woman(X)).
knowledge(~woman(X):-~human(X)).
knowledge(man(adam)).
knowledge(woman(eve)).
```

Database 6.2.1: KB clauses

If we ask the question "*exists(X, ~human(X)& ~man(X))*", then the following query clauses will be asserted into the database:

```
query(human(Y):- ~man(Y)).
query(man(Y):- ~human(Y)).
```

Database 6.2.2: The query clauses

Certainly we cannot prove or answer the above question as the (reason) clause " *~human(adam)*" does not exist in the database. If we permit the case that the mismatch clause has the same arguments but a different predicate name with the reason clause where the mismatch clause is a query clause, then we will get a suggestion that the predicate name, " *~human*", of " *~human(Y)*" is replaced with "*human*" as exist "*query(human(Y):- ~man(Y))*". If we accept this suggestion, then the new question becomes "*exists(X,human(X)& ~man(X))*". So the query clauses as in Database 6.2.2 will be retracted and will be replaced with a new set, i.e:

```
query(man(Y):- human(Y)).
query(~human(Y):- ~man(Y)).
```

Database 6.2.3: The new set of query clauses

It can clearly be seen from the above Database 6.2.3, that the old clause "*query(human(Y):- ~man(Y))*" does not exist any more in the database. As the replacement suggestion is based on this old clause and this one does not exist any more in the database, thus the new question will not be guaranteed to be successfully proved. Consequently the purpose of replacing something with something else which already or permanently exists in the database is defeated and this type of replacement does not always guarantee that the new question will be answered successfully.

Another reason why we can compare the arguments of the said clauses (the reason and the query clauses) is that one of the arguments of the reason clause or more may have been instantiated to the instantiated arguments of the given knowledge clauses but both clauses have the same predicate

name. For example, suppose the following knowledge clauses exist in the database:

```
knowledge((~wombat(X):-lives(X,londonzoo))).
knowledge((~lives(X,londonzoo):-wombat(X))).
```

Database 6.2.4: The knowledge clauses

If the question:

"~exists(X,wombat(X)&lives(X,twycrosszoo))"

is asked, then the following query will be asserted into the database:

```
query(wombat(fq(wombat0))).
query(lives(fq(wombat0),twycrosszoo)).
```

Database 6.2.5: The query clauses

As the fact "*lives(fq(wombat0),londonzoo)*" does not exist in the database, the above question will eventually fail or cannot be proved. However as the clause "*query(lives(fq(wombat0),twycrosszoo))*" exists in the database, the system will suggest that "*twycrosszoo*" of the question is to be replaced with "*londonzoo*". Consequently the new question becomes:

"~exists(X,wombat(X)&lives(X,twycrosszoo))"

and can be successfully proved. It can be seen here that "*londonzoo*" is the argument derived from the knowledge clause, i.e "*knowledge((~wombat(X):-lives(X,londonzoo)))*".

6.2.1.2.2 The second consideration

The second consideration in the finding mismatch clauses process, or procedure MISMATCH.2 as shown in the above Program 6.2.4, is that the reason clause is a head of any query clause and is matched with any knowledge clauses. So the first two lines of procedure MISMATCH.2 (i.e line 1 and

line 2) will test whether the reason clause is a head of any query clause. If so, the reason clause will be matched with the head of any knowledge clause.

In this case (the second consideration), both the reason and the non-query clauses with either the same predicate name but different arguments or the same arguments but the different predicate name will be matched or compared. This process will be carried out by procedure MISMATCH1 or predicate *mismatch_clause1/3* as shown in the following Program 6.2.5:

```
mismatch_clause1(R,K,[k,(K:-T)]):-
        /* procedure MISMATCH1.1 */
        fact_base_clause(K,T),          /* procedure FACTBASECL */
        R\=K.
mismatch_clause1(R,K,Rule):-
        /* procedure MISMATCH1.2 */
        functor1(R,Pred_name,N),
        f_list(R,[Pred_name|Args]),
        mismatch_clause1(Pred_name,N,R,Args,Rule).      /* procedure MISMATCH1/5 */
```

Program 6.2.5: Procedure MISMATCH1

So procedure MISMATCH1.1 will match the arguments of both the reason clause, R, and the non-query clauses as defined by procedure FACTBASECL, but with the same predicate name. On the other hand, the procedure MISMATCH1.2 will match any non-query clause with the reason clause which has the same arguments but different predicate name. This is carried out by procedure MISMATCH1/5 or predicate *mismatch_clause1/5* whose definition can be found in the Appendix. The definition of procedure FACTBASECL can also be found in the Appendix

After describing both considerations, let us see an example of how procedure MISMATCH works as shown in the following session by using the Database 6.2.1 and 6.2.2 and also

suppose that the reason clause is "*~human(m)*" such that the predicate which will be set up is "*~human(X)*":

```
?- fast_setof(B,mismatch_clause(~human(m),~human(X),B),L).

B = _1
X = _2
L = [[k, (human(m) :- man(m))], [k, (human(m) :- woman(m))],
     [k, (man(m) :- true)], [k, (~man(m) :- ~human(m))],
     [k, (~woman(m) :- ~human(m))]]
```

Session 6.2.2: An example of the usage of procedure MISMATCH

It can be seen from the above session that "L" is a list of mismatch clauses whose heads are having different predicate name but the same argument with the reason clause.

6.2.1.3 Finding the difference (Step 3)

In the step 2 above, a list of mismatch clauses has been set up. In this step 3, we will find the difference between the reason clause and all mismatch clauses. This is carried out by procedure FINDDIFFER or predicate *find_difference/2* which can be found in the Appendix.

In this step, only the mismatch clause which is thought to be a good replacement candidate will be asserted in the database as predicate *mismatch_pair([T,R,M,Diff],Len)* where "T" is a type of mismatch clause (either q(query) or k(knowledge)), "R" is a reason clause, "M" is a list of mismatch clauses which have the differences with the reason clause, "Diff" is a list of differences between the reason and mismatch clauses, and finally "Len" is a a number of differences between the reason and the mismatch clauses (i.e the length of the list "Diff").

A good replacement candidate clause is a mismatch clause which has neither of these properties:

[1] One of the body clauses of the mismatch clause is not equal to the reason clause.

For example, suppose the reason clause is "*g(s,s)*" and the mismatch clause is "*a(s,Y):-k(s,t),g(X,Y)*" then one of the body clauses of the mismatch clause, "*g(X,Y)*", is equal to the reason clause. Or, in terms of the Prolog language:

g(X,Y) = g(s,s)

where both variables "X" and "Y" will be instantiated to "s". So the above mismatch clause is not a good replacement one as there may cause a repetition of the suggestion as the clause "*g(s,s)*" does not exist in the database.

[2] The differences between the mismatch and the reason clauses do not involve any questioned Skolem function, i.e the Skolem function with the indicator "fq".

If a questioned Skolem function is involved then this case is considered as an unmatching quantifier and will be dealt in the suggestion process later.

For example, suppose the reason clause is "*g(s,s)*" and the mismatch clause is "*g(fq(a,X),s)*". So the difference between these two clauses is their first argument, i.e "s" and "fq(a,X)". So this mismatch clause is not a good replacement one. This is because the questioned Skolem function refers to the negation of the questioned universal quantifier as explained in Chapter 3.

6.2.2 The suggestion process

At the end of the matching process, predicates *mismatch_pair/2* are already in the database provided that some of the found mismatch clauses are good replacement candidates. As shown in the Program 6.2.2, this step is carried out by procedure SUGGESTION or predicate *suggestion/4* after procedure FINDMATCH is completely successful. The procedure SUGGESTION is divided into two subprocedures as shown in the following program (Program 6.2.6):

```
/* Procedure SUGGESTION */
suggestion(Option,Goal,Reason,Lisquant):-
        /* Procedure SUGGESTION.1 */
        process_reasons(Option,Goal,Reason,Reason),     /* procedure PROCESS-REASONS */
        acceptance(Option,Reason,Goal,Lisquant).        /* procedure ACCEPTANCE */
suggestion(Option,Goal,Reason,Lisquant):-
        /* Procedure SUGGESTION.2 */
        retract(accepted_substitution(S)),
        suggestion(Option,Goal,Reason,Lisquant).
```

Program 6.2.6: Procedure SUGGESTION

The first subprocedure, SUGGESTION 1 will deal with an actual suggestion process and its acceptance. If the enquirer does not agree with the suggestion then the second one, procedure SUGGESTION.2, will retract any previously accepted substitution (replacement) of the reason clause and will repeat the same process as the first subprocedure to find other possible suggestions.

As said above, the first subprocedure, SUGGESTION.1, will be divided into two stages; these stages are the reason clauses processing and the acceptance processing .

6.2.2.1. The reason clauses processing stage

This reason clauses processing step is carried out by procedure PROCESS-REASONS as shown in the following program, i.e Program 6.2.7:

```
/* procedure PROCESS-REASONS */
process_reasons(Option,Goal,Reason,[HeadR|TailR]):-
        process_suggestion(Option,Goal,Reason,HeadR),     /* procedure PROCESS-SUGGESTION */
        process_reasons(Option,Goal,Reason,TailR),
        !.
process_reasons(Option,Goal,Reason,[]).
```

Program 6.2.7: Procedures to process every reason clauses

As shown in the above Program 6.2.7, each member of the set of reason clauses will be processed one by one in order to make appropriate suggestions. The actual suggestion processing is carried out by the procedure PROCESS-SUGGESTION which is called from the procedure PROCESS-REASONS.

The program of the procedure PROCESS-SUGGESTION is shown in the following program (Program 6.2.8):

```
/* procedure PROCESS-SUGGESTION */
process_suggestion(Option,Goal,Reason,R):-
        /* procedure PROCESS-SUGGESTION.1 */
        functor(R,F,N),
        N1 is N+1,
        ordered_mismatch_clause(P,R,T,Diff],N1,1),      /* procedure ORDERED-MISMATCH */
        not_exists(reject([P,R,T,Diff]),
        subst_log(Diff),                                /* procedure SUBST-LOG */
        already_accepted_subs(Option,Goal,Reason,[P,R,T,Diff]),  /* procedure ACCEPT-SUBST */
        !.
process_suggestion(Option,Goal,Reason,R):-
        /* procedure PROCESS-SUGGESTION.2 */
        extract_fq(R,L),        /* procedure EXTRACT-FQ */
        process_others(R,L).    /* procedure PROCESS-OTHERS */
```

Program 6.2.8: Procedures to process suggestions for reason clauses

As the program 6.2.8 shown, the suggestion processing can be classified into two subprocedures. The first subprocedure (procedure PROCESS-SUGGESTION.1) is to deal with the predicate *mismatch_pair/2* and will eventually suggest a

substitution to the wrong reference. The second one (procedure PROCESS-SUGGESTION.2) deals with two things, i.e the mismatch quantifiers and the lack of knowledge and will suggest to change the quantifiers binding the question (goal) and assert a new KB clauses into the database respectively.

6.2.2.1.1 The first type: the substitution suggestion

As said above that the procedure PROCESS-SUGGESTION.1 will make appropriate suggestions of replacement or substitution for the wrong references. There are two ways how to implement the substitution. These ways are the clause and the atomic substitutions.

The clause substitution is a local substitution whereby we substitute the reason clause with the mismatch clause by disregarding the other clause in the question. For example, suppose the reason clause of the question "*a(q,r,s)&b(q,s)*" is the clause "*a(q,r,s)*" and its mismatch clause is "*a(w,r,s)*". In this local or clause substitution, the reason clause will be replaced by the mismatch clause such that the new question becomes "*a(w,r,s)&b(q,s)*". Although the difference between these two reason and mismatch clauses is their first argument, i.e "q" and "w", and also that "q" also occurred in the other clause, "*b(q,s)*" (of the old question), but this (other) clause, i.e "*b(q,s)*", remains the same in the new question.

The atomic substitution is a global substitution. In this global substitution, any substitution will be carried out on

the whole question. For example, by using the same example in the previous paragraph, the new question becomes "*a(w,r,s)&b(w,s)*" where all occurrences of "q" are substituted with "w".

There are advantages and disadvantages over these two types of substitutions. If there is no relationship between the arguments of the clauses in a question, then the local substitution has an edge over the global substitution.

For example, refer to the same example in the previous paragraph. The local substitution produces a new question where the clause "*a(q,r,s)*" is replaced by the clause "*a(w,r,s)*". This local substitution clause ("*a(w,r,s)*"), as we know, already exists in the database. So at least we know the direction of proving is towards to a successful one. We do not bother with the other clause ("*b(q,s)*") as it has already been successfully proved. However, in this example, the global substitution produces a new clause "*b(w,s)*" which has not been seen before, i.e a totally new clause which we do not know about its proving successful, and it may even produce more reason clauses during the next stage of proving.

If there is a relationship between the arguments of the question, then the global substitution has an edge over the local one. For example, suppose the reason clause of the question "*man(dan)&loves(dan,eve)*" is a clause "*man(dan)*", and the mismatch clause is a "*man(adam)*". The difference between the reason and the mismatch clauses is their argument, i.e "*dan*" and "*adam*".

Subsequently, the local substitution will produce a new question "*man(adam)&loves(dan,eve)*". This is quite different from the original meaning of the question. The original meaning of the question can be said to be '' there is a man called "dan" who loves "eve" ''. However the meaning of the new question resulting from the local substitution can be said as there is a man called "adam", and, "dan" loves "eve".

The new question resulted from the global substitution is "*man(adam)&loves(adam,eve)*" which can be said as there is a man called "adam" who loves "eve". This meaning is equivalent to the meaning of the original question. In this case, the enquirer has wrongly named the man. However we cannot say anything of the wrong reference in the context of the local substitution as the new question gives a totally new perspective.

As the predicate calculus representation of the question in this system usually results from the natural language, then the global substitution technique is adopted.

In this procedure PROCESS-SUGGESTION.1, for a particular member (the variable "R"), of set of reason clauses, a corresponding predicate *mismatch_pair/2* will be retracted by the procedure ORDERED-MISMATCH or the predicate *ordered_mismatch_clause/3* according to their ascending order of the number of differences between the reason and the mismatch clauses. The program of procedure ORDERED-MISMATCH can be found in the Appendix.

Before the replacement or substitution suggestion is made, some tests are carried out to make sure that the same suggestions have not been rejected (by checking the non-existence of predicate *reject([P,R,T,Diff])*) and have not been accepted before on the earlier proving attempts (i.e the successfulness of the procedure SUBST-LOG or the predicate *subst_log/1*). The program of the procedure SUBST-LOG can be found in the Appendix.

If the tests are successful, then the enquirer will be asked whether to accept the suggested substitution provided that the same suggested substitution has not been accepted before during the earlier attempt of the same question. In other words, the suggested substitution has not been accepted during processing of the other members of the same set of reason clauses. This is carried out by the procedure ACCEPT-SUBS or the predicate *already_accepted_subs/4*.

If the same suggestion has been accepted before then nothing is done and the system will process the next member of the set of reason clauses. If it has not been accepted before, then the enquirer will be presented with the suggested substitution and can choose one of the options as given in the following table (Table 6.2.2):

Option	Meaning
a	abort
b	break
n	not accepting the suggested substitution so see the next one.
l	list other possible substitution(s).
r	reject all possible substitutions.
s	show the failure tree of this particular reason clause.
t	type again the introduction.
w	why the substitution is suggested.
y or <return>	yes to accept the suggested substitution.
others	displaying help table (like this one)

Table 6.2.2: The options for the suggested substitution

The predicate *accept_substitution/4* will handle the above options and can be found in the Appendix. If the enquirer does accept the suggested substitution, then the suggested substitution will be added to the current list of the accepted substitution. The current list of the accepted substitution is kept in the database as an argument of the predicate *accepted_substitution/1*. This predicate will be used during the creation of new question later. For example, the following session shows the suggestion suggested and processed by procedure PROCESS-SUGGESTION.1:

```
                    !
If ' nazrul ' of the question's clause ' man(nazrul) ' is substituted
  with ' aizat '
                                    ---Do you agree ? n

' man(nazrul) ' fails, but exists
        ' man(aizat) '
                                    ---Do you agree ? b
```

Session 6.2.2: A part of suggested substitution session

The typed response after a remark "---Do you agree ?" is as shown in the above Table 6.2.2. The program of procedure ACCEPT-SUBS can be found in the Appendix.

6.2.2.1.2 The second type of suggestion processing

This second type of suggestion processing is carried out by the procedure PROCESS-SUGGESTION.2 as shown in the Program 6.2.8 above. This procedure is called when one of these cases occurred:

[1] If all the suggested substitutions have been rejected in the first subprocedure PROCESS-SUGGESTION.1.

[2] No corresponding mismatch clause exists in the database for a particular member of the set of reason clauses. In other words, no corresponding predicate *mismatch_pair/2* exists in the database.

[3] The difference between the mismatch and the reason clauses involves any questioned Skolem function.

In the second case above, it is possible that the reason clause may also contain any questioned Skolem function. So, another procedure is called within the procedure PROCESS-SUGGESTION.2 to extract any questioned Skolem function from the reason clauses. This procedure is called EXTRACT-FQ whose detailed program can be found in the Appendix. The procedure EXTRACT-FQ will return a list of any questioned Skolem function contained in the reason clause.

Then the resulting list of questioned Skolem function is passed to the procedure PROCESS-OTHERS (see Program 6.2.8 above). The program of the procedure PROCESS-OTHERS is as in the following program (Program 6.2.9):

```
/* procedure PROCESS-OTHERS */
process_others(R,[]):-
        /* procedure PROCESS-OTHERS.1 */
        !,
        assert_told_list(R).      /* procedure ASSERT-TOLDLIST */
process_others(R,L):-
        /* procedure PROCESS-OTHERS.2 */
        assert_skolemfq_list(L),  /* procedure ASSERT_LISTFQ */
        !.
```

Program 6.2.9: Procedures to process the second type of suggestion

If the list of questioned Skolem functions is an empty one, then the first subprocedure PROCESS-OTHERS.1 is executed to append the reason clause to the current list of subsidiary clauses (procedure ASSERT-TOLDLIST). A subsidiary clause is a reason clause which will be asserted into the database if it is agreed by the enquirer. The list of the subsidiary clauses is kept in the database as an argument of predicate *subsidiary_list/1*.

On the other hand, if any Skolem questioned function contained in the reason clause (i.e the list is not an empty one), the second subprocedure PROCESS-OTHERS.2 will be executed to append the list to the current list of the questioned Skolem function contained in the reason clause. The current list of the questioned Skolem function is kept in the database as an argument of the predicate *skolemfq_list/1*.

Both predicates *skolemfq_list/1* and *subsidiary_list/1* will be used in the acceptance processing and also in the creation of a new question. Both procedures ASSERT-TOLDLIST and ASSERT-SKOLEMFQ can be found in the Appendix.

6.2.2.2 The acceptance processing stage

The main work in the reason clauses processing stage as described in the last section (6.2.2.1), is to make a suggestion on the substitution of the reason clauses. Nothing is done on others, i.e asserting a new knowledge or changing the quantifier mismatching. So, at the end of the reason clauses processing stage, some or all of these three predicates *skolemfq_list/1*, *accepted_substitution/1* and *subsidiary_list/1* have been asserted into the database to indicate the list of unmatching questioned Skolem functions, the accepted suggested substitution for the reason clause, and the list of suggested subsidiary clauses respectively.

In this stage, the enquirer will be presented with the suggestions to assert more subsidiary knowledge into the

database or to change the particular quantifiers or both of them. This is carried out by the procedure ACCEPTANCE as shown in the following program (Program 6.2.10):

```
/* procedure ACCEPTANCE */
acceptance(Option,Reason,Goal,Listquant):-
        print_suggestions(Listquant),                    /* procedure PRINT-SUGGESTIONS */
        acceptance_choice(Option,Reason,Goal,Listquant), /* procedure ACCEPT-CHOICE */
        acceptance_action,                               /* procedure ACCEPT-ACTION */
        !.
```

Program 6.2.10: Procedure ACCEPTANCE

The above procedure (see Program 6.2.10) can be divided into three substages. These substages are printing the suggestions, choosing the acceptance options and finally the action taken upon the acceptance of the suggestions.

6.2.2.2.1 The Printing of the suggestions substage

The first substage is carried out by the procedure PRINT-SUGGESTION in order to print all sorts of suggestions. There are three sorts of suggestions depending on the existence of the three predicates, i.e *skolemfq_list/2*, *accepted_substitution/1* and *subsidiary_list/1*.

If predicate *accepted_substitution/1* exists, then a summary of the suggested substitutions which have been accepted is printed again as a reminder to the user. And also if predicate *subsidiary_list/1* exists then a list of clauses which will be suggested to be asserted into the database as subsidiary clauses, is printed for confirmation.

And finally if predicate *skolemfq_list/1* exists, then a suggestion of changing the relevant quantifier into its opposite quantifier is also printed for a confirmation.

There are some difficulties to be overcame when printing the mismatch quantifier. This is due to the method of copying clause technique implemented in Prolog as explained in the Chapter 2.

So the universal quantifier referred in the suggestion does not match with the universal quantifier referred in the original question. For example, suppose that the original question in terms of the variables given by Prolog is:

"*all(_1,man(_1) => all(_2,woman(_2) => likes(_1,_2)))*"

and further that this question cannot be proved due to the mismatched quantifier of the universal quantifier "_2" (the quantifier binding "*woman(_2)*"). However as we kept this variable in the database, then the printing of the suggestion of quantifiers changing may appear as follows:

"*if all(_4,...)*" *is replaced with* "*exists(_4,...)*"

In this case, we do not know which universal quantifier the suggestion referred to as there are two universal quantifiers in the original question.

To overcome this problem, we need to have a variable which keeps the original variables for any quantifier contained in the question as an argument which will be passed on from the top level predicate until the predicate to print the suggestion. So, the variable "Listquant" in the procedure ACCEPTANCE which is passed to procedure PRINT-SUGGESTIONS contains a list of quantifiers which have been Skolemized during the negation of the question. By having this list of skolemised quantifiers, the above suggestion will appear as:

"if all(_2,...)" is replaced with "exists(_2,...)"

It can be seen that the suggestion clearly refers to the universal quantifier binding the predicate "*woman(_2)*".

In order to pass the list of Skolemized quantifiers from the process of the negation of the question through to the printing the suggestion, some modifications need to be done in the process of the negation of the question. This will be explained in the process of linking up between the process of finding and rectifying faults later (see the next section 6.3).

The following session 6.2.2, shows how the suggestion is printed by the procedure PRINT-SUGGESTIONS by assuming that all the said three predicates exist:

```
If the following clause are true:
     " happy(aizat) "
     " likes(aizat,sweets) "

If "all(_1,...)" is replaced with "exists(_1,...)"

And, so you have already agreed that:
## " nazrul " of the question's clause " boy(nazrul) " is substituted
        with " aizat "


Do you like to try again by using the above assuaption(s)
   ---? b
```

Session 6.2.3: The sample printing done by procedure PRINT-SUGGESTION

It should be noted here that the sample printing above is not a real problem such there are no relationships among three given suggestions. In a real problem, there would be relationships among them.

6.2.2.2.2 The substage of choosing the acceptance options

At the foot of printed suggestion in the first substage, the enquirer will be asked to type an option for the next stage. By referring back the Session 6.2.3 above, the response is typed after a remark "---?" which is below a remark "Do you like to try again by using the above assumption(s)". In this substage, the typed option will be processed by procedure ACCEPT-CHOICE. The options available is given as in the following table (Table 6.2.3). The detail program of procedure ACCEPT-CHOICE can be found in the Appendix.

Option	Meaning
a	abort
b	break
n	not accepting the suggestions, so see the other alternative.
s	show the failure tree of this particular reason clause.
t	type again the introduction.
w	why the suggestion is suggested.
y or <return>	yes to accept the suggested substitution.
others	displaying help table (like this one)

Table 6.2.3: The options for accepting a suggestion

Although there are many options available, in the end there are only two options which will carry through into the next stage. If we response with "n" to refuse the given suggestion, then the system will backtrack to find any other possible suggestion, i.e back to the procedure PROCESS-SUGGESTION (see Program 6.2.8) If we do accept the suggestion, the actions taken upon acceptance will be described in the following section 6.2.2.2.3.

6.2.2.2.3. The acceptance action substage

If the suggestions are accepted, the procedure ACCEPT-ACTION will be executed to assert denotation predicates into the database. A denotation predicate is a predicate to denote a

particular meaning. The denotation predicate which will be asserted into the database are predicates *newsubsidiary/0* and *atomic_equiv/1*. The program of procedure ACCEPT-ACTION can be found in the Appendix.

The list of the differences between the reason and the mismatch clauses will be asserted in the database as an argument of the predicate *atomic_equiv/1*.

All members (fact clauses) of the list *subsidiary_list/1* will be asserted in the database as an argument of the predicate *subsidiary/1*. A denotation predicate *newsubsidiary/0* is also asserted into the database to denote that the predicates *subsidiary/1* have been asserted into the database. These predicates *subsidiary/1* will be considered as a new set of KB clauses and will be used to prove other questions as well as any new question resulting from the rectification process. So the new definition of KB clauses is as follows:

```
/* KB clauses */
knowledge_base(Q):-
        query(Q).       /* a query clause */
knowledge_base(K):-
        fact_base(K).   /* a non-query clause */

/* non-query clauses */
fact_base(K):-
        clause(knowledge(K),true).
fact_base(K):-
        clause(plausible(K),true).
fact_base(K):-
        clause(subsidiary(K),true).
```

Program 6.2.11: The new definition of KB clauses.

The new predicate *fact_base/1* is needed here to differentiate between query and non query clauses. This is a quite important feature in finding the mismatch clause as explained in section 6.2.1.2 above.

6.3 The link up procedure

In this section, we will be describing how the link up between the rectifying faults and the proving (faults finding) processes is carried out. The main technique used to link up those processes is a backtracking method adopted by the Prolog. So, many procedures described in the chapter 3 will be modified to suit the link up process. The link up procedure must be able to reformulate a new question according the accepted suggestion and then reprove it and also to cut out any unreasonable backtracking.

The first procedure needs to be modified is a top level procedure, i.e procedure QUEST or predicate *question/1* as described in Program 3.3.3.1 (see Chapter 3). So the new procedure QUEST is as follows:

```
/* the new procedure QUEST */
question(X):-
        clear_pc,                              /* Step 1: procedure CLEARPC */
        reset_newquery(X,Z),                   /* Step 2: procedure RESET-QUERY */
        pc_to_hornclause(Z,Clause,Listquant), /* Step 3: procedure PC-HORN */
        retry_search(Z,Listquant),             /* Step 4: procedure RETRY-SEARCH */
        answer_search(Clause,Z,Y,Listquant),   /* Step 5: procedure AS */
        print_answer(Z,Y,Clause,Listquant).    /* Step 6: procedure PRINT-ANSWER */
```

Program 6.3.1: The new definition of procedure QUEST

There are some obvious differences if we compare between the new and the old definitions, i.e the new definition has six steps compared to the old one which has four steps. The step 1 (procedure CLEARPC - see Appendix) of both old and new definition, and also step 3 of the old definition and step 5 of the new one (i.e procedure AS -see Program 3.3.3.3 of chapter 3) are the same. So, we will only describe the other steps, i.e steps 2, 3, 4 and 6.

6.3.1 Step 2: to set up a new query

This step 2 which is aimed to set up a new query resulted from the reformulation process is carried out by procedure RESET-QUERY. The procedure RESET-QUERY is given as follows:

```
/* procedure RESET-QUERY */
reset_newquery(Originalquest,Originalquest).    /* procedure RESET-QUERY.1 */
reset_newquery(Originalquest,Newquery):-
        /* procedure RESET-QUERY.2 */
        reset_newquery1(Originalquest,Newquery), /* procedure RESET-QUERY1 */
        print_newquery(Newquery).                /* procedure PR-NEWQUERY */

/* procedure RESET-QUERY1 */
reset_newquery1(Originalquest,Newquery):-
        /* procedure RESET-QUERY1.1 */
        retract(new_query(Newquery)),
        abolish(current_query,1),
        assert(current_query(Newquery).
reset_newquery1(Originalquest,Newquery):-
        /* procedure RESET-QUERY1.2 */
        exists(new_query(_)),
        reset_newquery1(Originalquest,Newquery).
```

Program 6.3.2: To set up a question to be proved

If the query or question is an original one, then take it as a question to be proved (Procedure RESET-QUERY.1). In other words, during the first attempt of proving the original question, nothing is done, i.e take the original question. Otherwise if the question is not the original one, i.e the new reformulated question, then take this one as a question to be proved (procedure RESET-QUERY.2). Actually, the second procedure RESET-QUERY.2 is executed as a result of Prolog's backtracking, i.e. when procedure RETRY-SEARCH fails.

The new reformulated query or question exists if predicate *new_query/1* exists in the database. This new query is retracted from the database by procedure RESET-QUERY1.1. The second procedure RESET-QUERY1.2 is used to set another new reformulated query for the next attempt of proving. Procedure RESET-QUERY1 is a sort of an iterative or a WHILE-DO procedure whereby as long as exists predicate

new_query/1, this procedure will be executed. The predicate *new_query/1* is asserted into the database at step 4 (procedure RETRY-SEARCH).

After the new reformulated query has been set up, then it will be printed. The printing of the new query is done by procedure PR-NEWQUERY which can be found in the Appendix.

6.3.2. Step 3: converting a PC into Horn Clauses

In this step, a question in the predicate calculus form is converted to Horn clause by procedure PC-HORN. The program of procedure PC-HORN is as follows:

```
/* procedure PC-HORN */
pc_to_hornclause(X,Clause,Listquant):-
        list_quantifiers(X,Allquantifiers),                 /* procedure LIST-QUANT */
        question_to_hornclause(*X,Clause,Sklist),           /* procedure QTHC */
        merge_quantifiers(Sklist,Allquantifiers,Listquant), /* procedure MERGE-QUANT */
        !.      /* to prevent a useless backtracking */
```

Program 6.3.3: Converting a PC into Horn clauses

Before the question is actually converted into Horn clauses by procedure QTHC (see Program 3.3.3.2 of chapter 3), all Prolog variables assigned to all quantifiers contain in the question is recorded by procedure LIST-QUANT denoted by variable "Allquantifiers" in the above program 6.3.3. A list of skolemized quantifiers which is denoted by the variable "Sklist" is also recorded by the procedure QTHC.

As we are only interested in the list of Prolog variables assigned to the skolemized quantifiers and their original quantifier before skolemization for the purpose of printing the suggestion (see section 6.2.2.1.2), and furthermore, the list "Allquantifiers" consists of all the Prolog variables assigned to all quantifiers occurred in the question, then

procedure MERGE-QUANT is called after the conversion of PC to Horn clauses in order to record the Prolog variables assigned to skolemized quantifiers only.

The programs of procedures LIST-QUANT and MERGE-QUANT can be found in the Appendix.

6.3.3 Step 4: procedure RETRY-SEARCH

This step is a reformulation of the new question according to the accepted suggestions. After we accept the suggestion, there are three predicates which may be asserted into the database depending on the certain cases as explained in section 6.3.2.2 before. These predicates are *atomic_equiv/1*, *skolem_fq/1* and *newsubsidiary/0*. This step is carried out by procedure RETRY-SEARCH as shown below:

```
/* procedure RETRY-SEARCH */
retry_search(Oldquery,Listquant).   /* procedure RETRY-SEARCH.1 */
retry_search(Oldquery,Listquant):-
        /* procedure RETRY-SEARCH.2 */
        retry_search1(Oldquery,Listquant). /* procedure RETRY-SEARCH1 */

/* procedure RETRY-SEARCH1 */
retry_search1(Oldquery,Listquant):-
        /* procedure RETRY-SEARCH1.1 */
        not_exists(atomic_equiv(_)),
        not_exists(skolemfq_list(_)),
        exists(newsubsidiary(_)).
retry_search1(Oldquery,Listquant):-
        /* procedure RETRY-SEARCH1.2 */
        reformulate_question(Oldquery,Listquant), /* procedure REFORMULATEQ */
        abolish(set_of_reason,3),
        abolish(reason_son_father,3),
        abolish(rsf_head,3),
        abolish(newsubsidiary,0),
        abolish(query,1),
        fail.
retry_search1(Oldquery,Listquant):-
        /* procedure RETRY-SEARCH1.3 */
        ( exists(newsubsidiary);
                exists(skolemfq_list(_));
                exists(atomic_equiv(_)) ),
        retry_search1(Oldquery,Listquant).
```

Program 6.3.4: Reformulating or reproving the question

The first procedure RETRY-SEARCH.1 is executed first time at the first attempt of the proving of a particular question. The second procedure RETRY-SEARCH.2 is executed as a result of backtracking when the suggestion are accepted (see section 6.2.2.1.3) such that the procedure PRINT-ANSWER fails (see Program 6.3.1).

However the reformulation of the new question depends on the existence of either predicate *skolemfq_list/1*, or *atomic_equiv/1* or both of them. If only predicate *newsubsidiary/0* exists in the database (see procedure RETRY-SEARCH1_1 of program 6.3.4 above), then no reformulation is carried out but the same question will be proved once again by using the extra subsidiary clauses which have been asserted before into the database as new knowledge.

Otherwise, a new question will be formulated by procedure REFORMULATEQ. After the new question is reformulated, all relevant predicates will be abolished or retracted from the database as the reproving of the new question soon will be executed again. Those relevant predicates are *set_of_reason/3*, *reason_son_father/3*, *rsf_head/3*, *query/1* and *newsubsidiary/0*. And the procedure RETRY-SEARCH1.2 is set to fail in order a backtracking can occur such that procedure RESET-QUERY.2 will be executed.

The third procedure RETRY-SEARCH1.3 is an iterative or a WHILE-DO procedure such that a new question can be reformulated or the question is reproved again for the next

proving attempt after the new suggestions are accepted for the previously new reformulated question.

As we said above that the reformulation of new question is carried out by procedure REFORMULATEQ which is shown as below:

```
/* procedure REFORMULATEQ */
reformulate_question(Oldquery,Listquant):-
        /* procedure REFORMULATEQ.1 */
        retract(atomic_equiv(Q)),
        new_goal(Newquery2),
        subst_skolem_not(Newquery2,Newquery1,Q),
        quantifiers_changing(Newquery1,Newquery,Listquant), /* procedure QUANT-CHANGE */
        assert(new_query(Newquery)),
        !.
reformulate_question(Oldquery,Listquant):-
        /* procedure REFORMULATEQ.2 */
        not_exists(atomic_equiv(_)),
        exists(skolemfq_list(Sq)),
        quantifiers_changing(Oldquery,Newquery,Listquant), /* procedure QUANT-CHANGE */
        assert(new_query(Newquery)),
        !.
```

Program 6.3.5: Procedure REFORMULATEQ

There are two cases in the reformulation process of the new question. The first one is that the predicate *atomic_equiv/1* exists in the database (procedure REFORMULATEQ.1). If this is the case, first of all, the atomic or global substitutions on the instantiation of the failed old question ("Newquery2") are carried out by predicate *subst_skolem_not/3*, and then finally all relevant quantifiers will be changed by procedure QUANT-CHANGE provided that predicate *skolemfq_list/1* exists.

The second case is that the predicate *atomic_equiv/1* does not exist and but the predicate *skolemfq_list/1* exists. If this is the case then the procedure QUANT-CHANGE is called to change the relevant quantifiers.

At the end of both cases, a predicate *new_query(Newquery)* is asserted into the database where "Newquery" is a new reformulated question. In both cases, we do not care whether the predicate *newsubsidiary/0* exists or not in the database as the relevant subsidiary clauses have already been asserted in the database. The program of procedure QUANT-CHANGE can be found in the Appendix.

6.3.4. Step 6: Printing the answer

In this step the top level predicate for printing the answer for the question is carried out by procedure PRINT-ANSWER. This procedure is equivalent to the procedure PA (see Program 3.3.3.4 of chapter 3). The only difference is the extra argument in the predicate *print_answer/4* of procedure PRINT-ANSWER which is shown as follows:

```
/* procedure PRINT-ANSWER :printing the answer or solution */
print_answer(Q,Y,[],Listquant):-
        /* Procedure PRINT-ANSWER.1: the clause is an inconsistent one */
        affirm(Ans),
        answer_form(Y,Ans),     /* procedure AF */
        write(' The clause is an inconsistent one '),
        !.
print_answer(Q,Y,Clause,Listquant):-
        /* Procedure PRINT-ANSWER.2: print either successful or failure proving */
        Clause\==[],
        print_answer0(Q,Y,Listquant).   /* procedure PRINT-ANSWER0 */
```

Program 6.3.6: Procedure PRINT-ANSWER

So the description of procedure PRINT-ANSWER is exactly the same as the procedure PA. However, the second procedure PRINT-ANSWER.2 has other aims than the printing of answer, it also means to rectify the failure question. This is

actually carried out by procedure PRINT-ANSWER0 which is called by procedure PRINT-ANSWER.2. The following is a program of procedure PRINT-ANSWER0:

```
/* procedure PRINT-ANSWER0 */
print_answer0(Q,Y,Listquant):-
        /* procedure PRINT-ANSWER0.1 */
        nonvar(Y),
        affirm(Ans),
        answer_form(Y,Ans),     /* procedure AF */
        !,
        rectifiers(Y,Ans,Listquant).    /* procedure RECTIFIERS */
print_answer0(Q,Y,Listquant):-
        /* procedure PRINT-ANSWER0.2 */
        affirm(yes),
        exists(set_of_reason(_,_,_)),
        rectifiers(Y,no,Listquant).     /* procedure RECTIFIERS */
print_answer0(Q,Y,Listquant):-
        /* procedure PRINT-ANSWER0.3 */
        exists(toptry([])),
        not_exists(set_of_reason(_,_,_)),
        test_finish_fact(Q).    /* procedure TFF */
```

Program 6.3.7: Procedure PRINT-ANSWER0

Procedure PRINT-ANSWER0 can be divided into three subprocedures. The first one, procedure PRINT-ANSWER0.1, is meant to print the answer if the proving is successful or otherwise to detect the reason clauses and their rectification.

The second subprocedure, PRINT-ANSWER0.2, is aimed to find any other possible suggestions for the question provided that the set of reason clauses exists in the database and also the proving of the question is successful (i.e predicate "*affirm(yes)*" exists in the database). Although the question is successfully proved, it may also produce some sets of reason clauses. So this subprocedure will handle those set of reason clauses as assuming that the

question can not be proved. For instance, let us see the following session (6.3.1):

```
?-listing(knowledge).

knowledge(human(nazrul)).
knowledge(human(aizat)).
knowledge(happy(nazrul)).
knowledge((success(_1):-happy(_1),human(_1))).

yes

?-pcquest.
|:success(X).

NEXT QUESTION:
        success(_1)

The translation of its negation:
[]:-success(_1).
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

>>answer: Yes,
        success(nazrul)
============================

PROVED: success(nazrul) ? ;

OPTION 1:
The reason why the goal:
        " success(aizat) "
fails is the non-existence of the following knowledge:

        happy(aizat)

However, we may able to prove the goal
after doing some corrections or additions

Do you like to continue ? a
```

Session 6.3.1: Finding other suggestion for a successful question

From the above session (6.3.1), we can see that although the question "*success(X)*" is successful, i.e "X" is instantiated to "nazrul", but the goal (question) "*success(aizat)*" fails. So the second subprocedure PRINT-ANSWER0.2 handles this type of result.

The third subprocedure, PRINT-ANSWER0.3, is to mark the end of proving and its exactly the same with procedure PA0.2

(see Program 3.3.3.5 of chapter 3) but with the extra condition, i.e no more predicate *set_of_reason/3* exists in the database.

So, in the first and second subprocedures, i.e procedures PRINT-ANSWER0.1 and PRINT-ANSWER0.2, the process of finding the reason clauses is carried out even though the proving is not necessarily unsuccessful. This finding is carried out by procedure RECTIFIERS which is as follows:

```
/* procedure RECTIFIERS */
rectifiers(Y,yes,Listquant):-
        /* procedure RECTIFIERS.1 */
        write_proved(user,Y),
        get0(X),
        answer_response(R,Y,Listquant). /* procedure ANS-RESPONSE
rectifiers(Y,no,Listquant):-
        /* procedure RECTIFIERS.2 */
        why_it_fails(Y,Listquant),      /* procedure WHY-FAILS */
        clear_pc1,                      /* procedure CLEARPC1 */
        !,fail.
```

Program 6.3.8: Procedure RECTIFIERS

The first subprocedure, RECTIFIERS.1, is to print the successful answer and the enquirer will be given a choice of options by procedure ANS-RESPONSE. The options available are the same as in the Table 3.3.1 of chapter 3. So the procedure AR (see Appendix) is equivalent to procedure ANS-RESPONSE apart from the different number of arguments. The program ANS-RESPONSE can be found in the Appendix). In other words, the procedure RECTIFIERS.1 is equivalent to procedure MORE-ANS (see Program 3.3.3.6 of chapter 3).

The second subprocedure, RECTIFIERS.2, is the main link up between the two procedures of finding and rectifying faults. The finding faults is called within procedure PC-HORN above (see Program 6.3.3) or to be precise from procedure ASK (see

Program 6.2.5 in section 6.2 above). The rectifying faults procedure which is called from this subprocedure RECTIFIERS.2 is a procedure WHY-FAILS (see Program 6.3.1 in section 6.3 above). After procedure WHY-FAILS is successfully executed, procedure CLEARPC1 is called to reset all the relevant control and denotation predicates. The program of procedure CLEARPC1 can be found in the Appendix.

This second subprocedure (RECTIFIERS.2) is set to fail such that a backtracking can occur to the point of step 4 (procedure RETRY-SEARCH) given that there is no more other possible answers to the question. However the first subprocedure (RECTIFIERS.1) will backtrack to the step 5 (procedure AR) when the user requires more answers to the question.

6.4. Examples

We have already described the fault detection algorithm (section 5.4) which has been incorporated with the theorem prover program (chapter 3), the fault rectification algorithm (section 6.2) and a system to link up both algorithms. In this section we will show some examples following the description of both algorithms. For simplicity, suppose the following KB clauses have already been asserted in the database:

```
knowledge(man(aizat)).
knowledge(man(nazrul)).
knowledge(man(rzak)).
knowledge(woman(rosie)).
knowledge(woman(sarah)).
knowledge(woman(eve)).
knowledge((loves(aizat,rosie)).
```

Database 6.4.1: *A list of sample KB clauses*

6.4.1. Example 1

In this section, we will show how a substituted suggestion is dealt.

```
?-pcquest.
|: man(rosie).

NEXT QUESTION:
        man(rosie)

The translation of its negation:
[]:-man(rosie).
&&&&&&&&&&&&&&&&&&&&&&&&&

>>answer: No, it cannot prove :
        " man(rosie) "
=========================================
```

Session 6.4.1.1: The first part of a session to prove "man(rosie)"

Certainly, from the above Database 6.4.1, we cannot prove or deduce "man(rosie)". Up to this stage, all sets of reason clauses have already been recorded by the system (see section 5.4). In other words, the first five steps of

procedure QUEST (see Program 6.3.1 of section 6.3) has finished. Let us see the continuation of the proving session which is actually the step 6 of procedure QUEST:

```
OPTION 1:
The reason why the goal:
        "man(rosie)"
fails is due to the non-existence of the the following fact:

        man(rosie)

However, we may able to prove the goal
after doing some corrections or additions

    --Do you like to continue ?1

Sorry, no other set of reason clauses

    --Do you like to continue ?s

"man(rosie)" fails
        ==> "goal(man(rosie))" fails

    --Do you like to continue ?y
```

Session 6.4.1.2: The second part of proving session of "man(rosie)"

As the proving fails, procedure PRINT-ANSWER0.1 (see Program 6.3.7 of section 6.3.4) will handle this situation. The system will list the first option of the reason why the goal (question) fails as shown in the above session 6.4.1.2, and will ask the user "*Do you like to continue ?*".

In the above session (6.4.1.2), the user response with "1" in order to see other possible set of reason clauses and the system will list all other possible set of reason clauses, if they exist, or will print a remark that no other set of reason clauses as shown in the above session 6.4.1.2, if this is the case. The user then responds again to the question by typing "s" in order to see the failure tree of the reason clause as shown in the above session where, in this case, the goal itself is actually the reason clause.

All other valid responses to this question are as in the Table 6.2.1 (of section 6.2). Finally, the user agrees to continue the session by typing an option "y" in order to see the rectification suggestions which will be made by the system. This reason clauses processing is carried out by procedure PROCESS-REASONS (see Program 6.2.7 of section 6.2.2.1). where then the suggestion will be made by procedure SUGGESTION (see Program 6.2.8 of section 6.2.2.2). The following session shows the continuation of proving session of "man(rosie)":

```
   if " rosie " of the question's clause " man(rosie)" is substituted
     with "aizat"
                          ---Do you agree ? w

     " man(rosie)" fails, but exists
         " man(aizat) "

...So,
   if " rosie " of the question's clause " man(rosie)" is substituted
     with "aizat"
                          ---Do you agree ? l

   The other possibility of substitution of "man(rosie)" are as follows:

     1 "rosie" can also be replaced with "nazrul" as exists
         "man(nazrul)"

     2 "rosie" can also be replaced with "rzak" as exists
         "man(rzak)"

     3 "man" can also be replaced with "woman" as exists
         "woman(rosie)"

...So,
   if " rosie " of the question's clause " man(rosie)" is substituted
     with "aizat"
                          ---Do you agree ? n

   if " rosie " of the question's clause " man(rosie)" is substituted
     with "nazrul"
                          ---Do you agree ? n

   if " rosie " of the question's clause " man(rosie)" is substituted
     with "rzak"
                          ---Do you agree ? n

   if " man " of the question's clause " man(rosie)" is substituted
     with "woman"
                          ---Do you agree ? y
```

Session 6.4.1.3: The third part of proving session of "man(rosie)"

In the above session (6.4.1.3), the various substitutions are presented by procedure ACCEPT-SUBS (see Appendix) after the matching process is carried out by procedure ORDERED-MISMATCH (see Appendix). Both procedures are called form procedure SUGGESTION.1 (see Program 6.2.8 of section 6.2.2.1 and also section 6.2.2.1.1). The above session shows the system's responses to the user's response of "*Do you agree*" such as "w" to see why the suggested substitution is made; "l" to list other possibility of substitution; "n" and "y" to reject and accept the suggested substitution respectively. Other valid responses of "*Do you agree*" is as shown in Table 6.2.2 of section 6.2.2.1.1. The following session shows part of the acceptance of other rectifications (see section 6.2.2.2), if they exist, of the proving session after the user has initially agreed to substitute "man" of "man(rosie)" with "woman".

```
...so you have already agreed that:
   "man" of the question's clause "man(rosie)" is substituted with
        "woman"

Do you like to try again by using the above assumption
   ---? y
```

Session 6.4.1.4: The fourth part of the proving session of "man(rosie)"

All the suggested substitutions which has been initially agreed in the third part will be presented again by the system for a confirmation. The valid responses to "*Do you like to try again by using the above assumption ?*" are shown in Table 6.2.3 of section 6.2.2.2.2.

All the processing shown in sessions 6.4.1.2, 6.4.1.3 and 6.4.1.4 above are carried out by procedure RECTIFIERS.2 (see Program 6.3.8 of section 6.3.4). As shown in session 6.4.1 4, the user confirms his (her) initial agreement of

the suggested substitution. Consequently procedure RECTIFIERS.2 is set to fail (by commands "*!,fail*") and thus procedure PRINT-ANSWER (The step 5 of procedure QUEST - see Program 6.3.1 of section 6.3) fails. Prolog will automatically backtrack to the step 3 of procedure QUEST (i.e procedure RETRY-SEARCH) where the new question will be reformulated.

As shown in session 6.4.1.4, there is only one substitution to be made to the original question (by procedure RECTIFIERS.2). As this is the case, the procedure RECTIFIERS will also be set to fail and Prolog will backtrack again to the step 2 (procedure RESET-QUERY) in order to reset the new question and start the proving all over again as shown in the following session (6.4.1.5):

```
RE-QUESTION:
   woman(rosie)

The translation of its negation:
[]:-woman(rosie)
%%%%%%%%%%%%%%%%%%%%%%%%%

>>answer: Yes,
   woman(rosie)
*****************************

PROVED:
   woman(rosie) ? ;

no more answer!
```

Session 6.4.1.5: The final (fifth) part of proving session of "man(rosie)"

And finally we succeed in proving a new question "woman(rosie)" after starting with the unsuccessful one, i.e "man(rosie)".

6.4.2 Example 2

In this example, we will be dealing with a non-matching reason clauses. In other words, the detected reason clauses do not have any mismatch KB clauses and as a result, new knowledge will be asserted into the database. Suppose we would like to prove "boy(tony)". We will use the same part of the session as explained in example 1 (see section 6.4.1) for an explanation of how the rectification is carried out.

By skipping the first part of a proving session as it is obvious that the proving fails. And also that the goal "boy(tony)" fails because it does not exists in the Database 6.4.1., so assuming the user agrees to see the rectification process. This means that we will also skip the second part of the proving session.

The system will then process the reason clauses and will present suggested substitutions to the user. As there is no corresponding mismatch KB clauses, the reason clause "boy(tony)" will be processed by procedure PROCESS-SUGGESTION.2 (see Program 6.2.8 of section 6.2.2.1). The third part of the proving session will also be skipped here as no substitution will be suggested. So we will move on the fourth part of the proving session which are as follows:

```
If the following clause is true:
    "boy(tony)"

Do you like to try again by using the above assumption
   ---? y
```

Session 6.4.2.1: The fourth part of the proving session of "boy(tony)"

The above session shows the system's suggestion, i.e if "boy(tony)" is true. By accepting the suggestion, the system will assert predicate "*subsidiary(boy(tony))*" into the database as explained in section 6.2.2.2. As a result of this, Prolog will backtrack to the step 4 of procedure QUEST (i,e the explanation is as in example 1 before).

In this case, procedure RETRY-SEARCH1.1 will be executed as only predicate *newsubsidiary* exists in the database. So, no reformulation of new question will be carried out, but the proving will be once again carried out, i.e the steps 6 and 7 of procedure QUEST will be executed. The following session shows the final part of proving "boy(tony)":

```
>>answer: Yes,
    boy(tony)
================================

PROVED:
  boy(tony) ?

yes
```

Session 6.4.2.2: The final (fifth) part of proving session of "boy(tony)"

In this example, no reformulation of the original question is performed. In other words, the same question is reproved again but this time a new knowledge (fact) has been asserted into the database. This example shows the second possible cause of failures as discussed in section 6.1.

6.4.3. Example 3

In this example, we will show how the system weakens the scope of the binding of the question's quantifier(s). The weakening process is performed when the difference between the reason clause and the mismatch KB clause contains a questioned Skolem function (denoted by "fq"). It should be noted here that it is not always the case to weaken the question's quantifiers, but it may also to universify the question's quantifiers, i.e changing from existential quantifier into universal quantifier. This case happens as a result of the negation of the question's clause before proving it. However, we will not show this case in this example as the process of universifying the quantifier is equivalent to the process of weakening the quantifier.

By assuming the present of Database 6.4.1, let us prove the unprovable question (for the sake of clarifying the system's work) as shown in the following session 6.4.3.1.:

```
?-pcquest.
!:  all(X,man(X)=>exists(Y,woman(Y)&loves(X,Y))).

NEXT QUESTION:
       all(_1,man(_1)=>exists(_2,woman(_2)&loves(_1,_2))

The translation of its negation:
man(fq(man0)).
[]:-woman(_2),loves(fq(man0),_2).
&&&&&&&&&&&&&&&&&&&&&&&&

>>>>>>>> can not prove man(fq(man0)) <<<<<
>>>>>>>> can not prove []:-woman(_2),loves(fq(man0),_2) <<<<<

>>answer: No, it cannot prove :
       all(_1,man(_1)=>exists(_2,woman(_2)&loves(_1,_2))
========================================
```

Session 6.4.3.1: The second part of the proving session

From the above session, a universal quantifier of the question, i.e X (or "_1"), have been changed to question's

Skolem function, i.e "fq(man0)". Now, let us see the second part of the proving session as shown in the following session (6.4.3.2):

```
OPTION 1:
The reason why the goal:
   " all(_1,man(_1)=>exists(_2,woman(_2,)&loves(_1,_2)) "
fails is due to the non-existence of the the following fact:

   "man(fq(man0))

However, we may able to prove the goal
after doing some corrections or additions

   --Do you like to continue ?s

" "man(fq(man0))" fails
   ===> "goal(all(_1,man(_1)=>exists(_2,woman(_2,)&loves(_1,_2))" fails

     --Do you like to continue ?y
```

Session 6.4.3.2: The second part of the proving session

One of the set of the reason clauses of the failed question is " ~man(fq(man0))" as shown in the above session. So, the rectification process continues by processing the reason clauses as explained in the example 1 before. The third part of the proving session will not be presented by the system as the difference between the reason clause and a mismatch KB clause involve a questioned Skolem function, i.e "fq(man0)". So the system will continue with the fourth part of the proving session as shown in the following session (6.4.3.3):

```
   If "all(_1,...)" is replaced with "exists(_1,...)"

Do you like to try again by using the above assumption
   ---? n

   the reason clause contains a Skolem function "fq"

Do you like to try again by using the above assumption
   ---? y
```

Session 6.4.3.3: The fourth part of the proving session

So the system will backtrack to the step 4 of procedure QUEST to reformulate new question as the user has agreed to weaken the universal quantifier binding "man(X)" to an existential quantifier. After the new question has been reformulated, then the system will backtrack again to the step 4 of procedure QUEST to restart again the proving process but with a new modified question and the process starts all over again as explained in the example 1 before.

```
RE-QUESTION:
   exists(_3,man(_3)&exists(_4,woman(_4)&loves(_3,_4)))

The translation of its negation:
[]:-man(_3),woman(_4),loves(_3,_4).
&&&&&&&&&&&&&&&&&&&&&&&&&

>>answer: Yes,
   exists(aizat,man(aizat)&exists(rosie,woman(rosie)&loves(aizat,rosie)))
==============================

PROVED:
   exists(aizat,man(aizat)&exists(rosie,woman(rosie)&loves(aizat,rosie)))

yes
```

Session 6.4.3.4: The final (fifth) part of the proving session

We can see from session 6.4.3.4, that the universal quantifier of the original question has been weakened to become an existential quantifier and furthermore the implication sign corresponding to the universal quantifier has also been changed to a conjunction sign (&). In other words, the old question "*all(X,man(X) =>...)*" is changed to a new question "*exists(X,man(X) &...)*". As shown in session 6.4.3.4, the new question has been successfully proved by the system.

6.5 Comments

In this chapter, we have discussed the fault rectification algorithm, the procedures to link up this algorithm and the fault detection algorithm (section 5.4) and in the last section (6.4), we have given three examples to show the ability of the system in detecting and rectifying fault. All the original questions of these examples are in the form of predicate calculus. It should be noted here that the system is able to diagnosis the question in the form of PC where the predicate is either in the form of ,say, "*man(X)*" or "*f(man,X)*" (see chapter 4).

Summarily, the system is able to rectify those reason clauses detected either by

[1] substituting all wrong references in the question's clause,

[2] weakening (universifying) the scope of the relevant quantifiers in the question's clause,

or [3] asserting the relevant reason clauses into the database as a subsidiary clauses (a new knowledge).

These rectifications are carried out on the assumption that the user make a wrong question and the database does not contain any faulty data. However, the user also can rectify any faulty clauses (facts) in the database. Those faulty facts can be spotted by the user during interacting with the system especially when asking why a substitution is suggested and also when printing the failure path of the failed goals (subgoal).

For example, suppose we would like to query about "*man(razak)*" from the database 6.4.1 (section 6.4). Unfortunately, the system cannot deduce or prove the particular question, i.e "*man(razak)*". And the user continues interacting with the system to find out the reasons or any rectification. In the third part of the proving session (as explained in section 6.4),which is as follows:

```
if " razak " of the question's clause " man(razak)" is substituted
   with "rzak"
                        ---Do you agree ? n

   " man(razak)" fails, but exists
       " man(rzak) "

...So,
   if " razak " of the question's clause " man(razak)" is substituted
   with "rzak"
                        ---Do you agree ? b
```

Session 6.5.1: The third part of proving session of "man(razak)"

The above session (6.5.1) shows the suggestion of substituting "*razak*" with "*rzak*" as exists "*man(rzak)*" in the database. We, as a user, could notice that we know the correct spelling of "*razak*", thus "*man(rzak)*" is wrong and should be corrected and not to rectify the question, in this case. Consequently we break out from the proving session and rectify the database contents. This method is still helpful although a bit manually. Even Shapiro's method needs the user to modify the faulty rules.

This method can also be extended to find and to rectify faulty rules by applying Shapiro's contradiction backtracking technique to the failure tree as shown in the second part of the session (see section 6.4) and also by assuming that the reason clauses are true (and thus the

proving succeeds). After detecting faulty rules or facts, the modification of the rules or facts can be made by using Prolog commands (such as retract, consult etc).

Shapiro's technique is used to detect and to modify any faulty rules or facts when the proving gives unexpectedly successful results. However, our techniques is used to detect and to rectify any query or fact when the proving ends up with unexpectedly unsuccessful results.

CHAPTER 7

A COMPLETE SYSTEM

7.1 Introduction

In chapter 4, we have discussed the problems in interfacing a subset of natural language processor to the theorem prover. Three methods of implementing the subset of natural language have been discussed. Those techniques are the tracing technique, the wording technique and the extra conditions of "nonvar(X)" and "var(X)". These techniques have been used to analyse English sentence into PC and also to synthesize them from the resulting PC.

In chapter 5 and 6, we have also discussed algorithms on how to detect and to rectify a faulty fact respectively and also the procedure to link both algorithms. We restrict ourselves in discussing both algorithms that the input question is in the form of PC However, the PC's form is in either, say, "man(X)" or "f(man,X)". The latter was adopted in translating an english sentence into PC (chapter 4). The Prolog-based theorem prover have also been modified to translate both PC forms into Horn clauses.

In the following section (7.2), we will discuss how the three techniques discussed in chapter 4 can be incorporated with the fault detecting and rectifying algorithms. In other words, we would like to build a complete system such that it can process both types of input, i.e an English (a natural language) sentence or a PC. We will show some examples how the system works with an English sentence as input in section 7.3.

7.2 Interfacing with the English grammar

In this section, we will discuss how the three techniques discussed in chapter 4, i.e the tracing, the wording and the extra conditioning techniques, will be incorporated into the detecting and the rectifying algorithms (as discussed in chapter 5 and 6 respectively).

In chapter 6, we assume two possible types of fault, i.e

[1]. The wrong references in the query.

[2]. The non-existence of the facts in the database.

After detecting those faults, we have taken one or more of the following rectification steps depending the nature of the fault detected:

(a) by substituting an atom in the question clause, for example, by substituting "*man*" of "*man(rosie)*" with "*woman*" to become *woman(rosie)*".

(b) by weakening (or universifying) the relevant quantifiers of the question clause, for example, by weakening a universal quantifier binding "*man(X)*" of "*all(X,man(X)=>loves(X,god))*" into an existential quantifier, thus "*exists(X,man(X)&loves(X,god))*" became a new question.

(c) by asserting a new knowledge clause into the database when the proposals as suggested in (a) or (b) are rejected and also when no mismatch clauses can be found. For example, "*man(razak)*" is asserted into the database as a subsidiary clause.

In both cases (a) and (b) and any combination of (a), (b) or (c), we will have to reformulate a new question. On the

other hand, in the case (c) alone (without cases (b) and (a)), the old question will not be changed but instead a reproving is carried out again.

Thus when a new question (in the PC form) is reformulated, we also need to rephrase a new English (a natural language) question corresponding to the new PC question. This is where the incorporation of the three techniques discussed in chapter 4 will be discussed.

7.2.1. The tracing technique

By using this technique, as discussed in chapter 4, we will be at ease if the analysing and synthesizing process deals in generating the same original English question (sentence), for example, the question sentence "Tony is kind?", says, will give the answer sentence as "Yes, Tony is kind".

However, if the need of substituting and/or weakening processes are required, this technique will be very difficult to implement. We would not have any difficulty in substituting and/or weakening processes of the question in PC form, but the new PC question will be very hard to be synthesized back into English sentence again, as the tracing variable recorded during the process of analysing is no use at all due to the rectification process of cases (a) and/or (b) have taken place. Thus the old tracing variable also needs to be modified according the rectification processes of (a) and/or (b) which were taken upon the old question.

All the grammar rules numbers and the wording itself must be modified in the tracing variable. Furthermore, the most

difficult modification is when the rectification of the case (b) is carried out, i.e the whole series of grammar rules used may be different from the grammar rule used as explicitly shown in the original tracing variable. In foreseeing this difficulty, this method has not been incorporated in the fault detection and rectification algorithms.

7.2.2 The wording technique

After a new PC question is reformulated, then if using the wording technique, the wording database which contains all the words of the question or the sentence, has to be modified according to the process taken upon the old question, i.e when cases (b) and/or (c) are carried out. For example, see the following session:

```
?-phrase.
I: Tony is kind?

NEXT SENTENCE:
       Tony is kind?
===============>

The listing of "word_used":
       word_used(Tony).
       word_used(is).
       word_used(kind).
              .
              .
=========>
No, it is false that Tony is kind.

OPTION 1:
The reason why the goal:
       "exists(Tony,proper_noun(Tony),f(kind,Tony))"
fails is due to the non-existence of the the following fact:

       f(kind,Tony)

However, we may able to prove the goal
after doing some corrections or additions

    --Do you like to continue ?y
```

Session 7.2.2.1: 1st and 2nd part of proving session of "Tony is kind?"

The above session shows that the question "Tony is kind?" fails due the non-existence of "*f(kind,Tony)*" in the database. If we see the next part of the proving session which is shown as follows:

```
if " Tony " of the question's clause "f(kind,Tony)" is substituted
   with "John"
                        ---Do you agree ? w

   " f(kind,Tony)" fails, but exists
       " f(kind,John)"

...So,
   if " Tony " of the question's clause " f(kind,Tony)" is substituted
   with "John"
                        ---Do you agree ? y
```

Session 7.2.2.2: The 3rd part of proving session of "Tony is kind?"

So, after accepting the suggested substitution, the system will reformulate the new question in the PC form which become "*exists(John,proper_noun(John),f(kind,John))*" after replacing "Tony" with "John". However, without modifying the wording database, the new corresponding question "John is kind?" will not be generated as "word_used(John)" does not exist in the database In order to generate the new corresponding sentence (question), we need also to assert "word_used(John)" in the wording database to replace "word_used(Tony)". And so, the proving session will continue until the user satisfies with the response from the system or no other possible solution.

From the above example, the substitution of type (a) can be easily be implemented. However, this is not so in case (b) where the quantifier of the question's clause (sentence) is involved. For example, suppose the question is "every man loves a woman" which will be translated into PC form as follows:

```
all(X,indefinite(X),f(man,X)=>exists(Y,indefinite(Y),f(woman,Y)&f(loves,X,Y)))
```

This PC representation is equivalent with the example 3 of section 6.4.3. And the wording database is as follows:

```
The listing of "word_used":
        word_used(every).
        word_used(man).
        word_used(loves).
        word_used(a).
        word_used(woman).
```

The new question reformulated after accepting the suggested rectification, i.e by weakening the universal quantifier binding "*f(man,X)*", is became:

```
exists(X,indefinite(X),f(man,X)&exists(Y,indefinite(Y),f(woman,Y)&f(loves,X,Y)))
```

The above new PC question is equivalent to "a man loves a woman". Then, by using the wording technique, the original wording database should be modified in order to generate the equivalent new PC. Thus, "*word_used(every)*" of the wording database should be replaced with *word_used(a)*". In this case, it is not very difficult to implement the modification of the wording database.

However, the question "all men love a woman" should give the same PC question but, of course, with a different wording database:

```
The listing of "word_used":
        word_used(all).
        word_used(men).
        word_used(love).
        word_used(a).
        word_used(woman).
```

Then the new reformulated PC question in English sentence will be hard to generate, as we need to replace three wordings, i.e "all", "men", and "love" with "a", "man" and "loves" respectively. This is one of the difficulties in implementing this technique when we would like to synthesize from the given PC into the corresponding English sentence.

Although we can write a program to extract all the relevant words of the sentence from the PC itself, but the relevant extracted words do not indicate their singularity or plurality. For example, the word "man" which is extracted from "*all(X,indefinite(X),f(man,X)=>...)*", does not indicate whether its original sentence form is "every man ..." or "all men ..." or other forms. Furthermore, the wording database does not also show the relationships between those words. For example, the wording databases for sentences "every man loves a woman" and "a man loves every woman" are the same.

In viewing these difficulties, the wording technique has not been incorporated into the fault detection and rectification algorithms.

7.2.3 The conditioning technique: "var(X)" and "nonvar(X)"

This technique is quite similar to the wording technique except that this conditioning technique does not keep a record of all wordings of the English sentence. It also puts more information about each word of the sentence into the corresponding PC itself as discussed in section 4.4.3.

In incorporating this conditioning technique into the fault detection and rectification algorithms, we will face the same problems as discussed in the last two sections, i.e sections 7.2.1 and 7.2.2.

We would not have any problem if the rectification of type (a) and/or (c) are involved. However, if the rectification of type (b) has to be carried out, then we would not have as

many problems as discussed in the last two sections (7.2.1 and 7.2.2) due to the new and old PC itself containing more information about the sentences they represented. For instance, referring back to the question "every man loves a woman" where its PC representation is as follows:

```
all(X,det(every),f(man,X)=>exists(Y,det(a),f(woman,Y)&f(loves,X,Y)))
```

and the proving session of the above session is exactly equivalent with the example 3 of section 6.4.3 except the above PC use the form of "*f(man,X)*" whereas the example 3 used the form of "*man(X)*". As shown in session 6.4.3.3 of section 6.4.3, i.e the fourth session of the proving session, the system has suggested weakening the universal quantifier binding "*man(X)*" (or in this case, "*f(man,X)*") to become an existential quantifier. Thus after weakening the universal quantifier, the new PC question is as follows:

```
exists(X,det(every),f(man,X)&exists(Y,det(a),f(woman,Y)&f(loves,X,Y)))
```

However, the system could not translate the new PC question into an English question because the new PC contains term "*det(every)*". This term, "*det(every)*", should also be changed into an appropriate term to correspond an existential quantifier such as "*det(a)*". The program below is aimed to accomplish this purpose:

```
/* to weaken (or universify) the corresponded determiners */
weaken_det(det(X),det(Y)):-change_det(X,Y).

/* to change from a universal determiner to an existential determiner */
change_det(every,a).
change_det(all,a).
change_det(everybody,somebody).

/* to change from an existential determiner to a universal determiner */
change_det(a,all).
change_det(some,every).
change_det(somebody,everybody).
```

Program 7.2.3.1: Changing the determiners

The program 7.2.3.1 which shows only some of the changing determiners can be extended to include more relationship between the universal and existential determiners. So by applying the above program, the new PC question becomes:

```
exists(X,det(a),f(man,X)&exists(Y,det(a),f(woman,Y)&f(loves,X,Y)))
```

which corresponds to the English question "a man loves a woman". In general, during weakening the quantifier, the PC of the form of "*all(X,det(U),f(man,X)=>...)*" is changed into "*exists(X,det(E),f(man,X)&...)*" where "*all*", "*det(U)*" and "*=>*" are changed into "*exists*", "*det(E)*" and "*&*" respectively Furthermore, the English question is changed from "*every man loves ...*" into "*a man loves ...*".

7.2.4 Comments on the incorporation of the three techniques

In the last three sections (7.2.1, 7.2.2 and 7.2.3), we have discussed the main problem in synthesizing an equivalent English sentence from the new reformulated question (in PC). We have also discussed why the tracing and wording techniques have not been incorporated into the fault detection and rectification algorithms. Thus, in foreseeing those difficulties, the conditioning technique has been incorporated into the fault detection and rectification algorithms.

Other changes in the fault detection and rectification algorithms are cosmetic only and are not very difficult to implement regardless of the techniques we choose, for example, to change the remark of "if 'all(_1,..)' is replaced with 'exists(_1,...)'" into says, "if 'every man' is replaced with 'a man'.

7.3 Examples

We have already described in section 7.2.3 how the English grammar is incorporated with the fault detection and rectification algorithms. In this section we will show some examples of the complete system. It should be noted that the system is still able to accept the inputs either in the form of English or PC. Furthermore, the input PC can also be either in the form of says, "*man(X)*" or "*f(man,X)*".

7.3.1 Example 1

We can start from scratch, i.e an empty database. Let see the following session:

```
?-phrase.
I: John loves Mary?
                .
                :
NEXT QUESTION:
    "exists(John,proper_noun(John),exists(Mary,proper_noun(Mary),f(loves,John,Mary)))".
                .
                :
=========>
No, it is false that John loves Mary.

OPTION 1:
The reason why the goal:
    "exists(John,proper_noun(John),exists(Mary,proper_noun(Mary),f(loves,John,Mary)))"
fails is due to the non-existence of the the following fact:

    f(loves,John,Mary)

However, we may able to prove the goal
after doing some corrections or additions

     --Do you like to continue ?n

Do you like to assert
    "John loves Mary"
as a fact in the database? y

yes

?- listing(knowledge).
knowledge(f(loves,'John','Mary')).

yes
```

Session 7.3.1.1: Proving "John loves Mary"

The above session shows how we start the session from scratch, i.e by asking "John loves Mary?" and, of course, the proving fails as the database contains nothing. At the end of the proving session, after rejecting to rectify it, the system will ask whether we would like to assert it as a fact in the database. The database then will contains the first fact as shown in the listing of KB clauses in the above session after we accept to assert it into the database.

7.3.2 Example 2

Let us continue the interaction (of example 1 above) with the system by asking a question "every man loves every woman" which is shown in the following session:

```
?-phrase.
|: every man loves every woman?

NEXT SENTENCE:
   every man loves every woman?
===============>

NEXT QUESTION:
     all(_1,det(every),f(man,_1)=>all(_2,det(every),f(woman,_2)=>f(loves,_1,_2)))
                    .
                    :
=========>
No, it is false that every man loves every woman.

OPTION 1:
The reason why the goal:
     "all(_1,det(every),f(man,_1)=>all(_2,det(every),f(woman,_2)=>f(loves,_1,_2)))"
fails is due to the non-existence of the the following fact:

   f(loves,fq(man0),fq(woman0))

However, we may able to prove the goal
after doing some corrections or additions

     --Do you like to continue ?y

if "every man" is replaced with "a man"
if "every woman" is replaced with "a woman"

Do you like to try again by using the above assumption
   ---? y
```

Session 7.3.2.1: Proving "every man loves every woman"

After accepting the replacement of the determiners (the weakening of the universal quantifiers binding "*f(man,_1)*" and "*f(woman,_2)*"), the system will reformulate a new PC question and then it (the system) will synthesize into a new English question from the new PC question. Thus the system will continue proving the new question as shown in the following session:

```
RE-PHRASE:
   a man loves a woman.
==================>

RE-QUESTION:
   exists(_3,det(a),f(man,_3)&exists(_4,det(a),f(woman,_4)&f(loves,_3,_4)))
                  .
                  :
=========>
No, it is false that a man loves a woman.

OPTION 1:
The reason why the goal:
    "exists(John,det(a),f(man,John)&exists(Mary,det(a),f(woman,Mary)&f(loves,John,Mary)))"
fails is due to the non-existence of the the following fact:

       f(man,John).
       f(woman,Mary).

However, we may able to prove the goal
after doing some corrections or additions

    --Do you like to continue ?y

If the following clauses are true:

       f(man,John).
       f(woman,Mary).

Do you like to try again by using the above assumption
   ---? y
```

Session 7.3.2.2: Proving "every man loves every woman"

As shown in the above session (7.3.2.2), the new question is became "a man loves a woman?". However the new question still fails due to non-existence of clauses "*f(man,John)*" and "*f(man,Mary)*" in the database. So after asserting both clauses into the database, the system will succeed in proving the new question.

7.3.3 Example 3

The following example is based on the following premises:

(1) No wombat who lives in Twycrosszoo is happy.

(2) Any animal who meets kind people is happy.

(3) People who visit Twycrosszoo is kind.

(4) Animals who lives in Twycrosszoo meets people who visits Twycrosszoo.

(5) Wombats are animals.

(6) Somebody visits Twycrosszoo.

In this case, Twycrosszoo is considered as a proper noun. For the sake of space, we will show the PC equivalence of some of the premises only as follows.

```
?-phrase.
|: no wombat who lives in Twycrosszoo is happy.
==============>

       .
       :
The translated clauses:
[]:-f(wombat,_1),f(lives,_1),f(in,_1,Twycrosszoo),f(happy,_1).

yes.

?-phrase.
|: any animal who meets kind people is happy.
==============>

       .
       :
The translated clauses:
f(happy,_1):-f(animal,_1),meets(_1,_2),f(person,_2),f(kind,_2).

yes.

?-phrase.
|: Animals who lives in Twycrosszoo meets people who visits Twycrosszoo.

==============>

       .
       :
The translated clauses:
f(meets,_1,_2):-f(animal,_1),f(lives,_1),f(in,_1,Twycrosszoo),f(person,_2),f(visits,_2,Twycrosszoo).

yes.
```

Session 7.3.3.1: Asserting premises into the database.

After asserting the above premises into the database, the following session (7.3.3.2) shows a proving session of a

question "No wombat lives in Londonzoo?" which is based on the above premises.

```
?-phrase.
I: no wombat who lives in Londonzoo?
        .
        :
NEXT QUESTION:
    "exists(_1,det(no),f(woman,_1)&f(lives,_1)&f(in,_1,Londonzoo)
        .
        :
==========>
No, it is false that no wombat who lives in Londonzoo.

OPTION 1:
The reason why the goal:
    "exists(_1,det(no),f(wombat,_1)&f(lives,_1)&f(in,_1,Londonzoo)"
fails is due to the non-existence of the the following fact:

   f(in,fq(wombat0),Twycrosszoo))

However, we may able to prove the goal
after doing some corrections or additions

    --Do you like to continue ?y

if " Londonzoo " of the question's clause " f(in,fq(wombat0),Londonzoo)) " is substituted
   with "Twycrosszoo"
                        ---Do you agree ? y
                .
                :
Do you like to try again by using the above assumption
   ---? y

RE-PHRASE:
   no wombat lives in Twycrosszoo.
==================>

RE-QUESTION:
    "exists(_1,det(no),f(wombat,_1)&f(lives,_1)&f(in,_1,Twycrosszoo)"
                .
                :
==========>
Yes, it is true that no wombat who lives in Twycrosszoo
                .
                :
```

Session 7.3.3.2: Proving "no wombat lives in Londonzoo ?"

The session above (7.3.3.2) shows a proving session of the original question "no wombat lives in Londonzoo" and after rectification process of type (a) has taken placed, the new question is became "no wombat lives in Twycrosszoo" and is successfully proved by the system.

7.4 Comments

In the last section (7.3) we have given some examples to show the ability of the system which can accept an English sentence and also PC(section 6.4) as input. The answer will accordingly depend on the type of input.

Example 1 (section 7.3.1) shows that we can start the database from scratch, i.e from nothing and build up the database. In this sense, we can do a test for each fact or rule before each of them is asserted in the database. Thus any provable fact or rule can be filtered before asserting them into the database.

Example 2 (section 7.3.2) shows how rectification processes of types (b) and (c) are carried out. At first the weakening of the quantifiers is carried out and is followed by asserting the subsidiary clauses after the new question after weakening process still fails However this example show only one reformulation and followed by another reproving process after new facts are known or are asserted into the database.

Example 3 (section 7.3.3) shows a rectification process of type (a) where a wrong reference is asked, i.e "Londonzoo" is asked instead of "Twycrosszoo". Although the non-existent fact is "*f(lives,fq(wombat0),Twycrosszoo)*" which does not contain "Londonzoo", but as exists a query clause:

```
query(f(lives,fq(wombat0),Londonzoo))
```

in the database, thus a suggestion is proposed to substitute "Londonzoo" which is a term of question clause with "Twycrosszoo" which is a term resulting from the

instantiation of the KB clauses. In other words, the assumption that the query is always wrong in the case of mismatching, is adopted.

There is not much modification made on the fault detection and rectification algorithms as much of the work done is to distinguish the types of input and output as explained in chapter 4.

More time should be devoted if we would like to incorporate the tracing and wording techniques into the fault detection and rectification algorithms to overcome the difficulties as explained in sections 7.2.1 and 7.2.2. However the conditioning technique is also able to perform the task as we already add more information into the PC in order to guide the synthesizing process.

CHAPTER 8

CONCLUSION

8.1 Discussion and comments

The problem of software reliability is getting a great deal of attention. One aspect of software reliability is program debugging. Program debugging could also be considered as a rule learning program where the user learn to write rules which eventually will become a program. In the process of learning to write the rules, debugging is carried out to make the rules as consistent as possible. Most of the rules are written in clausal form.

In this thesis, a mechanical theorem prover program has been written (see chapter 3) in Prolog to take advantage of Prolog as a theorem prover itself. The theorem prover which is a Prolog-based theorem prover has been written using two search strategies, i.e a depth-first strategy and a breadth-first strategy. We have also discussed the advantages and disadvantages of both methods. After weighing all the pros and cons (as discussed section 3.3.3.4), the depth-first method has been adopted for further work in this thesis Furthermore, Prolog also adopts a depth-first strategy. The Prolog-based theorem prover which has been incorporated with the looping test as its control feature is capable of handling four classes of questions as classified by Chang and Lee[1973]. The looping tests enable the prover to control the selection of the firing rules and thus the user is left with the problem of writing the rules as suggested by Kowalski[1979] to make:

"Logic + Control = Algorithm"

The Prolog-based theorem prover is then incorporated with a subset of English grammar (chapter 4) based on a Definite

Clause Grammars (DCGS) which was proposed by Pereira and Warren [1980]. The English grammar written is based on the grammar as used in Hinde[1983] and [1986] for his Fuzzy Prolog. Some modifications have been made in order to make the grammar reversible, i.e it can be used for analysing as well for synthesizing an English sentence (question). So the grammar is used to translate an English question (and sentence) into a corresponding questioned PC and here the Prolog-based theorem prover is applied to answer the questioned PC. The answered PC is then passed back to the grammar to retranslate it into a corresponding answered English sentence. For this purpose, three techniques have been investigated to make the grammar produce a sensible English sentence (answer). Those techniques are the tracing, the wording and the conditioning techniques. All three techniques are capable of synthesizing a sensible English sentence from the answered PC. But in the process of doing so, the PC representation has been changed to overcome some difficulties and to prevent an unsensible sentence.

A Prolog-based theorem prover can also be viewed as a question-answering system which can accept the question and give the answer in PC as its natural ability. But it also can accept the input in the form of clauses (such as Horn clauses) and in the form of a subset of natural language (English) as a result of incorporating the English grammar (Chapter 4).

In a question-answering system, we sometimes (or always) get an unexpectedly unsuccessful answer due to whether we ask a wrong question or the database does not contain any

information about what we ask. The wrong question may contain wrong references which do not match with the information contained in the database, or it (the question) is too general to be answered correctly.

The theorem prover is then modified to detect the faulty fact or non-existent clauses (Chapter 5). The modification is done mainly by recording the failed goals (or subgoals) during the process of proving of the main question (goal). It records all non-existent clauses at different levels during backtracking.

After detecting all the non-existent clauses, an algorithm to rectify those faults is written and explained as in Chapter 6. The assumptions made are that the non-existent clauses are as a result of wrong references in the query (question) or the lack of knowledge (information) in the database. Based on these assumptions, the algorithm will rectify the fault by suggesting the substitution of the wrong reference term or by asserting the new information into the database.

All the inputs of the algorithms discussed in chapter 5 and 6 are in the form of PC. The English grammar discussed in chapter 4 is then incorporated into both algorithms to make it as a complete question-answering system with the capability of fault detection and rectification. In foreseeing the difficulties discussed in chapter 7, the conditioning technique is adopted for the complete system.

8.2 Further work

In the last section, we have discussed what we have achieved or done in this thesis. A lot of works can be extended in future.

The English grammar used in the system is only a subset of English grammar. The grammar can be extended to include time reference (such as past tense, future tense etc), noun phrases (such as "it", possessive adjectives, etc). The Skolem functions used to denote noun phrases could be modified to refer the extended noun phrase included in the grammar for the purpose of theorem proving

Apart from the English grammar, other languages could be incorporated to make a multi-lingual question-answering system. Mawdsley[1984], Baker[1985] and Kok[1986] have designed multi-lingual machine translation of French, German and Malay languages respectively and English. These works could be incorporated into the system by modifying their PC representation to suit the question-answering system.

Fuzziness properties could also be included in the system, either at PC level or at a natural language level. If it is incorporated at PC level, the PC representation needs to be changed accordingly. If it is incorporated at a natural language level, the grammar has to be written again in order to capture the meaning of the fuzziness word in the sentence.

The question-answering system can also be used in writing a program direct from a natural language with suitable grammar or direct from the PC itself. As the system already incorporates loop checking, the tasks left are to detect faulty rules or facts. As, the system can detect faulty facts then it remains to incorporate detection of faulty rules. Here Shapiro's technique could be incorporated into the system. Thus the system would be able to cater for unexpectedly successful proving (Shapiro's technique) and unexpectedly unsuccessful proving (our system). Shapiro's technique can be fitted into the proving tree of the solution which shows the firing of rules or facts and ground oracles.

As Damerau[1964] indicates that 80% of typing errors are caused by transposition of two adjacent letters, one extra letter, one missing letter, or one wrong letter These may also apply to predicates as their arguments can be considered as letters in the Damerau's finding. The matching process between the reason clauses and the KB clauses could be extented to include these type of errors In other words, the matching could be in the form of spelling checkers, i.e in this case, the argument checkers. This method may be useful in the either form of inputs, i.e PC or English sentence. In PC's form, it may include to check, says, "*f(sort,a,b,c)*" with "*f(sort,b,a,c)*" where the second and third arguments are transposed with each other. Perharps it could be used in the context spelling checkers.

8.3 Conclusions

In this thesis, we have designed a question-answering system with the capability of detecting and rectifying faults which occur when we get unexpectedly unsuccessful answers. Furthermore the system is designed in such a way that it can accept both input either in the form of PC or a subset of English sentence. So we can add any language grammar as long as its PC representation is the same as the standard PC input.

The technique of detecting faults discussed is a complement of Shapiro's technique which detects a faulty rule when the answer is unexpectedly successful. In our rectification technique, we introduce matching between arguments with the same predicate or between predicates with the same arguments. In other words, both predicates must have the same number of arguments. This matching could be incorporated with techniques used in spelling checkers. Perhaps by combining both techniques of detecting fault (Shapiro's and ourselves) and incorporating the spelling checking technique, we will have a better system in future.

REFERENCES

Adam,A. and Laurent,J-P. [1980], "Automatic Diagnostics of Semantic errors", *Proceedings of the AISB-80 conference on Artificial Intelligence*, Amsterdam ,1-4th July, 1980, pp(ADAM-1)-(ADAM-10).

Anderson,J. and Bower,G. [1973], *Human Associative Memory* Winston, Washington, D.C. 1973.

Andrews,P.B. [1981], "Theorem proving via general matings", *Journal of the ACM*, 28(2), pp193-214.

Angus,J.E., Bowen,J.B. and VanDenberg,S.J. [1983], *RADC (Rome Air Development Centre)- TR-83-207*, Vol 1 (of two), August 1983.

Baker,W. [1985], *Refinements to an existing interlingual machine Translation system*, M. Sc Dissertation, Dept. of Computer Studies, Loughborough University of Technology, 1985.

Baldwin,J.F. [1981], "Fuzzy logic and Fuzzy reasoning", in *Fuzzy Reasoning and its Applications*, Mamdani,E.H and Gaines,B.R. (eds), Academic Press, pp133-148, 1981.

Balzer,R. [1975], "Automatic Programming", *Technical Report 1, USC/ISI*, September 1972.

Basili,V.R. and Perricone,B.T. [1984], "Software errors and complexity: an empirical investigation", *Communications of the ACM*, Vol 27(1), January 1984, pp42-52.

Bennett,P., Johnson,R., McNaught,J., Pugh,J., Somers,H. and Sager,J.C. [1986], *Multilingual Aspects of Information Technology*. Gower Publishing Co Ltd, England. 1986.

Bibel,W. [1976], "A syntactic connection between proof procedures and refutation procedures", *Second Conference on Automated Deduction*, Oberwolfach, West Germany.

Bibel,W. [1983], "Matings in matrices", *Communication of the ACM*, 26(11), pp844-852

Bledsoe,W.W. [1977], "Non-resolution theorem proving", *Artificial Intelligence*, 9(1), pp1-35, 1977.

Bobrow,D.G. and Winograd,T. [1977a], "An overview of KRL, s knowledge representation language" *Cognitive Science*, 1(1), pp3-46, 1977.

Bobrow,D.G. and Winograd,T. [1977b], "Experience with KRL-0: one cycle of a knowledge representation language" *Proceedings of the 5th International Joint Conference on Artificial Intelligence* pp 213-222, 1977.

Bobrow,D.G. and Winograd,T. [1979], "KRL: another perspective", in *Cognitive Science*, 3(1), pp29-42.

Bobrow et al. [1977], "GUS, A frame driven dialog system" *Artificial Intellignence*, 8(2), pp 155-173

Boehm,B.W. [1976], "Software engineering", *IEEE Trans. Computi.*, Vol c=25, pp1226-1241, 1976.

Bourne,C.P. [1977], "Frequency and impact of spelling errors in bibliographic data bases", *Inform. Processing and Mgmt.*, Vol 13, No. 1, 1977, pp1-12.

Bowen,K.A. [1982], "Programming with Full First-Order Logic, *Machine Intelligence 10*, 1982, pp421-440.

Boyer,R.S. [1971], *Locking: a restriction of resolution*, PhD. thesis, University of Texas.

Brachman,R.J. and Smith,B.C. [1980], "SIGART Newslatter 70, Special Issued on Knowledge Representation", 1980.

Bradzil,P. [1981], *A model for error detection and correction*, Ph.D thesis, University of Edinburgh, 1981.

Bratko,I. [1986], *Prolog Programming for Artificial Intelligence*, Addision Wesley, Great Britian, 1986.

Brooks,F.P. [1975], *The Mythical Man-Month*, Addison-Wesley, Reading ,Ma, 1975.

Bundy,A. [1983], *The Computer Modelling of mathematical Reasoning*, Academic Press, London, 1983.

Bundy,A., Sharpe,B., Uschold,M. and Harding,N. (eds) [1983], "Intelligent Front End", *Intelligent Front End Workshop Report No. 1*, Consener's House, abingdon, England, 26-27th Sept. 1983.

Bundy,A., Silver,B. and Plummer,D. [1985], "An analytial comparison of some rule-learning programs", *Artificial Intelligence*, Vol 27, 1985, pp137-181

Campbell,J.A. (ed) [1984], *Implementation of Prolog*, Ellis Horwood, England, 1984.

Chang,C.L [1970], "The unit proof and the input proof in theorem proving" *Journal of ACM*, 17, pp697-707.

Chang,C.L and Lee,R.C. [1973], *Symbolic Logic and Mechanical Theorem Proving* Academic Press, London, 1973.

Charniak,E. and McDermott,D. [1985], *Introduction to Artificial Intelligence*, Addison Wesley, 1985.

Church,A.L. [1940], "A formulation of the simple theory of types" in *Symbolic Logic*, 5(1), pp56-68, 1940.

Clark,K.L. [1978], "Negation as Failure", in *Logic and Databases*, Gallaire,H and Minker,J. (eds), Plenum Press, New York, 1978, pp293-322

Clark,K.L and Gregory,S. [1981], "A relational Language for Parallel Programming", *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, 1981, pp 171-178.

Clark,K.L and Gregory,S. [1983], "PARLOG: A Parallel Logic Programming Language", *Research Report DOC 83/5*, Dept. of Computing, Imperial College, 1983.

Clark,K.L. and Tarnlund,S.-A. (eds) [1982], *Logic Programming*, Academic Press, London, 1982.

Clifford,J. and Warren,D.S. [1983], "Formal semantics for time in database". *ACM TODS* 8(2), pp214-254.

Clocksin,W.F. and Mellish,C.S. [1981], *Programming in Prolog*. Springer Verlag, 1981.

Colmerauer,A., Kanoui,H., Rousel,P. and Pasero,R. [1973], "Un systeme de communication homme-machine en Francais", *Research Report, Artificial Intelligence Group*, Uni. Of Aix-Marseille, Luminy, France, 1973.

Cullingford,R. [1981], "SAM" in *Inside Computer Understanding* Schank,R.C and Riesbeck,C.K. (Eds), Erlbaum, Hillsdale, N.J., 1981.

Damerau,F.J. [1964], "A technique for computer detection and correction of spelling errors", *Comm. of the ACM, 7(3)*, March 1964, pp171-176.

Davis,M. and Putnam,H. [1960], " A computing Procedure for Quantification Theory" *Journal of the ACM*, 7(3), 1965, pp201-215.

Downs,T. [1985], "An approach to the modelling of software testing with some applications", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, April 1985, pp375-386.

Dowsing,R.D., Rayward-Smith,V.J. and Walter,C.D. [1986] *A First Course in Formal Logic and its Applications in Computer Science*, Blackwell Scientific Publications, Great Britian.

Doyle,J. [1979], "A Glimpse of Truth Maintenance" in *Artificial Intelligence: An MIT Perspective, Vol 1* MIT Press, Cambridege,Mass, 1979.

Doyle,J. [1982], "A Truth Maintenance System" *Artificial Intelligence* Vol 12, No 3. 1982.

Enderton,H.B. [1972], *A Mathematical Introduction to Logic* Academic Press, New York, 1972.

Findler,N.V.(eds) [1979], *Associative Networks - The representation and Use of Knowledge in Computers*, Academic Press, New York

Frost,R.A. [1986], *Introduction to knowledge base system* Collins, Great Britian,1986.

Gallaire,H. and Minker,J (eds) [1978], *Logic and Databases*, Plenum Press, New York, 1978.

Gallaire,H. and Minker,J (eds) [1981], *Advances in Database Theory*, vol 1, Plenum Press, New York, 1981.

Gallaire,H., Minker,J. and Nicolas,J-M. [1984], "Logic and Databases: A Deductive Approach", *Computing Surveys*, Vol 16, No. 2, June 1984.

Genesereth,M,R. and Ginsberg,M.L.[1985], "Logic Programming" *Communication of the ACM*, 28(9), pp933-941, Sept. 1985.

Genesereth,M.R., Greiner,R. and Smith,D.E.[1983], "MRS - a meta-level representation system", *HPP-83-27* ,Heurestic Programming Project, Standford University, Calif., 1983

Goodenough,J.B. and Gerhart,S.L. [1975], "Towards a theory of test data selection", *IEEE Transactions on Software Engineering*, Vol SE-1, June 1975, pp156-173.

Gilmore,P.C [1960], "A proof method for quantification theory: Its justification and realization", *IBM Journal of Research Development*, vol 4, 1960, pp28-35.

Gray,P. [1984], *Logic, Algebra and Databases*, Ellis Horwood, Great Britian.

Green,C. [1969], "Applications of Theorem Proving to Problem Solving", *International Joint Conferences on AI*, Walker,D.E. and Norton,L.M.(eds), Washington, 1969, pp219-239

Gries,D. [1981], *The Science of Programming*, Springer-Verlag, New York and Berlin, 1981.

Hanson,A., Haridi,S. and Tarnlund,S-A. [1982], "Properties of a lovic Programming Language", in *Logic Programming*, Clark,K.L. and Tarnlund,S-A. (eds), Academic Press, 1982, pp267-280

Haridi,S. and Sahlin,D. [1983], "Evaluation of Logic Programs Based on Natural Deduction", *TRITA-CS-8305*, Royal Institute of Technology, Sweden, 1983.

Hayes,P.J. [1973], "Computation and deduction", *Proceedings MFCS Conference*, Czechoslovakian Academy Press, 1973.

Hendrix,G.G. [1975], "Expanding the utility of semantic networks through partitioning" in *Proceedings of the Fourth IJCAI* Tiblis.

Hendrix,G.G. [1977],"Human Engineering for applied natural language processing", in *Proceedings of the Fifth IJCAI*, MIT, Cambridge, Mass.

Hendrix,G.G. [1979], "Encoding knowledge in partitioned networks" in *Associative Networks - The representation and Use of Knowledge in Computers*, Findler,N.V.(eds), Academic Press, New York, 1979.

Herbrand,J. [1930], "Researches in the Theory of Demonstration", in *From Fredge to Godel. A source book in Mathematical Logic, 1879-1931*, van Heijenoort,J. (eds), Harvard University Press, Mass, 1967, pp525-581.

Hill,R. [1974], "LUSH resolution and its completeness" *DCS memo No 78*. School of Artificial Intelligence, Edinburgh University.

Hinde,C.J. [1983], *Fuzzy Prolog*. Computer Studies Internal Report No. 199, Dept of Computer Studies, Loughborough University of Technology, September 1983

Hinde,C.J. [1984], *An application and use of higher order fuzzy predicates* Symposium on Fuzzy Inference, Cambridge 1984.

Hinde,C.J. [1986], "Fuzzy Prolog", *International Journal of Man-Machine Studies*, 24, pp569-595, 1986.

Hinde,C.J. and Mawdsley,A. [1984], *An Interlingual English to French Machine Translation System* Computer Studies Internal Report No. 218, Dept. of Computer Studies, Loughborough University of Technology, Dec 1984.

Hogger,C.J. [1984], *Introduction to Logic Programming* Academic Press, London.

Horn,A. [1951], "On sentences which are True of direct Unions of Algebras" *Journalof Symbolic Logic*, 16, pp14-21.

ISIS systems *Micro-Expert Reference Manual*

Jelinski,Z. and Moranda,P. [1972], "Software Reliability research" *Conference , Statsitical computer Performance Evaluation*, Academic Press, NewYork and London, 1972, pp465-484.

Kok,Y.P. [1986], *Implementation of Malay to an existing interlingual Machine Translation System* Final Year Report, Dept. Of Computer Studies, Loughborough Unversity of Technology, 1986

Kowalski,R.A. [1974], "Predicate Logic as Programming Language", *Proceding of IFIP 74*, North-Holland Publishing Co., Amsterdam, pp569-574.

Kowalski,R.A. [1979], *Logic for Problem Solving*, North-Holland, New York, 1979.

Kowalski,R.A. [1979a], "Algorithm = Logic + Control", *Communication of the ACM*, 22(7), pp424-436

Kowalski,R.A. and Kuehner,D. [1971], "Linear resolution with selection function", *Artificial Intelligence*, vol 2, pp227-260.

Lebowitz,M. [1980], "language and memory: generalization as a part of understanding" in *Proceedings of AAAI 80*, Standford University, California

Lenat,D.B. [1982], "AM: an artificial intelligence approach to discovery in mathematics as heuristic search" in *Knowledge Based Systems in Artificial Intelligence*, Davis,R. and Lenat,D.B., McGraw Hill, New York.

Littlewood,B. [1979], "How to measure software reliability and how not to", *IEEE Transaction on Reliability*, Vol r-28, No. 2, June 1979, pp 103-109.

Littlewood,B. [1980], "What makes a reliable program- Few bugs or a small failure rate?", *National Computer Conference 1980*, pp 707-712.

Littlewood,B. [1981], "Stochastic reliability-growth: A model for fault-reoval in computer programs and hardware designs", *IEEE Transaction on Reliability*, vol. R-30, No. $, October 1981, pp 313-320.

Lloyd,J.W. [1983], "An introduction to deductive Database Systems", *Australian Computer Journal*, 15(2), pp 52-57.

Lloyd,J.W. [1984], *The foundation of Logic Programming* Springer-Verlag, Germany, 1984.

Loveland,D.W. [1969], "Theorem provers combining model elimination and resolution" *Machine Intelligence 4*, Meltser,B. and Michie,D.(eds), Elsevier North Holland, New York.

Loveland,D.W. [1970], "A linear format for resolution", *Proceedings IRIA Symposium on Automatic Demonstration*, Versaillles, France, Springer-Verlag, New York, 1968, pp147-162.

Luckham,D. [1970], "Refinement theorems in resolution theory" *Proceedings IRIA Symposium on Automatic Demonstration*, Versaillles, France, Springer-Verlag, New York, 1968, pp163-190.

Mamdani,E.H. [1974], "Applications of fuzzy algorithms for control of simple dynamic plant" *Proc. IEEE (1974)* pp1585-1588.

Manna,Z. and Waldinger,R. [1978], "The logic of computer programming", *IEEE Transactions on Software Engineering*, Vol. SE-4, May 1978, pp199-229.

Manna,Z. and Waldinger,R. [1980], "A deductive approach to program synthesis", *ACM transactions on Programming languages and Systems*, 2(1), pp90-121.

Manna,Z. and Waldinger,R. [1985], *The Logical Basis for computer programming, Volume 1. deductive Reasoning*, Addison-Wesley.

Mawdsley,A. [1984], *An Inter-ligual English to French Machine Translation System* M Sc Dissertation, Loughborough University of Technology, September 1984.

McArthur,T. [1981], *Longman Lexicon of Contemporary English*. Longman Group Ltd. 1981.

McCarthy,J. and Hayes,P.J. [1969], "Some philosphical problems from the standpoint of artificial intelligence". In Meltzer,B and Mitchie, D. *Machine Intelligence 4* Edinburgh University Press,New York, 1969.

Mellish,S. and Hardy,S. [1984], "Integerating Prolog in the POPLOG environment", in *Implementation of Prolog*, Campbell,J.A.(ed), Ellis Horwood.

Mitchel,T.M., Utgoff,P.E and Banerji,R. [1983], "Learning by experimentation: acquiring and modifying problem-solving heuristics", in *Machine Learning*, Michalski,R.S., Carbonell,J.G. and Mitchell,T.M. (Eds), Tioga, Palo Alto, 1983, pp163-190.

Mitchel,T.M., Utgoff,P.E., Nudel,B. and Banerji,R. [1981], "Learning problem-solving heuristics through practice", in *Proceedings Seventh International Joint Conference on Artificial Intelligence*, Vancauver, BC, 1981, pp127-134.

Minsky,M. [1975], "A Framework for representing Knowledge" in *The Psychology of Computer Vision*, Winston,P.H (ed), McGraw Hill, New York.

Montague,R. [1973], "The proper reatment of quantification in ordinary English", in *Approaches to Natural Languages*, Hintikka,K.J.J., Dordrecht, Germany, 1973.

Montague,R. [1974], *Formal Philosophy: Selected papers of Richad Montague*, Yale University Press, New Haven, 1974.

Moto-Oka,T. (ed) [1982], "Fifth Generation Computer Systems", *Proceedings International Conference on Fifth Generation Computer Systems*, JIPDEC, North-Holland, 1982.

Murray,N.V. [1982], "Completely non-clausal theorem proving", *Artificial Intelligence*, 18(1), pp 67-85

Musa,J.D. [1979], "Validity of execution-time theory of software reliability" *IEEE Transactions on Reliability*, Vol R-28, No. 3, August 1979, pp181-191.

Myers,G.J. [1978], "A Controlled experiment in program testing and code walkthroughs/inspection", *COmmunication of the ACM*, Vol 21, pp 760-768, 1978

Mylopoulus,J., Borgida,A., Cohen,P., Roussopoulus,N., Tsotsos,J. and Wong,H.K.T. [1976], "TORUS - a natural language understanding system for data management", in *Proceedings of the Fourth IJCAI*, pp414-421, Tiblis

Naur,P. [1969], "Programming by action clusters", *BIT*, Vol 9, 1969, pp250-258.

Naur,P. and Randel,B. (eds) [1969], *Software Engineering*, NATO Scientific Affairs Division, Brussels, Belgium , 1969.

Nilsson,N.J. [1971], *Problem-Solving methods in Artificial Intelligence*, McGraw Hill, United States of America, 1971.

Nilsson,N.J. [1979], "A production system for automatic deduction", *Machine Intelligence 9*, Hayes,J.D., Michie,D., and Mikulich,L I. (eds), Ellis Horwood, Chichester, 1979.

Nilsson,N.J. [1980], *Principles of Artificial Intelligence*, Springer Verlag, Germany, 1971.

Nishida,T. and Doshita,S. [1983], "An application of Montague grammar to English-Japanese machine translation" in *"Proceedings of the Conference on Applied Natural Language Analysis*, Santo Monica, California.

Ogdin,C.A. [1979], *Software design for micro computers*, Prentice-Hall, Englewood CLiffs, NJ.

Palmer,F.R. [1976], *Semantics - a new outline* Cambridge University Press, 1976.

Pereira,F. [1982], *C-Prolog User's Manual* University of Edinburgh: Dept. of Computer Aided Architectural Design.

Pereira,L.M., Pereira,F. and Warren,D.H.D. [1978], *User's Guide to DEC-system-10 Prolog*, University of Edinburgh: Dept. of Artificial Intelligence.

Pereira,F.C.N. and Warren,D.H.D. [1980], "Definite clause grammars for language analysis- a survey of the formalism and a comparison with augmented transition networks". *Artificial Intelligence* 13:3(1980), pp231-178.

Peterson,J.L. [1980], "Computer Programs for Detecting and Correcting Spelling Errors", *Comm of the ACM, 23(12)*, Dec. 1980, pp676-687.

Peterson,J.L. [1986], "On note on undetected typing errors", *Comm. of the ACM, 29(7)*, July 1986, pp633-637.

Prather,R.E. [1984], "An axiomatic theory of software complexity measure", *The computer Jounal*, Vol 27, No. 14, 1984, pp340-347.

Prawitz,D. [1960], "An improved Proof Procedure", *Theoria*, 26(1960), pp102-139

Prawitz,D. [1976], "A proof procedure with matrix reduction", in *Lecture notes in Mathematics*, Spring-Verlag, Berlin and New York.

Quillian,R. [1968], "Semantic Memory" in *Semantic Information Processing*, Minsky,M.(eds), MIT Press, Cambridge, Mass., 1968.

Quintus Prolog User's Guide and Reference Manual [1985], *Quintus Computer System Inc.*, Palo Alto, 1985.

Rapheal,B. [1968], "A computer program for Semantic Information Retrieval" in *Semantic Information Processing*, Minsky,M.(eds), MIT Press, Cambridge, Mass., 1968.

Rault,J.C. [1979], "An approach towards reliable software", *IEEE 1979*, pp 220-227.

Reiter,R. [1971], "Two results on ordering for resolution with merging and linear format", *Journal of ACM*, vol 18, pp 630-646.

Reiter,R. [1978], "On closed world data bases", in *Logic and Databases*, Gallaire,H. and Minker,J. (eds), Plenum Press, New York, 1978, pp55-76.

Rich,E. [1983], *Artificial Intelligence* McGraw Hill, Japan, 1983.

Roberts,R.B. and Goldstein,I.P. [1977], *The FRL Primer* Memo 408, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Mass.

Robinson,J.A. [1965], "A machine-oriented Logic Based on the Resolution Principle" *Journal of the ACM*, 12(1), 1965, pp23-41.

Robinson,J.A. [1965a], "Automatic deduction with hyper-resolution" *International Journal of Computing Mathematic*, 1, pp227-234.

Robinson,J.A. [1979], *Logic: Form and Function, the mechanization of deductive reasoning*, Edinburgh University Press.

Rogers,H., Jr. [1967], *Theory of Recursive Functions and Effective Computability*, McGraw Hill, 1967.

Rumelhart,D.E. and Norman,D.A. [1975], "The active structural network" in *Explorations in Cognition*, Norman,D.A and Rumelhart,D.E. (eds), W.H.Freeman, San Francisco

Sallih,M.M. [1986], *A study of Models for Predicting Computer Software Reliability* M.Phil thesis, Loughborough University of Technology, January 1986.

Sandwell,E. [1973], "Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs", *Proceedings of third International Joint Conference on AI - 73*, Standford University, California, pp 230-234

Schank,R.C. [1973], "Identification of Conceptualization Underlying Natural language" in *Computer Models of Thought and Language* Schank,R.C and Colby,K.M.(Eds), Freeman, San Francisco, 1973.

Schank,R.C. [1975], *Conceptual Information Processing* North-Holland, Amsterdam, 1975.

Schank,R.C and Abelson,R.P. [1977], *Scripts, Plan, Goals, and Understanding* Erlbaum, Hillsdale, N.J., 1977.

Schagen,I.P. [1985], "Software Reliability", *Department of Computer Studies Seminar, Lough. Univ. of Technology*, 1985.

Shapiro,E.Y. [1982], *Algorithmic Program Debugging*, MIT Press, Cambridge Mass. and London, 1982.

Shapiro,E.Y. [1983], "A subset of Concurrent PROLOG and its interpreter", *Technical Report TR-003*, ICOT, Tokyo, 1983.

Shapiro,E.Y. and Takeuchi,A. [1983], "Object-oriented programming in Concurrent PROLOG", *New generation Computing*, 1(1), pp25-88.

Shepherdson,J.C. [1984] "Negation as Failure: A comparison of clark's completed data base and Reiter's closed world assumption", *Report PM-84-01*, School of Mathematics, University of Bristol, Bristol.

Shooman,M.L. [1972], "Probablistic models for software reliability prediction", *Conference, Statsitical computer Performance Evaluation*, Academic Press, NewYork and London, 1972, pp48-55.

Simmons,R.F. [1973], "Semantic networks: their computation and use for understanding English sentences" in *Computer Models of Thought and Language*, Schank,R. and Colby,K. (eds), pp63-113, W.H.Freeman, San Francisco.

Slagle,J.R. [1967], "Automatic theorem proving with renamable and semantic resolution", *Journal of ACM*, 14(2), pp687-697.

Smith,D.E. and Clayton,J.E. [1980], "A frame based production system architecture" in *Proceedings of the AAAI 80*, Stanford University, California

Smith,R.G. and Friedland,P. [1980], "A user guide to the UNITS system", *Technical Report*, Heuristic Programming Project, Stanford University, California.

Sterling,L. and Shapiro,E. [1986], *The Art of Prolog*, MIT Press, USA, 1986.

Stickel,M.E. [1982], "A non-clausal connection-graph resolution theorem proving program", *Proceedings of the AAAI 82*, University of Pittsburg, Pennsylvania.

Storm,E.F. [1974], "Evaluation procedures for resolution without normal forms" *System and Information Science Report*, Syracause University, Syracause, New York.

Synder,D.P. [1971], *Modal logic and its applications* Van Nostrand, New York, 1971

Szolovits,P., Hawkinson,L.R., and Martin,W.A. [1977], "An overview of OWL: a language for knowledge representation", *Report Massachusetts Institute of Technology/LCS/TM-86*, MIT, Cambridge, Mass.

Vessey,I. [1986], "Expertise in Debugging Computer Programs: An analysis of the Content of Verbal Protocols", *IEEE Transactions on Systems, Man, and Cybernatics*, Vol SMC-16, No. 5, Sept/Oct 1986, pp 621-637.

Walker,D.E.(ed) [1978], *Understanding Spoken Language*. North-Holland, New York.

Warren,D.H.D. [1980], "Higher-order extensions to Prolog - are they needed", *Department of Artificial Intelligence Research Report 154*, Edinburgh Unniversity.

Waterman,D.A. [1970], "Generalization learning techniques for automating the learning of heuristics", *Artificial Intelligence*, Vol 1, 1970, pp121-170.

Weinberg,V. [1979], *Structured Analysis*, Gower Publishing, Farnborough, Hants.

Wilkins,D. [1974], "A non-clausal theorem proving system", *Proceddings of the AISB Summer Conference*, Brighton, UK.

Winograd,T. [1980], *Language as a Cognitive Process. Volume 1: Syntax*. Addison-Wesley, 1980.

Winston,P.H. [1975], "Learning structural descriptions from example" in *The Psychology of Computer Vision*, Winston,P.H. (ed), McGraw Hill, 1975.

Wos,L., Carson,D.E. and Robinson,G.A. [1964], "The unit preference strategy in theorem proving", *Porc. AFIPS 1964 Fall Joint Computer Conference*, vol 26, pp 616-621.

Wos,L., Carson,D.E. and Robinson,G.A. [1965], "Efficiency and completeness of the set of support strategy in theorem proving" *Journal of ACM*, 12(4), pp536-541

Yourdan,E. and Constantine,L.L. [1979], *Structure Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

Zadeh,L.A. [1965], "Fuzzy sets" *Information and Control*, Vol 8, 1965. pp338-353.

Zadeh,L.A. [1973], "Outline of a new approach to the analysis of complex system and decision processes" *IEEE Trans. Syst. Man and Cybern.* Vol 1, 1973, pp28-44.

Zadeh,L.A. [1983], "Commonsense knowledge representation based on based fuzzy logic", *Computer* Vol 16(10),pp61-65,1983.

APPENDIX

```
/*********************************************************
   To define the operators and the most common predicates
*********************************************************/

:- [oplog].     /* to be used by prolog/poplog only */

/*********************************************************
  Transforming predicate calculus into clausal form and
  print it as Clocksin's format.
*********************************************************/

:- ['PC/pc_top'].
:- consult('PC/pc_skolem').
:- consult('PC/pc_bottom').
:- consult('PC/pc_skolemq').
:- ['PC/substitute'].
:- consult('PC/pc_gensym').
:- consult('PC/pc_trans').
:- consult('PC/hornclause').
:- ['PC/outputfp4'].

/*********************************************************
newprove::
  (a) the replacement of relevent quantifiers only
  (b) no duplication of reason
  (c) handles both english and pc as input and output
*********************************************************/

:- ['PC/ansearch'].
:- consult('PC/fprolog13').
:- consult('PC/listquant').
:- consult('PC/mergequant').
:- consult('PC/reasontest').
:- consult('PC/topfprolog').
:- ['Y/aaccsubst'].
:- consult('Y/acceptance').
:- consult('Y/acceptsubs').
:- consult('Y/asserting').
:- consult('Y/changequant').
:- consult('Y/equatevars').
:- consult('Y/fdifference').
:- consult('Y/listreason').
:- ['Y/mismatclause'].
:- consult('Y/pracceptance').
:- consult('Y/prtreefail').
:- consult('Y/topc').
:- consult('Y/whyitfails').
:- [othermeaning].
:- consult(pctop).
:- consult(rectifiers).
:- consult(remarkwriter).
:- consult(toplevel).
```

```
:- op(255,xfx,:).
:- op(225,xfx,<=>).
:- op(225,xfx,=>).
:- op(200,xfy,&).
:- op(200,xfy,#).
:- op(30,fx,~).
:- op(255,xfx,::).
:- op(15,xfx,^).
:- library(log).
:- library(date).

/*to print system date and time */
date:-date([H,B,T,J,M,S]),
        write(B),write(','),write(H),write('  '),
        write(T), write('  '),write(J),write(':'),
        two_digit(M),write(':'),two_digit(S),nl.

two_digit(Num):-
        name(Num,List),
        two_digitlist(List),
        write(Num).

two_digitlist([X]):-!,write('0').
two_digitlist(X).

logdate:-log,
        write(' on '),date,nl.

nologdate:-
        nl,write('On '),date,nolog.

/* to check membership */
member(X,Y):-var(Y),!,fail.
member(X,[Y|T]):-X==Y.
member(X,[Y|T]):-member(X,T).

append([X|A],B,[X|C]):-append(A,B,C).
append([],B,B).

/* to prevent backtracking */
append1(X,Y,Z):-append(X,Y,Z),!.


/* to change the output file from file X to user */
tells(X):-telling(X),tell(user).

/* asserting an atomic only once */
assert_once(X):-clause(X,true),!.
assert_once(X):-assert(X).

/* asserting a predicate F only once */
assert_pred_once(X):-
                functor(X,F,N),functor(Y,F,N),
                clause(Y,true),!.
assert_pred_once(X):-
                assert(X).
```

```
/* replacing old predicate or atomic with a new one */
assertz_new(X):-
        functor(X,F,N),functor(Y,F,N),
        retractall(Y),assert(X).

/* existence and non existence test of
         a predicate in a database */
not_exists(X):-
        clause(X,true),
        !,fail.
not_exists(X):-
        true.

exists(X):-
        clause(X,true),!.

/* defination of digit(D) */
digit(D):- 47<D,D>58.
```

```
/*  TOP OF PREDICATE TRANSLATE (PC_TOP) */

/* Stage 1: taking out implication sign */
implout((P<=>Q),(P1&Q1):-
        !,
        implout((P=>Q),P1),
        implout((Q=>P),Q1).
implout((P=>Q),(~P1#Q1)):-
        !,
        implout(P,P1),
        implout(Q,Q1).
implout(T:P,T:Q):-
        !,
        implout(P,Q).
implout((P&Q),(P1&Q1)):-
        !,
        implout(P,P1),
        implout(Q,Q1).
implout((P#Q),(P1#Q1)):-
        !,
        implout(P,P1),
        implout(Q,Q1).
implout((~P),(~P1)):-
        !,
        implout(P,P1).
implout(all(X,P),all(X,P1)):-
        !,
        implout(P,P1).
implout(all(X,D,P),all(X,D,P1)):-
        !,
        implout(P,P1).
implout(exists(X,P),exists(X,P1)):-
        !,
        implout(P,P1).
implout(exists(X,D,P),exists(X,D,P1)):-
        !,
        implout(P,P1).
implout(P,P).

/* Stage 2: to bring in negation sign */
negin((~P),P1):-
        !,
        neg(P,P1).
negin(T:A,T:B|C]):-
        !,
        negin(A,B,C).
negin((P&Q),(P1&Q1)):-
        !,
        negin(P,P1),
        negin(Q,Q1).
negin((P#Q),(P1#Q1)):-
        !,
        negin(P,P1),
        negin(Q,Q1).
```

```
negin(all(X,P),all(X,P1)):-
        !,
        negin(P,P1).
negin(all(X,D,P),all(X,D,P1)):-
        !,
        negin(P,P1).
negin(exists(X,P),exists(X,P1)):-
        !,
        negin(P,P1).
negin(exists(X,D,P),exists(X,D,P1)):-
        !,
        negin(P,P1).
negin(P,P).

neg((~P),P1):-
        !,
        negin(P,P1).
neg(T:A,T:B|C]):-
        !,
        neg(A,B,C).
neg((P&Q),(P1#Q1)):-
        !,
        neg(P,P1),
        neg(Q,Q1).
neg((P#Q),(P1&Q1)):-
        !,
        neg(P,P1),
        neg(Q,Q1).
neg(exists(X,P),all(X,P1)):-
        !,
        neg(P,P1).
neg(exists(X,det(the),P),exists(X,det(the),P1)):-
        !,
        neg(P,P1).
neg(exists(X,D,P),all(X,D,P1)):-
        !,
        neg(P,P1).
neg(all(X,P),exists(X,P1)):-
        !,
        neg(P,P1).
neg(all(X,D,P),exists(X,D,P1)):-
        !,
        neg(P,P1).
neg(P,(~P)).
```

```
/* STAGE 3: skolemizing a variable of knowledge statements */
skolem(T:A,T:B,C):-
        !,
        skolem(A,B,C).
skolem((P#Q),(P1#Q1),Vars):-
        !,
        skolem(P,P1,Vars),
        skolem(Q,Q1,Vars).
skolem((P&Q),(P1&Q1),Vars):-
        !,
        skolem(P,P1,Vars),
        skolem(Q,Q1,Vars).
skolem(all(X,P),all(X,P1),Vars):-
        !,
        skolem(P,P1,[X|Vars]).
skolem(all(X,D,P),all(X,D,P1),Vars):-
        !,
        skolem(P,P1,[X|Vars]).
skolem(exists(X,P),P2,Vars):-
        pickname(X,P,Name),
        pickskolem(Name,anything,Vars,Sk),
        substitute(Sk,X,P,P1),
        skolem(P1,P2,Vars).
skolem(exists(X,proper_noun(X),P),P1,Vars):-
        !,
        skolem(P,P1,Vars).
skolem(exists(X,definite(X),P),P2,Vars):-
        pickname(X,P,Name),
        pickskolem(Name,the,Vars,Sk),
        substitute(Sk,X,P,P1),
        skolem(P1,P2,Vars).
skolem(exists(X,det(The),P),P2,Vars):-
        The==the,!,
        pickname(X,P,Name),
        pickskolem(Name,the,Vars,Sk),
        substitute(Sk,X,P,P1),
        skolem(P1,P2,Vars).
skolem(exists(X,D,P),P2,Vars):-
        pickname(X,P,Name),
        pickskolem(Name,D,Vars,Sk),
        substitute(Sk,X,P,P1),
        skolem(P1,P2,Vars).
skolem(P,P,Vars).

/* Procedure PICKSK: picking a Skolem variable
                     for knowledge statements */
pickskolem(Name,the,Vars,Sk):-
        skolem_the(Name,Sk),
        !.
pickskolem(Name,D,Vars,Sk):-
        gensym(Name,F),
        append([F],Vars,Fandargs),
        Sk=..[fs|Fandargs],
        asserting(skolem_the(Name,Sk)).
```

```
asserting(F):-
        F=..[F1,N,Sk],
        G=..[F1,N,X],
        retractall(G),
        asserta(F).
```

```
/* BOTTOM'S PART OF PREDICATE TRANSLATE (PC_BOTTOM) */

/* STAGE 4: deleting all universal quantifiers */
univout(T:P,T:P1):-
        univout(P,P1).
univout(all(X,P),P1):-
        !,
        univout(P,P1).
univout(all(X,D,P),P1):-
        !,
        univout(P,P1).
univout((P&Q),(P1&Q1)):-
        !,
        univout(P,P1),
        univout(Q,Q1).
univout((P#Q),(P1#Q1)):-
        !,
        univout(P,P1),
        univout(Q,Q1).
univout(P,P).

/* STAGE 5: transforming into conjunction form */
conjn((P#Q),R):-
        !,
        conjn(P,P1),
        conjn(Q,Q1),
        conjn1((P1#Q1),R).
conjn((P&Q),(P1&Q1)):-
        !,
        conjn(P,P1),
        conjn(Q,Q1).
conjn((A:P),(A:P1)):-
        conjn(P,P1).
conjn(P,P).

conjn1(((P&Q)#R),(P1&Q1)):-
        !,
        conjn((P#R),P1),
        conjn((Q#R),Q1).
conjn1((P#(Q&R)),(P1&Q1)):-
        !,
        conjn((P#Q),P1),
        conjn((P#R),Q1).
conjn1((P#T:(Q)),T:(R)):-
        conjn1((P#Q),R).
conjn1((A:P),(A:P1)):-
        conjn1(P,P1).
conjn1(P,P).

/* STAGE 6: transforming into clausal form, cl(A,B) */

clausify((P&Q),C1,C2):-
        !,
        clausify(P,C1,C3),
        clausify(Q,C3,C2).
```

```
clausify(P,[cl(A,B)|Cs],Cs):-
        inclause(P,A,[],B,[]),
        !.
clausify(P,C,C).

inclause((P#Q),A,A1,B,B1):-
        !,
        inclause(P,A2,A1,B2,B1),
        inclause(Q,A,A2,B,B2).
inclause((~P),A,A,B1,B):-
        !,
        notin(P,A),
        putin(P,B,B1).
inclause(P,A1,A,B,B):-
        notin(P,B),
        putin(P,A,A1).

notin(X,[Y|L]):-
        X == Y,
        !,
        fail.
notin(X,[Y|L]):-!,
        notin(X,L).
notin(X,[]).

putin(X,[],[X]):-!.
putin(X,[Y|L],[Y|L]):-
        X == Y,
        !.
putin(X,[Y|L],[Y|L1]):-
        putin(X,L,L1).

/* STAGE 7: writing cl(A,B) in Clocksin's format */

buildclauses([cl(A,B)|Cs],[D|D1]):-
        buildclause(A,B,D),
        buildclauses(Cs,D1).
buildclauses([],[]):-!.

buildclause(L,[]):-!,disjunc(L,D).
buildclause([],L,[]:-B):-!,conjunc(L,B).
buildclause(L1,L2,(A:-B)):-
        disjunc(L1,A),conjunc(L2,B).

disjunc([L],L):-!.
disjunc([L|Ls],(L;A)):-disjunc(Ls,A).

conjunc([L],L):- !.
conjunc([L|Ls],(L,A)) :-conjunc(Ls,A).

/* STAGE 8: print clauses */
printclauses([]):-!.
printclauses([X|Y]):-
        write(X),write('.'),nl,
        printclauses(Y).
```

```
/* STAGE 3q:  skolemizing a question */

/* to convert existential quantifiers into skolem functions */
skolemq(T:A,T:B,C,S):-
        !,
        skolemq(A,B,C,S).
skolemq((P#Q),(P1#Q1),Vars,S):-
        !,
        skolemq(P,P1,Vars,S1),
        skolemq(Q,Q1,Vars,S2),
        append1(S1,S2,S).
skolemq((P&Q),(P1&Q1),Vars,S):-
        !,
        skolemq(P,P1,Vars,S1),
        skolemq(Q,Q1,Vars,S2),
        append1(S1,S2,S).
skolemq(all(X,P),all(X,P1),Vars,S):-
        !,
        skolemq(P,P1,[X|Vars],S).
skolemq(all(X,D,P),all(X,D,P1),Vars,S):-
        !,
        skolemq(P,P1,[X|Vars],S).
skolemq(exists(X,P),P2,Vars,S):-
        picking_skolem(exists(X,P),P2,anything,Vars,Sk,S).
skolemq(exists(X,proper_noun(X),P),P2,Vars,S):-
        !,
        skolemq(P,P2,Vars,S).
skolemq(exists(X,definite(X),P),P2,Vars,[s12,S):-
        !,
        picking_skolem(exists(X,P),P2,the,Vars,Sk,S).
skolemq(exists(X,indefinite(X),P),P2,Vars,[s12,S):-
        !,
        picking_skolem(exists(X,P),P2,anything,Vars,Sk,S).
skolemq(exists(X,det(D),P),P2,Vars,S):-
        picking_skolem(exists(X,P),P2,D,Vars,Sk,S).
skolemq(P,P,Vars,[]).

substitute1(X,Y,A,B):-substitute(X,Y,A,B),!.

/* by converting to a skolem function */
picking_skolem(exists(X,P),P2,D,Vars,Sk,[[X,Sk]|S]):-
        pickname(X,P,Name),
        pickskolemq(Name,D,Vars,Sk),
        substitute1(Sk,X,P,P1),
        skolemq(P1,P2,Vars,S).
```

```
/* picking the Skolem function for existential quantifiers */
pickskolemq(Name,the,Vars,Sk):-
        skolem_the(Name,Sk),
        !.
pickskolemq(Name,the,Vars,Sk):-
        skolemq_the(Name,Sk),
        !.
pickskolemq(Name,the,Vars,Sk):-
        gensym(Name,F),
        append1([F],Vars,Fandargs),
        Sk=..[fs|Fandargs],
        asserting(skolem_the(Name,Sk)),
        !.
pickskolemq(Name,D,Vars,Sk):-
        gensym(Name,F),
        append1([F],Vars,Fandargs),
        Sk=..[fq|Fandargs],
        asserting(skolemq_the(Name,Sk)),
        !.
```

```
/* PC/substitute */

/* to substitute a variable Old1 with New in clause Old2 */
substitute(New, Old1, Old2, New) :-
        /* to tackle Old1 and Old2 in the form of fq(...) */
        nonvar(Old1),
        nonvar(Old2),
        Old1 = Old2,
        !.
substitute(New,Old1,Old2,New):-
        Old1 == Old2,
        !.
substitute(New1,Old1,Old2,Old2):-
        var(Old2),
        Old1 \== Old2,
        !.
substitute(New,Old,Val,Val):-
        not(var(Val)),
        atomic(Val),
        !.
substitute(New,Old,(~P),(~P1)):-
        substitute(New,Old,P,P1).
substitute(New,Old,Val,Newval):-
        Val=..[Fn|Args],
        subst_args(New,Old,Args,Newargs),
        substitute(New,Old,Fn,Fn1),
        newval(Fn1,Newargs,Newval).

newval(~Fn1,Newargs,~Newval):-!,
        Newval=..[Fn1|Newargs].
newval(Fn1,Newargs,Newval):-!,
        Newval=..[Fn1|Newargs].

subst_args(X,Y,[],[]):-
        !.
subst_args(New,Old,[Arg|Args],[Newarg|Newargs]):-
        substitute(New,Old,Arg,Newarg),
        subst_args(New,Old,Args,Newargs).
```

```
/* Create a new atom starting with a root provided and
        finishing with a unique number */
gensym(Root,Root):-
        var(Root),
        !.
gensym(Root,Atom):-
        get_num(Root,Num),
        name(Root,Name1),
        name(Num,Name2),
        append(Name1,Name2,Name),
        name(Atom,Name).

get_num(Root,Num):-
        /* this root encountered before */
        retract(current_num(Root,Num1)),
        !,
        Num is Num1 + 1,
        asserta(current_num(Root,Num)).
get_num(Root,0):-
        /* first time for this root */
        asserta(current_num(Root,0)).


/* to pickup the name of atom  */
pickname(X,(P#Q),Name):-
        pickname(X,P,Name),
        !.
pickname(X,(P&Q),Name):-
        pickname(X,P,Name),
        !.
pickname(X,(T:(P)),Name):-
        pickname(X,P,Name),
        !.
pickname(X,~P,Name):-
        pickname(X,P,Name),
        !.
pickname(X,P,Name):-
        P =.. [f,Name|Args], /* fact => f */
        !.
pickname(X,P,Name):-
        P =.. [Name|Args],
        Name\==exists,
        Name\==all,
        !.
pickname(X,P,skolem).
```

```
/* Procedure STASZ: assert "knowledge(X)"into
        the database */
stassertz([]):-!.
stassertz([X|Y]):-
        assertz(knowledge(X)),
        stassertz(Y).

/* Procedure TRPCK: to translate PC, "X", into
        Clocksin format, "Clause" */
translate(X,Clauses):-
        translate_top(X,X2),
        skolem(X2,X3,[]),
        translate_bottom(X3,Clauses).

/* procedure TRTOP */
translate_top(X,X2):-
        implout(X,X1),
        negin(X1,X2),
        !.

/* procedure TRTOP */
translate_bottom(X3,Clauses):-
        univout(X3,X4),
        conjn(X4,X5),
        clausify(X5,X6,[]),
        buildclauses(X6,Clauses),
        printclauses(Clauses),
        !.

/* Procedure TRQ: to translate PC, "X", into
        Clocksin format, "Clause" */
translateq(X,Clauses,Sk):-
        translate_top(X,X2),
        skolemq(X2,X3,[],Sk),
        translate_bottom(X3,Clauses).
```

```
/* Procedure QASS: assert "query(X)" into the database */
qassertz([]):-!.
qassertz([X|Y]):-
        assertz(query(X)),qassertz(Y),!.

/* Procedure ASSKG: to assert knowledge clauses into
        the database */
assert_knowledge(X):-
        nl,write('knowledge statement:'),nl,tab(4),write(X),nl,
        write('=======================>'),nl,
        write('The translated clauses :'),nl,nl,
        translate(X,Clause),
        horn_clauses(Clause,Horn),
        write('&&&&&&&&&&&&nextnextnext&&&&&&&&&&&&'),nl,
        stassertz(Horn).

/* Procedure QTHC */
question_to_hornclause(Q,Clause,Sklist):-
        assert_query(~Q,Clause,Sklist).

/* Procedure ASSQ: to assert knowledge clauses into
        the database */
assert_query(X,Clause,Sklist):-
        translateq(X,Clause,Sklist),
        horn_clauses(Clause,Horn),
        write('&&&&&&&&&&&&&&&&&&&&&&&&'),nl,
        qassertz(Horn).

/* Procedure HC: converting a clausal into headed Horn clause */
horn_clauses([],[]):-!.
horn_clauses([Second1|Second2],First):-
        horn_clauses1(Second1,First1),
        horn_clauses(Second2,First2),
        append(First1,First2,First),
        !.

/* Procedure HC1: creating N equivalence headed Hornclause */
horn_clauses1([]:-Body,C):-
        !,
        horn_clauses2(Body,[],C).
horn_clauses1(Head:-Body,C):-
        !,
        convert(Head,Body1),
        appendbody(Body1,Body,Body2),
        horn_clauses1([]:-Body2,C),!.
horn_clauses1((Head;Head1),C):-
        !,
        convert((Head;Head1),Body),
        horn_clauses1([]:-Body,C),!.
horn_clauses1(Atomic,[Atomic]).
```

```
/* Procedure HC2 */
horn_clauses2([],Body,[]):-!.
horn_clauses2((Body,Body1),Body2,[Nothead:-Body3|D2]):-
        !,
        convert(Body,Nothead),
        appendbody(Body2,Body1,Body3),
        appendbody(Body2,Body,Body4),
        horn_clauses2(Body1,Body4,D2).
horn_clauses2(Body,[],[Nothead]):-!,
        convert(Body,Nothead).
horn_clauses2(Body,Body2,[Nothead:-Body2]):-
        convert(Body,Nothead).

/* Procedure CONV: moving literals into the lefts hand side
        or the right hand side of a rule as appropriate */
convert(A;B,(A1,B1)):-
        /* Procedure CONV.1 */
        !,convert(A,A1),convert(B,B1).
convert((K,L),(K1;L1)):-
        /* Procedure CONV.2 */
        !,convert(K,K1),convert(L,L1).
convert(~P,P):-!.       /* Procedure CONV.3 */
convert(P,~P).          /* Procedure CONV.4 */

/* Procedure APPB */
appendbody(K,[],K).
appendbody([],M,M).
appendbody((K,L),M,(K,N)):-appendbody(L,M,N).
appendbody(K,M,(K,M)).
```

```
/* file : PC/outputfp4 */

/* procedure ASSPRV2 */
assertaproving2(Q,N):-
        retractall(proving(_,N)),
        asserta(proving(Q,N)).

/* procedure ASSGL1 */
assertagoal1(Q,N):-
        node(N),
        retractall(goal(_,N)),asserta(goal(Q,N)),
        updating_node(N).


/* procedure UPNODE  */
updating_node(N):-N1 is N+1,assertnode(N1).
updating_node(N):-assertnode(N),fail.

assertnode(N):-rectractall(node(_)),asserta(node(N)).


/* Procedure HAVEPR */
haveproved(Q,[Head|Tail]):-Q==Head,!.
haveproved(Q,[Head|Tail]):-haveproved(Q,Tail).

haveproved_top(Q,Hp):-haveproved(Q,Hp),!.
haveproved_top(Q,Dummy):-nonvar(Q),
        proven(Q),!.


/* procedure NOTTRY1 */
nottry1(Q,[]).
nottry1(Q,[H|T]):-
        Q\==H,
        nottry1(Q,T),!.
nottry1(Q,[H|T]):-
        Q==H,
        asserta(looping(Q)),
        !,fail.

/* procedure ASSGL1 */
assertz_proven([Hp|Hptail]):-
        proven(Hp),assertz_proven(Hptail),!.
assertz_proven([Hp|Hptail]):-
        assertz(proven(Hp)),assertz_proven(Hptail).
assertz_proven([]).

/* procedure ASSGL1 */
check_failure(Q):-failure(Q),!,fail.
check_failure(Q):-asserta(failure(Q)).
```

```
/*******************************************************/
/* TO PRINT THE SOLUTION'S PATH OF THE QUERY(QUESTION) */
print_solution:-node(N),nl,nl,nl,
        write('The path of solution of the goal clause: '),
        goal(G,1),nl,tab(5),write('[]:-'),writegoal(G),
        nl,nl,print_solution1(N,1),!.

print_solution1(N,N1):-proving(P,N1),
        N1<N,
        goal(G,N1),
        nl,tab(4),write('[]:-'),writegoal(G),
        nl,tab(8),write('|'),
        nl,tab(8),write('| '),write(P),write('.'),
        nl,tab(8),write('|/'),
        N2 is N1+1,
        print_solution1(N,N2),!.
print_solution1(N,N1):-
        nl,tab(4),write('[]:-[].'),!,nl.

writegoal([G1|G2]):-
        writegoal1(G2),
        write(G1),write('.'),!. /* right to left */
writegoal(G):-
        write(G),write('.').  /* for left to right */

writegoal1([G2|G3]):-writegoal1(G3),write(G2),write(','),!.
writegoal1([]):-!.
```

```
/** PC/ansearch== **/

/* AS:searching for answers of the question */
answer_search(Clause,X,Y,Newskquant):-
        answer(X,Clause,Ans),
        assertz_new(affirm(Ans)),
        subst_skolem(X,Y,Newskquant).
answer_search(Clause,X,Y,Newskquant):-
        /* no more exists possible answers
                i.e the proving ends */
        Clause\==[],
        exists(toptry([])),!.
/* end AS:searching for answers of the question */
```

```
/** PC/fprolog13 == to prove from right to left */

/* Procedure ASK: to prove each clause of the question */
asking(Pc,[X|Y]):-
    top_asking(X,Q),
    factprolog(Q,[],[],Hp,[],[],Fc),
    reason_testing(Pc,Fc),
    successful_action(Hp).
asking(Pc,[X|Y]):-
    print_comment(X),
    asking(Pc,Y).
asking(Pc,[]):-
    assertz(toptry([])),fail.

/* Procedure FACTPR */
factprolog((Q1,Q2),Usedclauses,Hp,Hp1,Goalclause,Fc,Fc1):-
    !,
    factprolog(Q2,Usedclauses,Hp,Hp2,[Q1|Goalclause],Fc,Fc2),
    factprolog(Q1,Usedclauses,Hp2,Hp1,Goalclause,Fc2,Fc1).
factprolog(Q,Usedclauses,Hp,Hp1,Goalclause,Fc,Fc1):-
    assertagoal1([Q|Goalclause],N),
    baseprolog(Q,Usedclauses,Hp,Hp1,Goalclause,N,Fc,Fc1).
/* recording an unsatisfiable or a failure atom */
factprolog(Q,Usedclauses,Hp,Hp,Goalclause,Fc,[[Q,N]|Fc]):-
    test_failure_loop(Q),
    node(N),
    updating_node(N).

/* Procedure BASEPR: to match with a single atom */
baseprolog(Q,Usedclauses,Hp,Hp,Goalclause,N,Fc,Fc):-
    clause(Q,_),
    call(Q),   /* if Q is a system predicate */
    assertaproving2(Q,N).
baseprolog(Q,Usedclauses,Hp,Hp,Goalclause,N,Fc,Fc):-
    knowledge_base(Q),
    assertaproving2(Q,N).
baseprolog(Q,Usedclauses,Hp,Hp1,Goalclause,N,Fc,Fc1):-
    knowledge_base(Q:-A),
    factclause(Q:-A,Usedclauses,Hp,Hp1,Goalclause,N,Fc,Fc1).

/* Procedure FACTCL: to match with a headed clause */
factclause(Q:-A,Usedclauses,Hp,Hp,Goalclause,N,Fc,Fc):-
    haveproved_top(Q:-A,Hp),!,
    assertaproving2(Q:-A,N).
factclause(Q:-A,Usedclauses,Hp,[Q:-A|Hp1],Goalclause,N,Fc,Fc1):-
    nottry1(Q:-A,Usedclauses),      /* cycling test */
    check_failure(Q:-A),
    assertaproving2(Q:-A,N),
    factprolog(A,[Q:-A|Usedclauses],Hp,Hp1,Goalclause,Fc,Fc1),
    retractall(failure(Q:-A)).
```

```
/* to extract the failure or unproveable clause/fact */
test_failure_loop(Q):-
    not(knowledge_base_head(Q)),
    not(clause(Q,_)),!.
test_failure_loop(Q):-
    exists(looping(Q:-T)),
    exists(failure(Q:-T)),!.

/* Procedure SUCCESS */
successful_action(Hp):-
    assertz_proven(Hp).
```

```
/** PC/listquant  */

/* to list all quantifiers of PC formula */
list_quantifiers((P<=>Q),L):-
        !,
        list_quantifiers(P,L1),
        list_quantifiers(Q,L2),
        append(L1,L2,L).
list_quantifiers((P=>Q),L):-
        !,
        list_quantifiers(P,L1),
        list_quantifiers(Q,L2),
        append(L1,L2,L).
list_quantifiers(T:P,L):-
        !,
        list_quantifiers(P,L).
list_quantifiers((P&Q),L):-
        !,
        list_quantifiers(P,L1),
        list_quantifiers(Q,L2),
        append(L1,L2,L).
list_quantifiers((P#Q),L):-
        !,
        list_quantifiers(P,L1),
        list_quantifiers(Q,L2),
        append(L1,L2,L).
list_quantifiers((~P),L):-
        !,
        list_quantifiers(P,L).
list_quantifiers(all(X,P),[[all,X]|L]):-
        !,
        list_quantifiers(P,L).
list_quantifiers(all(X,D,P),[[all,X]|L]):-
        !,
        list_quantifiers(P,L).
list_quantifiers(exists(X,P),[[exists,X]|L]):-
        !,
        list_quantifiers(P,L).
list_quantifiers(exists(X,D,P),[[exists,X]|L]):-
        !,
        list_quantifiers(P,L).
list_quantifiers(P,[]).
```

```
/* to merge two lists i.e Skhead and Listquant
    into a new list, ie Newlist */

merge_quantifiers([Skhead|Tail],Listquant,[Newlist1|Newlist2]):-
    merge_quantifier(Skhead,Listquant,Newlist1),
    merge_quantifiers(Tail,Listquant,Newlist2),!.
merge_quantifiers([],Listquant,[]).

merge_quantifier([X,Sk],[[Quant,Y]|Tail],[X,Sk,Quant]):-
    X==Y,!.
merge_quantifier(Skhead,[Listqhead|Tail],Newlist1):-
    merge_quantifier(Skhead,Tail,Newlist1),!.
merge_quantifier(Skhead,[],[]).
```

```
/* transf(P,Q,R): to transform Q and P into R */
transf(Q,~P,~R):-transf(Q,P,R).
transf(all(Y,Q2=>Q1),all(X,P2=>P1),
        all(X,P2=>all(Y,Q2&Q1=>P1))):-!.
transf(all(Y,Q2=>Q1),exists(X,P2&P1),
        exists(X,P2&all(Y,Q2&Q1&P1))):-!.
transf(exists(Y,Q),all(X,P2=>P1),
        all(X,P2&exists(Y,Q)=>P1)):-!.
transf(exists(Y,Q),exists(X,P2&P1),
        exists(X,P2&exists(Y,Q&P1))):-!.
transf(Q,all(X,P2=>P1),all(X,P2&Q=>P1)):-!.
transf(Q,exists(X,P2&P1),exists(X,P2&Q&P1)):-!.
transf(Q,P,P&Q):-!.


incorporate(P,Q,R):-incorporate1(P,Q,R).
incorporate(P,Q,R):-incorporate2(P,Q,R).

incorporate1(P1,all(X,P=>P2),all(X,P=>Q)):-
        incorporate1(P1,P2,Q),!.
incorporate1(P1,exists(X,P),exists(X,P=>P1)):-!.
incorporate1(P1,all(X,Z,P=>P2),all(X,Z,P=>Q)):-
        incorporate1(P1,P2,Q),!.
incorporate1(P1,exists(X,Z,P),exists(X,Z,P=>P1)):-!.
incorporate1(P1,exists_the(X,Z,P),exists_the(X,Z,P=>P1)):-!.
incorporate1(P1,P=>P2,~P=>Q):-
        incorporate1(P1,P2,Q),!.
incorporate1(P1,P,P=>P1):-!.

incorporate2(P1,all(X,P=>Q),all(X,P&P1=>Q)):- !.
incorporate2(P1,exists(X,P),exists(X,P&P1)):-!.
incorporate2(P1,all(X,Z,P=>Q),all(X,Z,P&P1=>Q)):- !.
incorporate2(P1,exists(X,Z,P),exists(X,Z,P&P1)):- !.
incorporate2(P1,P,P&P1):- !.
```

```
/* to remove any duplicate reason from the list of reasons */

/* Procedure REASONING: to test the answers */
reason_testing(Goal,Fc):-
        Fc==[],
        !,used_query_only.
reason_testing(Goal,Fc):-
        abolish(failure_son,2),
        split_reason(Fc,Reason),
        not_exists(reject(Reason)),
        not_exists(consulted_sor(Goal,Reason)),
        not_exists(set_of_reason(_,goal(Goal),reason(Reason))),
        assert_reason_son_father(Goal),
        assert_set_of_reason(Goal,Reason),
        !,fail.

/* checking the use of query clauses only
                during proving  */
used_query_only:-
        node(N),
        used_query_only1(N),
        !,retract(used_fact_clause).

used_query_only1(N):-
        N1 is N-1,
        proving(Q,N1),
        used_query_only2(Q,N1).
used_query_only1(N).

used_query_only2(Q,N1):-
        query(Q),
        !,used_query_only1(N1).
used_query_only2(Q,N1):-
        fact_base(Q),
        assert(used_fact_clause),
        !,fail.

/* to split reason and its node number,
        and to assert failure_son/2
        and to remove any duplication of reason */
split_reason([[R,N]|Tail],Rtail):-
        exists(failure_son(R,N1)),
        assert(failure_son(R,N)),
        split_reason(Tail,Rtail),!.
split_reason([[R,N]|Tail],[R|Rtail]):-
        not_exists(failure_son(R,N1)),
        assert(failure_son(R,N)),
        split_reason(Tail,Rtail),!.
split_reason([],[]).

/* assert reason_son_father/3 by using proving/2 */
assert_reason_son_father(Goal):-
        node(N),
        N1 is N-1,
        access_proving(Goal,N1).
```

```
access_proving(Goal,0):-
        goal(G,1),
        assert_rsf(G,goal(Goal),0),
        abolish(failure_son,2),
        abolish(failure_father,2),
        !.
access_proving(Goal,N):-
        exists(proving(H:-T,N)),
        assert_rsf(T,H:-T,N),
        N1 is N-1,
        access_proving(Goal,N1).
access_proving(Goal,N):-
        not_exists(proving(H:-T,N)),
        N1 is N-1,
        access_proving(Goal,N1).

assert_rsf([G1|G2],Father,N):-
        assert_rsf(G1,Father,N),
        assert_rsf(G2,Father,N).
assert_rsf([],Father,N).

assert_rsf((T1,T2),Father,N):-
        assert_rsf(T2,Father,N),
        assert_rsf(T1,Father,N).
assert_rsf(T,Father,N):-
        exists(failure_son(T,N1)),
        N=<N1,
        option(Option),
        retract(failure_son(T,N1)),
        assert(rsf_head(Option,T,Father)),
        assert(failure_father(Father,N)).
assert_rsf(T,Father,N):-
        exists(failure_father(T:-Tail,N1)),
        N=<N1,
        option(Option),
        assert(reason_son_father(Option,T,Father)),
        assert(failure_father(Father,N)).
assert_rsf(T,Father,N).
```

```
/* proving the conclusion of questions as prolog system */
/* with addition of testing failure clause */
/* in order to reduce execution time*/
/* set_of_reason are taken in FIFO's order */

/* Procedure CLEARPC */
clear_pc:-
        abolish(reject,1),
        abolish(query,1),
        abolish(current_numqs,1),
        abolish(toptry,1),
        abolish(toptry0,1),
        abolish(option,1), assert(option(1)),
        abolish(set_of_reason,3),
        abolish(old_set_of_reason,3),
        abolish(failure_father,2),
        abolish(failure_son,2),
        abolish(substitution_log,1),
        abolish(consulted_sor,2),
        abolish(reason_son_father,3),
        abolish(rsf_head,3),
        abolish(mismatch_pair,4),
        abolish(howmany_subst,2),
        abolish(skolemfq_list,1),
        abolish(mismatch_pair,2),
        abolish(mismatch_list,1),
        abolish(old_goal,1),
        clear_pc1.

/* Procedure CLEAR-PC1 */
clear_pc1:-
        abolish(accepted_substitution,1),
        abolish(subsidiary_list,1),
        abolish(scan_fail,1), /* depth+breadth first */
        abolish(failure,1),
        abolish(looping,1),
        retractall(toptry([])),
        abolish(proven,1),
        abolish(proving,2),
        abolish(goal,2),
        abolish(try,2),
        abolish(try,1).

/* Procedure ANS: answering the question */
answer(Pc,Q,yes):-
        asking(Pc,Q),
        assert_once(toptry0(Pc)).
answer(Pc,Q,no):-
        not_exists(toptry0(Pc)).

/* Procedure TOPASK */
/* top portion of predicate 'asking' */
top_asking(X,Q):-
        assertnode(1),  /* to initialize node(1) */
        format_goal(X,Q),
        !.
```

```
/* Procedure PRCMT: to print comment */
print_comment(X):-
        not_exists(toptry(X)),
        nl,write('>>>>>> can not prove '),
        write(X),write(' <<<<<<'),nl,
        !.
print_comment(X).

/* defination of KB clauses (data base) */
knowledge_base(Q):-
        query(Q).
knowledge_base(Q):-
        fact_base(Q).

/* defination of fact base clause (data base) */
fact_base(Q):-
        clause(knowledge(Q),true).
fact_base(Q):-
        clause(subsidiary(Q),true).
fact_base(Q):-
        clause(plausible(Q),true).

/* defination of knowledge_base_head */
knowledge_base_head(Q):-knowledge_base(Q).
knowledge_base_head(Q):-knowledge_base(Q:-T).

/* Qh:to finding the head clause of the query clause */
queryhead(H):-query(H).
queryhead(H):-query(H:-T).

/* new defination of query_clause */
query_clause(Q:-true):-
        query(Q).
query_clause(Q:-T):-
        query(Q:-T).

/* new defination of fact_base_clause */
fact_base_clause(K,true):-
        fact_base(K).
fact_base_clause(K,T):-
        fact_base(K:-T).


/*  Procedure FG: to format goal */
format_goal([]:-Body,Body):-!.
format_goal(Head:-Body,Goal):-
        !,
        convert(Head,Nothead),
        appendbody(Nothead,Body,Goal),!.
format_goal(Head,Goal):-
        convert(Head,Goal),!.

rightside(X,Y):-format_goal(X,Y).
```

```
already_accepted_subs(Option,Goal,Reason,[P,R,T,Diff]):-
   already_accepted_subs1(Diff,Newdiff),
   accept_subs(Option,Goal,Reason,[P,R,T,Newdiff]).

already_accepted_subs1(Diff,Newdiff):-
   exists(accepted_substitution(S)),
   already_accepted_subs2(Diff,S,Newdiff).
already_accepted_subs1(Diff,Diff):-
   not_exists(accepted_substitution(S)).

already_accepted_subs2([],S,[]).
already_accepted_subs2([H|T],S,NewT):-
   already_accepted_subs3(H,S,H1),
   already_accepted_subs2(T,S,T1),
   insertfront(H1,T1,NewT),!.

already_accepted_subs3(H,[[P,R,Mis,[H|T]]|Tail],[]):-!.
already_accepted_subs3([S,Q],[[P,R,Mis,[[Q,S]|T]]|Tail],[]):-!.
already_accepted_subs3(H,[[P,R,Mis,[H1|T]]|Tail],T1):-
   already_accepted_subs3(H,[[P,R,Mis,T]|Tail],T1).
already_accepted_subs3(H,[[P,R,Mis,[]]|Tail],T1):-!,
   already_accepted_subs3(H,Tail,T1).
already_accepted_subs3(H,[],[H]).

insertfront([],T,T):-!.
insertfront([H],T,[H|T]).

accept_subs(Option,Goal,Reason,[P,R,T,[]]):-!.
accept_subs(Option,Goal,Reason,[P,R,T,Diff]):-
   accept_substitution(Option,Goal,Reason,[P,R,T,Diff]).
```

```
/* acceptance enquiry of the whole suggestion */
acceptance(Option,Reason,Goal,Listquant):-
        acceptance1(Listquant),
        print_acceptance(Option,Reason,Goal,Listquant),
        acceptance_action,
        eliminate_failure_atom,!.

acceptance1(Listquant):-
        exists(subsidiary_list(S)),
        nl,write(' If the following clause'),
        test_is(S),write(' true'),
        nl,write_reason(S),nl,
        fail.
acceptance1(Listquant):-
        exists(skolemfq_list(Sq)),
        test_and_if(subsidiary_list(S)),
        print_quantifier(Sq,Listquant),nl,
        fail.
acceptance1(Listquant):-
        exists(accepted_substitution(S)),
        nl,print_accepted_substitution,nl,
        fail.
acceptance1(Listquant).

/* action taken on the acceptance of the suggestion */
acceptance_action:-
        retract(subsidiary_list(S)),
        assert_subsidiary(S),
        fail.
acceptance_action:-
        retract(accepted_substitution(E)),
        assert_atom_equiv22(E),
        fail.
acceptance_action.
```

```
/* to ask for an agreement of the suggested substitutions */
accept_substitution(Option,Goal,Reason,S):-
   print_substitute([S]),
   tab(40),write('---Do you agree ? '),
   get0(X),
   substitution_response(X,Option,Goal,Reason,S),nl.

/* analysizing the accept_substitution response */
substitution_response(10,Option,Goal,Reason,S):-
   /*return(<nl>)  10: to accept the suggestion */
   !, assert_accepted_substitution(S).
substitution_response(121,Option,Goal,Reason,S):-
   /* y(es) 121: to accept the suggestion */
   !,get0(_),
   assert_accepted_substitution(S).
substitution_response(97,_,_,_,_):-
   /* a(bort) 97: to abort the session */
   !,get0(_),
   abort.
substitution_response(98,Option,Goal,Reason,S):-
   /* b(reak) 98:to break from the session */
   !,get0(_),
   break,
   prompt(Old,' '),
   accept_substitution(Option,Goal,Reason,S).
substitution_response(119,Option,Goal,Reason,S):-
   /* w(hy) 119: to show why it was is suggested */
   !,get0(_),
   print_why_substitute([S]),
   nl,write(' ...So,'),
   accept_substitution(Option,Goal,Reason,S).
substitution_response(116,Option,Goal,Reason,S):-
   /* t(ype) 116:to type again the assumption */
   !,get0(_),
   write_haveproved(Option,Goal,Reason),
   print_accepted_substitution,
   accept_substitution(Option,Goal,Reason,S).
substitution_response(110,_,_,_,S):-
   /* n(o) 110:not to accept the suggestion */
   !,get0(_),
   assert(reject(S)),
   fail.
substitution_response(114,_,_,_,[P,R,T,Diff]):-
   /* r(eject) 114:to reject all substitutions of R */
   !,get0(_),
   assert(reject([P,R,T,Diff])),
   retract(mismatch_pair([P1,R,T1,Diff1],N)),
   assert(reject([P1,R,T1,Diff1])),
   fail.
substitution_response(115,Option,Goal,Reason,[P,R,T,Diff]):-
   /* s(how) 115: to show the effect of the failure atoms */
   !,get0(_),
   print_tree_failure(Option,[R]),
   accept_substitution(Option,Goal,Reason,[P,R,T,Diff]).
```

```
substitution_response(108,Option,Goal,Reason,S):-
   /* l(isting) 108: listing other possibilities */
   !,get0(_),
   print_other_substitution(S),
   nl,write(' ...So,'),
   accept_substitution(Option,Goal,Reason,S).
substitution_response(X,Option,Goal,Reason,S):-
   /* others(help): to print the menu */
   get0(_),!,
   nl,tab(4),write('Response options'),
   nl,tab(4),write('================'),nl,
   nl,tab(8),write('a      abort'),
   nl,tab(8),write('b      break'),
   nl,tab(8),write('n      not accepting and try others '),
   nl,tab(8),write('l      list other poss substitution(s)'),
   nl,tab(8),write('r      reject all poss. substitution(s)'),
   nl,tab(8),write('s      show the failure path of the goal'),
   nl,tab(8),write('t      type again the assumption'),
   nl,tab(8),write('w      why the substitution is suggested'),
   nl,tab(4),write('y or <nl>      yes (accepting) '),
   nl,tab(4),write('<others>       this message'),
   nl,accept_substitution(Option,Goal,Reason,S).

/* asserting accepted_substitution */
assert_accepted_substitution(S):-
   retract(accepted_substitution(OldS)),
   assert(accepted_substitution([S|OldS])),!.
assert_accepted_substitution(S):-
   assert(accepted_substitution([S])).

/* printing accepted substitution */
print_accepted_substitution:-
   exists(accepted_substitution(S)),
   nl,nl,tab(5),write('... so you have already agreed that: '),
   print_accepted_substitution1(S),!.
print_accepted_substitution.

print_accepted_substitution1([]):-
   nl.
print_accepted_substitution1([Head|Tail]):-
   nl,tab(5),
   write_subs_diff(Head),
   print_accepted_substitution1(Tail),!.
```

```
/* to list other possibility of substitutions of R */
print_other_substitution([P,R,T,Diff]):-
   exists(mismatch_pair([_,R,_,_],N)),
   nl,nl,tab(5),
   write('The other possible substitution of '),
   write_quote(R),
   write_plural_clause(mismatch_pair([_,R,_,_],N)),
   write(' as follows: '),
   print_other_substitution1(R),!.
print_other_substitution([P,R,T,Diff]):-
   nl,nl,tab(5),
   write('Sorry, no other possible substitution of '),
   write_quote(R),nl.

print_other_substitution1(R):-
   mismatch_pair([G,R,M,Diff],N),
   nl,tab(6),write('**'),
   write_subs_diff2(Diff,Diff1),
   write(' can also be replaced with '),
   write_subs_diff3(Diff1),write_respective(Diff1),
   write(' as exists '),
   print_exists_fact(M),
   fail.
print_other_substitution1(R):-
   nl.

/* printing the comment of why the substitution is suggested */
print_why_substitute([[G,R,M,Diff]|Tail]):-
   nl, tab(5),write_quote(R),write(' fails ,but exists '),
   print_exists_fact(M),
   nl,to_continue(Tail),
   print_why_substitute(Tail),!.
print_why_substitute([]):-!,
   nl.

/* to print the comment of substitution of failure atoms */
print_substitute([H|Tail]):-
   tells(F),write_subs(F,H),
   print_substitute(Tail).
print_substitute([]):-nl.

write_subs(user,Sublist):-!,
   nl, tab(5),write('If '),
   write_subs_diff(Sublist).
write_subs(F,H):-
   nl,write_subs(user,H),tell(F),nl,write_subs(user,H).

write_subs_diff([q,R,[(M:-T)|Tail],Diff]):-!,
   write_subs_diff1(M,Diff).
write_subs_diff([k,R,Mis,Diff]):-!,
   write_subs_diff1(R,Diff).
```

```
write_subs_diff1(FClause,[[S,Q]]):-!,
   write_quote(Q),
   write(' of the question\'s clause '),write_quote(FClause),
   write(' is substituted with '),write_quote(S),nl.
write_subs_diff1(FClause,[[S,Q]|T]):-
   write_subs_diff2([[S,Q]|T],Slist),
   write(' of the question\'s clause '),write_quote(FClause),
   write(' are substituted with '),
   write_subs_diff3(Slist),write(' respectively'),nl.

write_subs_diff2([[S,Q]],[S]):-!,
   write_quote(Q).
write_subs_diff2([[S,Q]|T],[S|S1]):-
   write_quote(Q),
   write(' AND  '),write_subs_diff2(T,S1).

write_subs_diff3([S]):-!,
   write_quote(S).
write_subs_diff3([S|S1]):-
   write_quote(S),
   write('AND  '),write_subs_diff3(S1).

print_exists_fact([]):-nl.
print_exists_fact([M|T]):-
   nl,tab(12),print_head_tail(M),
   write_and_nl(T),
   print_exists_fact(T).
```

```
/* asserting the subsidiary list */
assert_subsidiary_list(S):-
        assert_subsidiary_list1(S).
assert_subsidiary_list(S) :-
        retract(subsidiary_list([S|S1])),
        assert_subsidiary_list2(S1),
        !,fail.

assert_subsidiary_list1(S):-
        retract(subsidiary_list(S1)),
        assert(subsidiary_list([S|S1])),!.
assert_subsidiary_list1(S):-
        assert(subsidiary_list([S])),!.

assert_subsidiary_list2([]):-
        !,fail.
assert_subsidiary_list2(X):-
        assert(subsidiary_list(X)).

/* asserting skolemfq's list */
assert_skolemfq_list(S):-
        retract(skolemfq_list(S1)),
        check_duplicate(S,S1,S2),
        assert(skolemfq_list(S2)),!.
assert_skolemfq_list(S):-
        assert(skolemfq_list(S)),!.

check_duplicate([S],S1,S1):-
        a_member(S,S1),!.
check_duplicate(S,S1,S2):-
        append(S,S1,S2),!.

a_member(X,[X|Z]):-!.
a_member(X,[Y|Z]):-a_member(X,Z).

/* to assert subsidiary clauses after agreeing it */
assert_subsidiary([H|T]):-
        clause(subsidiary(H),true),
        assert_subsidiary(T),!.
assert_subsidiary([H|T]):-
        assert(subsidiary(H)),
        assert_subsidiary(T),!.
assert_subsidiary([]):-
        assert(newsubsidiary),!.

/* asserting predicate 'set_of_reason' */
assert_set_of_reason(G,Fc):-
        exists(set_of_reason(_,goal(G),reason(Fc))).
assert_set_of_reason(G,Fc):-
        not_exists(set_of_reason(_,goal(G),reason(Fc))),
        retract(option(Option)),
        Option1 is Option+1,
        assert(option(Option1)),
        assert(set_of_reason(option(Option),goal(G),reason(Fc)))
```

```
/* asserting atom_equiv22(list) */
assert_atom_equiv22([[Q,R,T,Diff]|Tail]):-
        assert(substitution_log(Diff)),
        assert_atomic_equiv22(Diff),
        assert_atom_equiv22(Tail).
assert_atom_equiv22([]).

assert_atomic_equiv22(Diff):-
        retract(atomic_equiv(OldDiff)),
        assert_atomic_equiv22a(Diff,OldDiff,NewDiff),
        assert(atomic_equiv(NewDiff)),!.
assert_atomic_equiv22(Diff):-
        assert(atomic_equiv(Diff)).

assert_atomic_equiv22a([[Qarg,Kbarg]|Tail],OldDiff,NewDiff):-
        check_member([[Qarg,Kbarg]],OldDiff,NewDiff1),
        assert_atomic_equiv22a(Tail,NewDiff1,NewDiff),!.
assert_atomic_equiv22a([],NewDiff,NewDiff).
```

```
/* to change the relevent quantifiers  */
quantifiers_out(Oldpc,Newpc,Newskquant):-
        retract(skolemfq_list(Sq)),
        subst_skolemr(Oldpc,Newskquant),
        change_quantifiers(Oldpc,Z1,Sq),
        subst_skolem(Z1,Newpc,Newskquant),
        !.
quantifiers_out(Oldpc,Newpc,Newskquant):-
        subst_skolem(Oldpc,Newpc,Newskquant),!.

/* to change relevent quantifiers of PC */
change_quantifiers((P<=>Q),(P1<=>Q1),Sq):-
        !,
        change_quantifiers(P,P1,Sq),
        change_quantifiers(Q,Q1,Sq).
change_quantifiers((P=>Q),(P1=>Q1),Sq):-
        !,
        change_quantifiers(P,P1,Sq),
        change_quantifiers(Q,Q1,Sq).
change_quantifiers(T:P,T:Q,Sq):-
        !,
        change_quantifiers(P,Q,Sq).
change_quantifiers((P&Q),(P1&Q1),Sq):-
        !,
        change_quantifiers(P,P1,Sq),
        change_quantifiers(Q,Q1,Sq).
change_quantifiers((P#Q),(P1#Q1),Sq):-
        !,
        change_quantifiers(P,P1,Sq),
        change_quantifiers(Q,Q1,Sq).
change_quantifiers((~P),(~P1),Sq):-
        !,
        change_quantifiers(P,P1,Sq).
change_quantifiers(all(X,P),Pc,Sq):-
        !,
        change_quantifiers(P,P1,Sq),
        change_quantifier(all(X,P1),Sq,Pc).
change_quantifiers(all(X,D,P),Pc,Sq):-
        !,
        change_quantifiers(P,P1,Sq),
        change_quantifier(all(X,D,P1),Sq,Pc).
change_quantifiers(exists(X,P),Pc,Sq):-
        !,
        change_quantifiers(P,P1,Sq),
        change_quantifier(exists(X,P1),Sq,Pc).
change_quantifiers(exists(X,D,P),Pc,Sq):-
        !,
        change_quantifiers(P,P1,Sq),
        change_quantifier(exists(X,D,P1),Sq,Pc).
change_quantifiers(exists_the(X,D,P),exists_the(X,D,P1),Sq):-
        !,
        change_quantifiers(P,P1,Sq).
change_quantifiers(exists_the(X,P),exists_the(X,P1),Sq):-
        !,
        change_quantifiers(P,P1,Sq).
change_quantifiers(P,P,Sq).
```

```
change_quantifier(Oldpc,Sq,Newpc):-
        arg(1,Oldpc,Varquant),
        nonvar(Varquant),
        not(atomic(Varquant)),
        functor(Varquant,fq,_),
        memberfq(Varquant,Sq),
        change_quantifier1(Oldpc,Newpc),!.
change_quantifier(Oldpc,Sq,Oldpc).

change_quantifier1(all(X,D,P=>Q),exists(X,D,P&Q)):-!.
change_quantifier1(exists(X,D,P&Q),all(X,D,P=>Q)):-!.
change_quantifier1(all(X,P=>Q),exists(X,P&Q)):-!.
change_quantifier1(exists(X,P&Q),all(X,P=>Q)):-!.
change_quantifier1(all(X,P),exists(X,P)):-!.
change_quantifier1(exists(X,P),all(X,P)):-!.

memberfq(Var,[H|T]):-
        functor(Var,fq,N),
        functor(H,fq,N),
        arg(1,Var,Skolemization),
        arg(1,H,Skolemization),!.
memberfq(Var,[H|T]):-
        memberfq(Var,T).
```

```
/* to equate the variables of original and current goals */
equatevars(Goal,Originalgoal):-
        equate_vars(Goal,Originalgoal),!.

equate_vars(P<=>Q,P1<=>Q1):-
        !,
        equate_vars(P,P1),
        equate_vars(Q,Q1).
equate_vars(P=>Q,P1=>Q1):-
        !,
        equate_vars(P,P1),
        equate_vars(Q,Q1).
equate_vars(P&Q,P1&Q1):-
        !,
        equate_vars(P,P1),
        equate_vars(Q,Q1).
equate_vars(P#Q,P1#Q1):-
        !,
        equate_vars(P,P1),
        equate_vars(Q,Q1).
equate_vars(~P,~P1):-
        !,
        equate_vars(P,P1).
equate_vars(all(X,P),all(Y,P1)):-
        !,
        equate_vars1(X,Y),
        equate_vars(P,P1).
equate_vars(all(X,D,P),all(Y,D1,P1)):-
        !,
        equate_vars1(X,Y),
        equate_vars(P,P1).
equate_vars(exists(X,P),exists(Y,P1)):-
        !,
        equate_vars1(X,Y),
        equate_vars(P,P1).
equate_vars(exists(X,D,P),exists(Y,D1,P1)):-
        !,
        equate_vars1(X,Y),
        equate_vars(P,P1).
equate_vars(P,P1).

equate_vars1(X,Y):-
        var(X),var(Y),
        X=Y,!.
equate_vars1(X,Y).
```

```
/* finding the differences between R and Q */
find_difference(R,D):-
        f_list(R,Rlist),
        find_difference0(R,Rlist,D).

/* finding the differences between R and Q */
find_difference0(R,Rlist,[H|Tail]):-
        find_difference1(R,Rlist,H).
find_difference0(R,Rlist,[H|Tail]):-
        find_difference0(R,Rlist,Tail),!.
find_difference0(R,Rlist,[]):-!.

find_difference1(R,Rlist,[G,(M:-T)]):-
        test_reason_tail(R,T),
        f_list(M,Mlist),
        find_difference2(G,Rlist,Mlist,D),
        length(D,Len),
        assert_mismatch_pair([G,R,(M:-T),D],Len),
        !,fail.

find_difference2(q,Rlist,Mlist,D):-!,
        find_difference3(Mlist,Rlist,D).
find_difference2(k,Rlist,Mlist,D):-!,
        find_difference3(Rlist,Mlist,D).

find_difference3([Rhead|Rtail],[Qhead|Qtail],D1):-
        Rhead\=Qhead,
        not_contain_fq2(Rhead),
        find_difference3(Rtail,Qtail,D),
        check_member([[Qhead,Rhead]],D,D1),
        !.
find_difference3([Rhead|Rtail],[Qhead|Qtail],D):-
        Rhead=Qhead,
        find_difference3(Rtail,Qtail,D),!.
find_difference3([],[],[]):-!.

assert_mismatch_pair([G,R,Mis,Diff],Len):-
        retract(mismatch_pair([G,R,X,Diff],Len)),
        assert(mismatch_pair([G,R,[Mis|X],Diff],Len)).
assert_mismatch_pair([G,R,Mis,Diff],Len):-
        not_exists(mismatch_pair([G,R,X,Diff],Len)),
        assert(mismatch_pair([G,R,[Mis],Diff],Len)).
```

```
/* to list all reasons */
list_reason(Option,Goal,Reason):-
   write_haveproved(Option,Goal,Reason),
   do_you_like_to_continue(Option,Goal,Reason),
   assertz_new(new_goal(Goal)),
   !.

/* to print an introduction of the suggestion */
write_haveproved(Option,Goalpc,Reason):-
   not_exists(toptry0(Goalpc)),
   tells(F),write_order(F,Option,Goalpc,Reason,S),!.
write_haveproved(Option,Goalpc,Reason):-
   exists(toptry0(Goalpc)),
   tells(F),write_already(F,Option,Goalpc,Reason,S),!.

write_order(user,Option,Goalpc,Reason,S):-!,
   nl,nl,write('OPTION: '),write(Option),
   nl,write('  The goal :'), nl,tab(10),write(' \" '),
   write_goal(Goalpc,S),write(' \"'),nl,
   write('fails due to the failure of the following clause:'),
   nl,write_reason0(Reason),
   nl,write('  However, we may be able to prove the goal'),
   nl,write('  after doing some corrections or additions'),nl.
write_order(F,Option,Goalpc,Reason,S):-
   write_order(user,Option,Goalpc,Reason,S),
   nl,tell(F),write_order(user,Option,Goalpc,Reason,S).

write_already(user,Option,Goalpc,Reason,S):-!,
   nl,nl,write('OPTION: '),write(Option),
   nl,write(' We have already proved '),nl,tab(10),
   write(' \" '), write_goal(Goalpc,S),write(' \" '), nl,
   write('It fails due to the failure of the following clause:'),
   nl,write_reason0(Reason),
   nl,write(' However, we may be able to reprove it again'),
   nl,write(' after doing some corrections or additions '),nl.
write_already(F,Option,Goalpc,Reason,S):-
   write_already(user,Option,Goalpc,Reason,S),
   nl,tell(F),write_already(user,Option,Goalpc,Reason,S).

write_goal(Goalpc,S):-
   form_of_answer(Form),
   write_goalform(Form,Goalpc,S).

write_goalform(pc,Goalpc,S):-
   !,write(Goalpc).
write_goalform(eng,Goalpc,S):-
   nonvar(S),
   writelist(S),!.
write_goalform(eng,Goalpc,S):-
   var(S),
   statement_phrase_one(Goalpc,S,[]),
   writelist(S).
```

```
/* analysizing the do_you_like_to_continue response */
do_you_like_to_continue(Option,Goal,Reason):-
   nl,tab(5),write(' --Do you like to continue ? '),
   get0(X),
   continue_response(X,Option,Goal,Reason).

continue_response(121,_,_,_):-
   /* y(es) 121: to continue and see  the suggestion */
   !,get0(X).
continue_response(10,_,_,_):-
   /* <nl> = 10: to continue and see the suggestion */
   !.
continue_response(97,_,_,_):-
   /* a(bort) 97: to abort the session */
   get0(X),
   abort.
continue_response(98,Option,Goal,Reason):-
   /* b(reak) 98: to abort the session */
   !,get0(X),
   break,
   prompt(Old,' '),
   do_you_like_to_continue(Option,Goal,Reason).
continue_response(115,Option,Goal,Reason):-
   /* s(how) 115: to show the effect of the failure atoms */
   !,get0(X),
   print_tree_failure(Option,Reason),
   do_you_like_to_continue(Option,Goal,Reason).
continue_response(116,Option,Goal,Reason):-
   /* t(ype) 116:to type again the assumption */
   !,get0(_),
   write_haveproved(Option,Goal,Reason),
   do_you_like_to_continue(Option,Goal,Reason).
continue_response(108,Option,Goal,Reason):-
   /* l(ist) 108:to list others set_of_reason */
   !,get0(_),
   print_others_setofreason,
   do_you_like_to_continue(Option,Goal,Reason).
continue_response(110,Option,_,_):-
   /* n(o) 110:not to accept the suggestion */
   get0(_),
   retractall(rsf_head(Option,_,_)),
   retractall(reason_son_father(Option,_,_)),
   !, fail.
continue_response(X,Option,Goal,Reason):-
   /* others(help) : to print the summary of responses */
   get0(_),!,
   nl,tab(4),write('Response options'),
   nl,tab(4),write('================'),nl,
   nl,tab(8),write('a       abort'),
   nl,tab(8),write('b       break'),
   nl,tab(8),write('n       not accepting and try other alternative'

   nl,tab(8),write('l       list other set of reason , if exists '),
   nl,tab(8),write('s       show the effect of failure atoms'),
   nl,tab(8),write('t       type again the introduction'),
   nl,tab(4),write('y or <nl>       yes to see the suggestion(s)'),
   nl,tab(4),write('<others>        this message'),
   nl,nl,
   do_you_like_to_continue(Option,Goal,Reason).
```

```
/* printing others set of reason */
print_others_setofreason:-
   exists(set_of_reason(_,_,_)),
   nl,tab(2),
   write('The others set of reason clauses'),
   write_plural_clause(set_of_reason(_,_,_)),
   write(' as follows:'),nl,
   print_others_setofreason1,!.
print_others_setofreason:-
   nl,nl,tab(2),
   write('Sorry, no other set of reason clause'),nl.

print_others_setofreason1:-
   set_of_reason(option(O),goal(G),reason(R)),
   nl,tab(2),write('OPTION '),write(O),write(': '),
   write('SET OF REASON  = '),write(R),
   nl,tab(10),write('THE FAILURE GOAL : '), write_quote(G),
   fail.
print_others_setofreason1:-
   nl,nl.

/* a new defination of printing failure atoms one */
write_reason0([H|T]):-
   exists(dup_reason(H)),!,
   write_reason0(T).
write_reason0([H|T]):-
   nl,tab(10),write(H),
   assert(dup_reason(H)),
   write_reason0(T).
write_reason0([]):-
   !,abolish(dup_reason,1),nl.
```

```
/* Procedure FINDMATCH */
mismatch_clause(R,Q,[q,(M:-T)]):-
        /* the mismatch clause(Q) is a query clause */
        query_clause(M:-T),
        Q\=R.
mismatch_clause(R,M,Rule):-
        /* Procedure MISMATCH.2 */
        convert(R,NotR),
        queryhead(NotR),
        mismatch_clause1(R,M,Rule).

mismatch_clause1(R,M,[k,(M:-T)]):-
        /* to find a rule which have the same
                predicate name but different arguments */
        fact_base_clause(M,T),
        R\=M.
mismatch_clause1(R,M,Rule):-
        /* to find a rule which have a different
                predicate name but the same arguments */
        functor1(R,Pred_name,N),
        f_list(R,[Pred_name|Args]),
        mismatch_clause1(Pred_name,N,R,Args,Rule).

mismatch_clause1(Pred_nameR,N,R,Args,[k,(M:-T)]):-
        /* Pred_nameR is not in the form of
                either f, ~f or not_f */
        f\==Pred_nameR,
        not_f\==Pred_nameR,
        ~f\==Pred_nameR,!,
        fact_base(K),
        a_rule(K,[M,T]),
        functor1(M,Pred_nameM,N),
        Pred_nameM\==Pred_nameR,
        f_list(M,[Pred_nameM|Args]).
mismatch_clause1(Pred_name,N,R,[X|Args],[k,(M:-T)]):-
        /* Pred_name = f or ~f or not_f */
        arg1(1,R,X),
        f_list(M,[Pred_name,Y|Args]),
        fact_base_clause(M,T),
        Y\==X.
```

```
/* to print the response of the suggestion */
print_acceptance(Option,Reason,Goal,Listquant):-
        tells(F),write_in(F,assumption),
        get0(X),
        tells(F),write_in(F,X,response),
        acceptance_response(X,Option,Reason,Goal,Listquant),
        nl.

/* checking the response */
acceptance_response(10,Option,_,_,_):-
        /* <nl>(return) 10: to accept the suggestion */
        !,
        retractall(rsf_head(Option,_,_)),
        retractall(reason_son_father(Option,_,_)),
        abolish(mismatch_pair,2).
acceptance_response(121,Option,_,_,_):-
        /* y(es) 121: to accept the suggestion */
        !,get0(_),
        retractall(rsf_head(Option,_,_)),
        retractall(reason_son_father(Option,_,_)),
        abolish(mismatch_pair,2).
acceptance_response(97,_,_,_,_):-
        /* a(bort) 97: to abort the session */
        !,get0(_),
        abort.
acceptance_response(98,Option,Reason,Goal,Listquant):-
        /* b(reak) 98: to break the session */
        !,get0(_),
        break,
        prompt(Old,' '),
        print_acceptance(Option,Reason,Goal,Listquant).
acceptance_response(115,Option,Reason,Goal,Listquant):-
        /* s(how) 115: to show the effect of the failure atoms *

        !,get0(_),
        print_tree_failure(Option,Reason),
        print_acceptance(Option,Reason,Goal,Listquant).
acceptance_response(119,Option,Reason,Goal,Listquant):-
        /* w(hy) 119: why it was suggested **/
        !,get0(_),
        complete_resp_why,
        print_acceptance(Option,Reason,Goal,Listquant).
acceptance_response(116,Option,Reason,Goal,Listquant):-
        /* t(ype) 116:to type again the assumption */
        !,get0(_),
        write_haveproved(Option,Goal,Reason),
        acceptance1(Listquant),
        print_acceptance(Option,Reason,Goal,Listquant).
acceptance_response(110,_,_,_,_):-
        /* n(o) 110 :not to accept the suggestion */
        !,get0(_),
        abolish(subsidiary_list,1),
        abolish(skolemfq_list,1),
        fail.
```

```
acceptance_response(X,Option,Reason,Goal,Listquant):-
        /* others(help) : to print the summary of responses */
        get0(_),!,
        nl,tab(4),write('Response options'),
        nl,tab(4),write('================'),nl,
        nl,tab(8),write('a      abort'),
        nl,tab(8),write('b      break'),
        nl,tab(8),write('n      not accepting and try others'),
        nl,tab(8),write('s      show the failure paths'),
        nl,tab(8),write('t      type again the assumption'),
        nl,tab(8),write('w      why the substitution is accepted

        nl,tab(4),write('y or <nl>      yes (accepting) '),
        nl,tab(4),write('<others>       this message'),
        print_acceptance(Option,Reason,Goal,Listquant).

complete_resp_why:-
        exists(accepted_substitution(E)),
        print_why_substitute(E),
        fail.
complete_resp_why:-
        exists(skolemfq_list(Sq)),
        nl,tab(5),
        write('...because the failure clause(s)'),
        write('  contained Skolem\'s function'),
        nl,tab(5),write('i.e '),write_reason(Sq),nl,
        fail.
complete_resp_why:-
        exists(subsidiary_list(S)),
        nl,tab(5),
        write('...because the following clause'),
        write_does(S),
        write(' not exist in the database:'),
        nl,write_reason(S),
        fail.
complete_resp_why.
```

```
/* to print the proving tree of each failure atom */
print_tree_failure(Option,[Son|T]):-
        print_tree_failure0(Option,Son),
        to_continue(T),
        print_tree_failure(Option,T),!.
print_tree_failure(Option,[]):-!,
        nl.

print_tree_failure0(Option,Son):-
        rsf_head(Option,Son,Father),
        print_tree_failure01(Option,Son,Father),
        fail.
print_tree_failure0(Option,Son):-!.

print_tree_failure01(Option,Son,Father):-
        nl,tab(4),write(Son),write(' fails '),nl,tab(10),
        write('===>   '),write(Father),write('    fails'),
        print_tree_failure1(Option,Father),
        !.

print_tree_failure1(Option,goal(Pc)):-!,
        nl,nl.
print_tree_failure1(Option,(H:-T)):-!,
        reason_son_father(Option,H,Father), nl,tab(10),
        write('===>   '),write(Father),write('    fails'),
        print_tree_failure1(Option,Father).
print_tree_failure1(Option,H):-
        reason_son_father(Option,H,Father), nl,tab(10),
        write('===>   '),write(Father),write('    fails'),
        print_tree_failure1(Option,Father).

to_continue([]):-!.
to_continue(T):-
        tab(4),write('Press <return>!!'),
        get0(Press),(Press=10;get0(_)),!.
```

```
/* SSr:to equate E with S in X */
subst_skolemr(X,[]):-!.
subst_skolemr(X,[[E,S]|T]):-
        E=S,
        subst_skolemr(X,T),!.

/* SS:substitute S in X with a variable becomes Y*/
subst_skolem(X,X,[]):-!.
subst_skolem(X,Y,[[E,S]|T]):-
        substitute(Z,S,X,X1), /* Z replaces E */
        subst_skolem(X1,Y,T),!.

/* SSN: to substitute S in X with a variable and becomes Y
        and replacing sign ~ between E and S */
subst_skolem_not(X,X,[]):-!.
subst_skolem_not(X,Y,[[E,S]|T]):-
        replacing(E,S,E1,S1),
        substitute(E1,S1,X,X1),
        subst_skolem_not(X1,Y,T),!.

/* SSc: */
subst_skolemc(X,X,[]):-!.
subst_skolemc([],[],E):-!.
subst_skolemc([X|Y],[Xnew|Ynew],E):-
        subst_skolem(X,Xnew,E),
        subst_skolemc(Y,Ynew,E).

/* new defination using Newskquant */
subst_skolemr(X,[]):-!.
subst_skolemr(X,[[E,S,Quant]|T]):-
        E=S,
        subst_skolemr(X,T),!.

subst_skolem(X,X,[]):-!.
subst_skolem(X,Y,[[E,S,Quant]|T]):-
        substitute(Z,S,X,X1), /* Z replaces E */
        subst_skolem(X1,Y,T),!.

subst_skolem_not(X,X,[]):-!.
subst_skolem_not(X,Y,[[E,S,Quant]|T]):-
        replacing(E,S,E1,S1),
        substitute(E1,S1,X,X1),
        subst_skolem_not(X1,Y,T),!.

/* replacing "not_"  with "~"  */
replacing(~ E,~ S, E, S):-!.
replacing( E,~ S,~ E, S):-!.
replacing( E, S, E, S).
```

```
replacing(X,~Y):-
        functor(X,F,N),
        name(F,Flist),
        append([110,111,116,95],Flist1,Flist),!,
        name(F1,Flist1),
        X=..[F|A],
        Y=..[F1|A].
replacing(X,X).

/* NNfq: checking whether the reason predicate contains "fq" */
not_contain_fq(R):-
        functor1(R,F,N),
        not_contain_fq1(R,N).

not_contain_fq1(R,0):-!.
not_contain_fq1(R,N):-arg(N,R,Nth),
        not_contain_fq2(Nth),
        N1 is N-1,
        not_contain_fq1(R,N1).

not_contain_fq2(Nth):-var(Nth),!.
not_contain_fq2(Nth):-
        functor1(Nth,Fq,N1),Fq\==fq,
        not_contain_fq1(Nth,N1).

/* checking whether the reason predicate contained "fq" */
not_contained_fq(R,L):-
        functor1(R,F,N),
        not_contained_fq1(F,R,L).

not_contained_fq1(fq,R,[R]):-!.
not_contained_fq1(F,R,L):-
        f_list(R,[F|Arguments]),
        not_contained_fq2(Arguments,L).

not_contained_fq2([],[]):-!.
not_contained_fq2([A1|Tail],L):-
        not_contained_fq3(A1,L1),
        not_contained_fq2(Tail,L2),
        append(L1,L2,L),!.

not_contained_fq3(Nth,[]):-
        var(Nth),!.
not_contained_fq3(Nth,L):-
        functor1(Nth,F,N1),
        not_contained_fq1(F,Nth,L).
```

```
setup_predicate(R,Q,N):-
        /* for predicates f(P,....)/not_f(P,....) */
        functor1(R,F,N),
        (F==f; F==not_f ;~f==F ),
        functor1(Q,F,N),
        arg1(1,R,P),
        arg1(1,Q,P1),
        P1=P,
        !.
setup_predicate(R,Q,N):-
        /* for predicates P(.,..) where P\=f or not_f */
        functor1(R,P,N),
        functor1(Q,P,N).

/* to check membership */
check_member([E2],E1,E1):-
        member(E2,E1),!.
check_member(E2,E1,E):-
        append(E2,E1,E),!.

not_rejected(R):-        /* collective test */
        not(reject(R)),!.

/* checking such that the reason is not same with one of
                the tail of substitution rule */
test_reason_tail(R,true):-!.
test_reason_tail(R,(H,T)):-
        R\=H,  /* have been changed to \= from \== */
        test_reason_tail(R,T),!.
test_reason_tail(R,T):-
        R\=T,!.  /* have been changed to \= from \== */

/* to eliminate a duplicate  */
eliminate_failure_atom:-
        retract(set_of_reason(Option,G,reason(Fc))),
        eliminate_failure_atom(Fc,Newfc),
        Newfc\==[],
        assert(temp_list(Option,G,Newfc)),
        fail.
eliminate_failure_atom:-
        retract(temp_list(Option,G,Newfc)),
        assertz(set_of_reason(Option,G,reason(Newfc))),
        fail.
eliminate_failure_atom:-!.

eliminate_failure_atom([H|T],Newtail):-
        clause(subsidiary(H),_),
        eliminate_failure_atom(T,Newtail),!.
eliminate_failure_atom([H|T],[H|Newtail]):-
        eliminate_failure_atom(T,Newtail),!.
eliminate_failure_atom([],[]):-!.

/* defination of functor1 to accomodate ~f or notf(f) */
functor1(~F,~P,N):-!,functor(F,P,N).
functor1(F,P,N):-functor(F,P,N).
```

```
f_list(~M,[~H|T]):-!,
        M=..[H|T].
f_list(M,Mlist):-
        M=..Mlist.

arg1(N,~F,P):-!,arg(N,F,P).
arg1(N,F,P):-arg(N,F,P).

/* to copy a clause to another clause */
copy_clause(Clause1,Clause2):-
        call(Clause1),
        assertz_new(Clause2),
        fail.
copy_clause(Clause1,Clause2).

/* testing whether a given clause is a rule or an atom */
a_rule((M:-T),[M,T]):-!.
a_rule(M,[M,true]).
```

```
/* to retract a set of reason clauses */
why_it_fails(Originalgoal,Lisquant):-
   retract(set_of_reason(option(Option),goal(G),reason(R))),
   not_exists(reject(R)),
   assert(consulted_sor(G,R)),
   equatevars(G,Originalgoal),
   list_reason(Option,G,R),
   find_reason(Option,G,R,Lisquant).

/*to print the reason */
find_reason(Option,Goal,Reason,Lisquant):-
   find_mismatch_clause(Reason),
   suggestion(Option,Goal,Reason,Lisquant).
find_reason(Option,Goal,Reason,Lisquant):-
   assert(reject(Reason)),
   fail.

suggestion(Option,Goal,Reason,Lisquant):-
   process_reasons(Option,Goal,Reason,Reason),
   acceptance(Option,Reason,Goal,Lisquant).
suggestion(Option,Goal,Reason,Lisquant):-
   retract(accepted_substitution(S)),
   suggestion(Option,Goal,Reason,Lisquant).

process_reasons(Option,Goal,Reason,[H|T]):-
   process_suggestion(Option,Goal,Reason,H),
   process_reasons(Option,Goal,Reason,T),!.
process_reasons(Option,Goal,Reason,[]):-!.

process_suggestion(Option,Goal,Reason,R):-
   functor(R,F,N),
   N1 is N+1,
   ordered_mismatch_clause([P,R,T,Diff],N1,1),
   not_exists(reject([P,R,T,Diff])),
   not_exists(substitution_log(Diff)),
   already_accepted_subs(Option,Goal,Reason,[P,R,T,Diff]),!.
process_suggestion(Option,Goal,Reason,R):-
   extract_fq(R,L),
   process_others(R,L).

process_others(R,[]):-!,
   assert_told_list(R).
process_others(R,L):-
   assert_skolemfq_list(L),!.

assert_told_list(R):-
   assert_subsidiary_list(R).
```

```
find_mismatch_clause([R|T]):-
   setup_predicate(R,Q,N),
   fast_setof(D,mismatch_clause(R,Q,D),List),
   find_difference(R,List),
   find_mismatch_clause(T),!.
find_mismatch_clause([R|T]):-
   find_mismatch_clause(T),!.
find_mismatch_clause([]):-!.

ordered_mismatch_clause(Triplet,N,Ni):-
   retract(mismatch_pair(Triplet,Ni)).
ordered_mismatch_clause(Triplet,N,Ni):-
   Nn is Ni+1,
   Nn =< N,
   ordered_mismatch_clause(Triplet,N,Nn).

write_proved(user,X):- !,
   nl,write(' ****(rectifiers) PROVED:'),
   nl,tab(10),write(X),write(' ? ').
write_proved(F,X):-
   write_proved(user,X),
   tell(F),write_proved(user,X).

extract_fq(R,L):-not_contained_fq(R,L).
```

```
other_meaning:-
        form_of_answer(eng),!,
        write_meaning(user),
        get0(A),
        meaning_response(A).
other_meaning.

/* to print the response of the given answer
        for finding different meaning */
meaning_response(97):-
        !,get0(_),  /* a=97 -to abort */
        abort.
meaning_response(98):-
        !,get0(_),  /* b=98 -to break */
        break,
        prompt(Old,' '),
        other_meaning.
meaning_response(59):-
        /* ;=59 -finds other meaning, if exists */
        !,get0(_),
        fail.
meaning_response(121):-
        /* y=121 -finds other meaning, if exists */
        !,get0(_),
        fail.
meaning_response(10):-
        !.          /* <return>=10 : that's all folks */
meaning_response(110):-
        /* n=110- not to find other meaning*/
        !,get0(_).
meaning_response(A):-
        !,get0(_),  /* to print help's table */
        nl,tab(4),write('Response options'),
        nl,tab(4),write('================'),nl,
        nl,tab(8),write('a      abort'),
        nl,tab(8),write('b      break'),
        nl,tab(4),
        write('; or y           finds other meaning'),
        nl,tab(4),
        write('n or <nl>        not to find other meaning'),
        nl,tab(4),write('<others>       this message'),
        nl,nl,other_meaning.
```

```
/* Procedure PCFACT */
pcfact:-
        prompt(Old,' '),write(Old),
        read(S),
        assert_knowledge(S).

/* Procedure PCQUEST */
pcquest:-
        prompt(Old,' '),write(Old),
        read(Q),
        assertz_new(form_of_answer(pc)),
        question(Q).
```

```
/* Procedure RECTIFIERS  */
rectifiers(X,yes,Newskquant):-
        form_of_answer(Formanswer),
        print_answer1(Formanswer,X),
        get0(A),
        answer_response(A,X,Newskquant).
rectifiers(X,no,Newskquant):-
        why_it_fails(X,Newskquant),
        clear_pc1,
        !,fail.

/* PA1: print_answer1 */
print_answer1(pc,X):-
        write_proved(user,X).
print_answer1(eng,X):-
        write_more(user).

/* to print the response of the given answer */
answer_response(97,X,Newskquant):-
        !,get0(_),  /* a=97: to abort */
        abort.
answer_response(98,X,Newskquant):-
        !,get0(_),  /* b= 98: to break */
        break,
        prompt(Old,' '),
        rectifiers(X,yes,Newskquant).
answer_response(121,X,Newskquant):-
        !,get0(_),
        fail.
answer_response(59,X,Newskquant):-
        !,get0(_),
        fail.
answer_response(112,X,Newskquant):-
        !,get0(_),  /* p=112: to print the solution tree */
        print_solution,
        rectifiers(X,yes,Newskquant).
answer_response(10,X,Newskquant):-
        /* <return>=10 : that's all folks */
        !.
answer_response(110,X,Newskquant):-
        !,get0(_).  /* p=110: not to find other answers */
answer_response(A,X,Newskquant):-
        !,get0(_),  /* others : to print help's table */
        nl,tab(4),write('Response options'),
        nl,tab(4),write('================'),nl,
        nl,tab(8),write('a      abort'),
        nl,tab(8),write('b      break'),
        nl,tab(8),write('p      print solution\'s tree'),
        nl,tab(4),
        write('n or <nl>        not to find other answers'),
        nl,tab(4),
        write('y or ;           finds other answers'),
        nl,tab(4),write('<others>       this message'),
        nl,nl,rectifiers(X,yes,Newskquant).
```

```
/* Wpc: */
writepc(X,Y,Sk):-
        subst_skolem(X,Y,Sk),
        tells(F),writepc(F,Y),!.

writepc(user,Y):-!,
        write_quote(Y),nl,
        write('================================'),nl,nl.
writepc(F,Y):-
        writepc(user,Y),
        tell(F),writepc(user,Y).

/* WI-response: */
write_in(user,X,response):-!.
write_in(F,X,response):-
        tell(F),name(Y,[X]),write(Y),nl,nl,!.

/* WI-assumption: */
write_in(user,assumption):-
        nl, write(' Do you like to reprove again by using'),
        nl, write(' the above assumption(s)  ---? '),!.
write_in(F,assumption):-
        write_in(user,assumption),
        tell(F),write_in(user,assumption).

/* WI-if: */
write_in(user,R,if):-!,
        nl,write('if  '),
        print_sentence(R),!.
write_in(F,R,if):-
        write_in(user,R,if),
        tell(F),write_in(user,R,if).

/* WI-subsidiary: */
write_in(user,R,subsidiary):-!,
        nl,write('subsidiary('),write(R),write(').'),nl.
write_in(F,R,subsidiary):-
        write_in(user,R,subsidiary),
        tell(F),write_in(user,R,subsidiary).

/* WI-equiv: */
write_in(user,[R,Q],equiv):-
        print_message(Q,R),!.
write_in(F,[R,Q],equiv):-
        write_in(user,[R,Q],equiv),
        tell(F),write_in(user,[R,Q],equiv).


/* WA-yes: */
write_answer(user,yes):-!,
        nl,write('>>answer: Yes,'),nl,tab(4).
write_answer(F,yes):-write_answer(user,yes),
        tell(F),write_answer(user,yes).
```

```
/* WA-no: */
write_answer(user,no):-!,
        nl,write('>>answer:  No, it cannot prove :'),
        nl,write('     ').
write_answer(F,no):-write_answer(user,no),
        tell(F),write_answer(user,no).

/* WA-finish: */
write_answer(user,finish):-!,
        nl,nl,write('no more answers!'),nl.
write_answer(F,finish):-write_answer(user,finish),
        tell(F),write_answer(user,finish).

/* to print all the failure atoms or the reason */
write_reason([H|T]):-
        tells(F),write_reason1(F,H),
        write_reason(T).
write_reason([]):-!,nl.

write_reason1(user,X):- !,
        nl,tab(10),write(X).
write_reason1(F,X):-
        write_reason1(user,X),
        tell(F),write_reason1(user,X).

/* writing the answer as an english sentence */
write_sent(user,A,Ans):-
        nl,write('>> Answer: '),
        nl,write_rem(Ans),writelist(A),write('.'),
        nl,nl,!.
write_sent(F,A,Ans):-
        write_sent(user,A,Ans),
        tell(F),nl,write_sent(user,A,Ans),!.

/* writing remarks */
write_rem(yes):-write(' Yes, it is true that ').
write_rem(no):-write(' No, it is false that ').

/* to get more answers */
more_answers:-tells(F),write_more(F),other_answers.

write_more(user):-
        !,nl,write('** top(phrase): more answers? ').
write_more(F):-write_more(user),
        tell(F),nl,nl,write_more(user).

write_meaning(user):-!,nl,
        write('** top(phrase): more answers'),
        write(' with a different meaning ? ').
write_meaning(F):-write_meaning(user),
        tell(F),nl,nl,write_meaning(user).

/* writing a list as a sentence */
writelist([X|Y]):-write(X),write(' '),writelist(Y).
writelist([]).
```

```
/* printing a comment of replacing relevent quantifiers */
print_quantifier([Skhead|Sktail],Listquant):-
        print_quantifier1(Skhead,Listquant),
        print_quantifier(Sktail,Listquant).
print_quantifier([],Listquant).

print_quantifier1(Skhead,[[Vars,Sk,Quant]|T]):-
        Skhead = Sk,  /* modify on 0740 081285 */
        nl,tab(5),write(' ** " '),
        write(Quant),write('('),write(Vars),
        write(', ...) " is replaced with " '),
        print_quantifier2(Quant,Newquant),
        write(Newquant),
        write('('),write(Vars), write(', ...) " '),!.
print_quantifier1(Skhead,[H|T]):-
        print_quantifier1(Skhead,T),!.
print_quantifier1(Skhead,[]).

print_quantifier2(all,exists).
print_quantifier2(exists,all).

/*to write X in a quote  i.e "X" */
write_quote(X):-
        write(' \" '),write(X),write(' \" ').

/* to print respectively where appropriate */
write_respective([H]):-!.
write_respective(Diff):-write(' respectively ').

/* to print the conjuction 'and' */
test_and_if(S):-
        exists(S),
        nl,nl,tab(5),write('AND  if '),!.
test_and_if(S):-
        nl,nl,tab(5),write('If ').

/* to test the plurality of the sentence */
test_is([H]):-!,
        write(' is ').
test_is([H|T]):-
        write('s are ').

write_plural_list([L]):-
        length([L],N),
        write_plural(N).

write_plural_clause(X):-
        fast_setof(X,X,L),
        length(L,N),
        write_plural(N).

write_plural(1):-!,write(' is ').
write_plural(N):-write(' are ').  /* N>1 */

write_and_nl([]):-!.
write_and_nl(X):- nl,tab(8),write(' AND').
```

```
write_does([S]):-!,write(' does').
write_does(S):-write('s do').

/* to print a rule */
print_head_tail(Q:-true):-!,
        write_quote(Q).
print_head_tail(Q):-
        write_quote(Q).
```

```
/* Procedure QUEST */
question(X):-
        prompt(_,' '),
        clear_pc,
        nl,nl,write('NEXT QUESTION:'),nl,tab(4),write(X),nl,
        reset_newquery(X,Z),
        pc_to_hornclause(Z,Clause,Newskquant),
        retry_search(Z,Newskquant),
        answer_search(Clause,Z,Y,Newskquant),
        print_answer(Z,Y,Clause,Newskquant).

question(Z,Y,Clause,Newskquant):-
        pc_to_hornclause(Z,Clause,Newskquant),
        retry_search(Z,Newskquant),
        answer_search(Clause,Z,Y,Newskquant).


/* Procedure PC-HORN */
pc_to_hornclause(X,Clause,Newskquant):-
        nl,write('The translation of its negation:'),nl,
        list_quantifiers(X,Listquant),
        question_to_hornclause(~X,Clause,T,Sk),
        merge_quantifiers(Sk,Listquant,Newskquant),
        !.


/* Procedure RESET-QUERY: to set a query or
        to reset a new query and prove it */
reset_newquery(Oldquery,Oldquery).
reset_newquery(Oldquery,Newquery):-
        reset_newquery1(Oldquery,Newquery),
        print_newquery(Newquery).

reset_newquery1(Oldquery,Newquery):-
        retract(new_query(Newquery)),
        abolish(current_query,1),
        assert(current_query(Newquery)).
reset_newquery1(Oldquery,Newquery):-
        exists(new_query(_)),
        reset_newquery1(Oldquery,Newquery).
```

```
/* Procedure RESET-SEARCH: to reformulate new query
                          or to reprove again */
retry_search(Oldquery,Newskquant).
retry_search(Oldquery,Newskquant):-
        retry_search1(Oldquery,Newskquant).

retry_search1(Oldquery,Newskquant):-
        not_exists(atomic_equiv(_)),
        not_exists(skolemfq_list(_)),
        retract(newsubsidiary).
retry_search1(Oldquery,Newskquant):-
        reformulate_question(Oldquery,Newskquant),
        abolish(set_of_reason,3),
        abolish(reason_son_father,3),
        abolish(rsf_head,3),
        abolish(newsubsidiary,0),
        abolish(query,1),
        fail.
retry_search1(Oldquery,Newskquant):-
        ( exists(newsubsidiary);
                exists(skolemfq_list(_));
                exists(atomic_equiv(_)) ),
        retry_search1(Oldquery,Newskquant).


/* procedure REFORMULATEQ */
reformulate_question(Oldquery,Newskquant):-
        /* reformulation when atomic/predicate
           are equivalent and also if 'fq' exists */
        retract(atomic_equiv(Q)),
        new_goal(Newquery2),
        subst_skolem_not(Newquery2,Newquery1,Q),
        quantifiers_changing(Newquery1,Newquery,Newskquant),
        assert(new_query(Newquery)),
        !.
reformulate_question(Oldquery,Newskquant):-
        /* reformulation when only 'fq' exists */
        not_exists(atomic_equiv(_)),
        exists((skolemfq_list(Sq)),
        quantifiers_changing(Oldquery,Newquery,Newskquant),
        assert(new_query(Newquery)),
        !.

/* Procedure PRINT_ANSWER: printing the answer */
print_answer(Q,Q1,[],Newskquant):-
        /* the question clause is [],
                i.e the inconsistent clause */
        form_of_answer(Formanswer),
        affirm(Ans),
        answer_form(Formanswer,Q1,Q2,Ans),
        nl,nl,
        write('It fails because the question'),
        write('  clause is an inconsistent one!'),
        !,nl,nl,other_meaning.
print_answer(Q,Q1,Clause,Newskquant):-
        Clause\==[],
        print_answer0(Q,Q1,Newskquant).
```

```
print_answer0(Q,Q1,Newskquant):-
        /* either printing yes answers
                or finding the cause of failure
                if the answer is no */
        nonvar(Q1),
        affirm(Ans),
        form_of_answer(Formanswer),
        answer_form(Formanswer,Q1,A,Ans),
        !,rectifiers(Q1,Ans,Newskquant).
print_answer0(Q,Q1,Newskquant):-
        /* checking if we can find any other answers
                (proving tree) if the reason exists
                and that the answer is yes */
        affirm(yes),
        exists(set_of_reason(O,G,R)),
        rectifiers(Q,no,Newskquant).
print_answer0(Q,Q1,Newskquant):-
        /* finding more answers with
                 different meanings of the sentence */
        exists(toptry([])),
        not_exists(set_of_reason(_,_,_)),
        test_finish_fact(Q),
        !,nl,nl, other_meaning.

/* print new query */
print_newquery(Newquery):-
        /* if the top language is a predicate calculus */
        form_of_answer(pc),
        nl,nl,write('RE-QUESTION:'),
        nl,tab(4),write(Newquery),nl.
print_newquery(Newquery):-
        /* if the top language is an english */
        form_of_answer(eng),
        statement_phrase_one(Newquery,S,[]),
        nl,nl,write('RE-PHRASE:'),
        nl,tab(4), writelist(S),write(' ?'),nl,
        nl,nl,write('RE-QUESTION:'),
        nl,tab(4),write(Newquery),nl.

/* AF:answer_form */
answer_form(pc,Y,Y,Ans):-
        write_answer(user,Ans),
        writepc(user,Y).
answer_form(eng,Y,S,Ans):-
        statement_phrase_one(Y,S,[]),
        nl,write_sent(user,S,Ans).
```

```
/*  Procedure TFF */

test_finish_fact(Q):-
        affirm(yes),
        write_answer(user,finish),!.
test_finish_fact(Q):-
        affirm(no),
        nl,nl,tab(5), write('Do you like to assert '),
        nl,tab(8),write_quote(Q),
        nl,tab(5),write('as a fact in the database (y/n)? '),
        get0(X),
        ask_assert_fact(X,Q).
ask_assert_fact(121,Q):-
        /* y(es) 121: to assert Q in the database */
        get0(X),
        assert_knowledge(Q),!.
ask_assert_fact(110,Q):-
        /* n(o) 110: not to assert Q in the database */
        get0(X), !.
ask_assert_fact(X,Q):-
        /* others (help) */
        (X==10;get0(_)),!,
        nl,nl,tab(5),
        write(' type "y"(yes) or "n"(no) only please'),
        nl,tab(5),write(' type your response now ? '),
        get0(Y),
        ask_assert_fact(Y,Q).


quantifiers_changing(Old,New,Listquant):-
        quantifiers_out(Old,New,Listquant).
```

```
/* a new version of read_in in poplog (using name(X,Y))*/
/* Read a sentence from the terminal, and convert it into
   a list of atoms and integers

   Main predicate provided:

     read_in(P)  - Read a sentence and unify P with
                   the list of atoms/ integers
*/


read_in(P) :- initread(L), words(P,L,[]), !.

/* Get list of characters - everything up to a
   full stop, exclamation mark or question mark */

initread([K1,K2|U]) :- get(K1), get0(K2), readrest(K2,U).

readrest(46,[]) :- !.  /* "." */
readrest(63,[]) :- !.  /* "?" */
readrest(33,[]) :- !.  /* "!" */
readrest(K,[K1|U]) :- K<33, !, get(K1), readrest(K1,U).
readrest(K1,[K2|U]) :- get0(K2), readrest(K2,U).

/* Convert list of characters into a list of atoms and
        integers. This bit is written as Prolog grammar
        rules
*/

words([W|Ws],S0,S3) :-
   word(W,S0,S1), !, blanks(S1,S2), words(Ws,S2,S3).
words([],S,S).

word(W,[C|S0],S1) :-
   basic_character(C), !,
   alphanums(As,S0,S1),
   name(W,[C|As]) .
word(P,[C|S],S) :- name(P,[C]).

alphanums([A|As],S0,S2) :-
   alphanum(A,S0,S1), !,
   alphanums(As,S1,S2).
alphanums([],S,S).

alphanum(A,[C|S],S) :- lc(C,A), !.
alphanum(C,[C|S],S) :- digit(C), !.

rnumber(N1,N3,[C|S0],S1) :- digit(C), !,
   N2 is (C-48 )+(10*N1),
   rnumber(N2,N3,S0,S1).
rnumber(N,N,S,S).

blanks([C|S0],S1) :- blank(C), !,
   blanks(S0,S1).
blanks(S,S).

/* Basic Character types */
```

```
basic_character(C):- lc(C,X).
basic_character(C):- digit(C).

blank(X) :- X<33.

/** digit(X) :- 47<X, X<58.  **/

lc(X,X):-64<X,X<91.  /* retaining the capital letters */
lc(X,X) :- 96<X, X<123.
lc(95,95).  /* _=95 : to retain underscore
                      for denoting a variable */
```

```
/* reading an english sentence */
read_phrase(Y,Z):-read_in(X),get0(_),
        nl,write('NEXT PHRASE: '),
        nl,tab(4),writelist(X),
        nl,write('==============>'),
        append(Y,Z,X),Z\==[],
        !.      /* to prevent analysizing all words */

/* asserting a knowledge or answering a question */
stat_or_quest(Y,Z):-
        assertz_new(form_of_answer(eng)),
        question_phrase(Q,Y,[]),
        question(Q).
stat_or_quest(Y,Z):-
        Z\==[?],
        statement_phrase(S,Y,[]),
        assert_knowledge(S).
stat_or_quest(Y,[?]):-
        assertz_new(form_of_answer(eng)),
        sentence(Q,Y,[]),
        question(Q).

statement_phrase_one(Q,A,[]):-
        /* to regenerate one sentence only */
        statement_phrase(Q,A,[]),!.
```

```
/* The TRACING TECHNIQUE  */

/********** SENTENCE **********/

sentence(Z,Y,X,W):-
        statement_phrase(Z,Y,X,W).
sentence(Z,Y,X,W):-
        question_phrase(Z,Y,X,W).

/********** STATEMENT_PHRASE **********/

statement_phrase(Z,[s3,Y,X],W,V):-
        noun_phrase(U,T,S,Z,Y,W,R),
        cp_verb_phrase(U,T,S,X,R,V).
statement_phrase(Z,[s4,Y,X],W,V):-
        noun_phrase_not(U,T,S,Z,Y,W,R),
        cp_verb_phrase_not(U,T,S,X,R,V).

/********** CP_CLASS_NAME **********/

cp_class_name(Z,Y,X,[ccn0,W],V,U):-
        class_name(Z,Y,X,W,V,U).
cp_class_name(Z,Y,X,[ccn1,W,[V,U],T],[S|R],Q):-
        conj_pair(S,U),
        class_name(Z,Y,P,W,R,O),
        conjunction(P,N,X,[V,U],O,M),
        cp_class_name(Z,Y,N,T,M,Q).
cp_class_name(Z,Y,X,[ccn2,W,V,U],T,S):-
        class_name(Z,Y,R,W,T,Q),
        conjunction(R,P,X,V,Q,O),
        cp_class_name(Z,Y,P,U,O,S).

/********** CLASS_NAME **********/

class_name(singular,Z,Y,[cn1|X],[a|W],V):-
        gnoun(singular,Z,Y,X,W,V).
class_name(singular,Z,Y,[cn2|X],[an|W],V):-
        gnoun(singular,Z,Y,X,W,V).
class_name(plural,Z,Y,[cn3|X],W,V):-
        gnoun(plural,Z,Y,X,W,V).
class_name(Z,Y,X,[cn4|W],V,U):-
        gadjective(Z,Y,X,W,V,U).

/********** CP_NOUN_PHRASE **********/

cp_noun_phrase(Z,Y,X,W,[cnp0,V],U,T):-
        noun_phrase(Z,Y,X,W,V,U,T).
cp_noun_phrase(Z,Y,X,W,[cnp1,V,[U,T],S],[R|Q],P):-
        conj_pair(R,T),
        substitute(O,Y,X,N),
        noun_phrase(Z,O,N,M,V,Q,L),
        conjunction(M,K,W,[U,T],L,J),
        noun_phrase(Z,Y,X,K,S,J,P).
cp_noun_phrase(Z,Y,X,W,[cnp2,V,U,T],S,R):-
        substitute(Q,Y,X,P),
        noun_phrase(Z,Q,P,O,V,S,N),
        conjunction(O,M,W,U,N,L),
```

```
        cp_noun_phrase(Z,Y,X,M,T,L,R).
cp_noun_phrase(Z,Y,~X,W,[cnp3,V],U,T):-
        noun_phrase_not(Z,Y,X,W,V,U,T).
cp_noun_phrase(Z,Y,X,W,[cnp4,V],U,T):-
        noun_phrase_not(Z,Y,~X,W,V,U,T).

/********** NOUN_PHRASE **********/

noun_phrase(Z,Y,X,W,[np1,V,U,T],S,R):-
        gdeterminer(Z,Y,Q,P,W,V,S,O),
        gnoun(Z,Y,Q,U,O,N),
        rel_clause(Z,Y,M,T,N,R),
        incorporate2(X,M,P).
noun_phrase(Z,Y,X,W,[np2,V,U],T,S):-
        gdeterminer(Z,Y,R,X,W,V,T,Q),
        gnoun(Z,Y,R,U,Q,S).
noun_phrase(Z,Y,X,X,[np3,W],V,U):-
        gnoun(Z,Y,X,W,V,U).
noun_phrase(Z,Y,X,W,[np4,V,U],T,S):-
        quan_pronoun(Z,Y,R,W,V,T,Q),
        rel_clause(Z,Y,P,U,Q,S),
        incorporate2(X,P,R).
noun_phrase(Z,Y,X,W,[np5,V],U,T):-
        quan_pronoun(Z,Y,X,W,V,U,T).
noun_phrase(Z,Y,X,W,[np6,V,U],T,S):-
        gproper_noun(Z,Y,U,T,R),
        rel_clause(Z,Y,Q,V,R,S),
        incorporate2(X,Q,W).
noun_phrase(Z,Y,X,X,[np7,W],V,U):-
        gproper_noun(Z,Y,W,V,U).

/********** CP_VERB_PHRASE **********/

cp_verb_phrase(Z,Y,X,[cvp0,W],V,U):-
        verb_phrase(Z,Y,X,W,V,U).
cp_verb_phrase(Z,Y,X,[cvp1,W,V,U],T,S):-
        verb_phrase(Z,Y,R,W,T,Q),
        conjunction(R,P,X,V,Q,O),
        cp_verb_phrase(Z,Y,P,U,O,S).


/********** VERB_PHRASE **********/

verb_phrase(Z,Y,X,[vp00,W,not,V],U,T):-
        trans_verb(Z,Y,S,f(is,Y,S),W,U,R),
        negatif(R,Q),
        cp_class_name_not(Z,Y,X,V,Q,T).
verb_phrase(Z,Y,X,[vp11,W,not,V,U],T,S):-
        auxiliary(Z,W,T,R),
        negatif(R,Q),
        trans_verb(plural,Y,P,O,V,Q,N),
        cp_noun_phrase(M,P,~O,X,U,N,S).
verb_phrase(Z,Y,~X,[vp21,W,not,V],U,T):-
        auxiliary(Z,W,U,S),
        negatif(S,R),
        intrans_verb(plural,Y,X,V,R,T).
```

```
verb_phrase(Z,Y,~(X&W),[vp31,V,not,U,T],S,R):-
        auxiliary(Z,V,S,Q),
        negatif(Q,P),
        intrans_verb(plural,Y,X,U,P,O),
        glocator(Y,W,T,O,R).
verb_phrase(Z,Y,X,[vp01,W,V,U],T,S):-
        auxiliary(Z,W,T,R),
        trans_verb(plural,Y,Q,P,V,R,O),
        cp_noun_phrase(N,Q,P,X,U,O,S).
verb_phrase(Z,Y,X,[vp02,W,V],U,T):-
        auxiliary(Z,W,U,S),
        intrans_verb(plural,Y,X,V,S,T).
verb_phrase(Z,Y,X&W,[vp03,V,U,T],S,R):-
        auxiliary(Z,V,S,Q),
        intrans_verb(plural,Y,X,U,Q,P),
        glocator(Y,W,T,P,R).
verb_phrase(Z,Y,X,[vp0,W,V],U,T):-
        trans_verb(Z,Y,S,f(is,Y,S),W,U,R),
        cp_class_name(Z,Y,X,V,R,T).
verb_phrase(Z,Y,X,[vp4,W,V],U,T):-
        trans_verb(Z,Y,S,f(is,Y,S),W,U,R),
        glocator(Y,X,V,R,T).
verb_phrase(Z,Y,X,[vp1,W,V],U,T):-
        trans_verb(Z,Y,S,R,W,U,Q),
        cp_noun_phrase(P,S,R,X,V,Q,T).
verb_phrase(Z,Y,X,[vp2,W],V,U):-
        intrans_verb(Z,Y,X,W,V,U).
verb_phrase(Z,Y,X&W,[vp3,V,U],T,S):-
        intrans_verb(Z,Y,X,V,T,R),
        glocator(Y,W,U,R,S).

/********** GLOCATOR **********/

glocator(Z,Y,[gl,X|W],[X|V],U):-
        locator(X,T),
        noun_phrase(S,R,f(T,Z,R),Y,W,V,U).

/********** GPLACE **********/

gplace(Z,Y,X,[gp|W],V,U):-
        noun_phrase(Z,Y,X,X,W,V,U).

/********** REL_CLAUSE **********/

rel_clause(Z,Y,X,[rc,W|V],[W|U],T):-
        rel_clause(W),
        verb_phrase(Z,Y,X,V,U,T).

/********** GDETERMINER **********/

gdeterminer(Z,Y,X,W,all(Y,indefinite(Y),X=>W),
                                [gdet1,V],[V|U],U):-
        determiner(Z,universal,V).
gdeterminer(Z,Y,X,W,exists(Y,indefinite(Y),X&W),
                                [gdet2,V],[V|U],U):-
        determiner(Z,existential,V).
```

```
gdeterminer(singular,Z,Y,X,exists(Z,definite(Z),Y&X),
                                    [gdet3],[the|W],W):-
        true.
gdeterminer(plural,Z,Y,X,all(Z,indefinite(Z),Y=>X),
                                    [gdet6],W,W):-
        true.


/********** GNOUN **********/

gnoun(Z,Y,X&W,[gn1,V,U],T,S):-
        gadjective(Z,Y,W,V,T,R),
        gnoun(Z,Y,X,U,R,S).
gnoun(Z,Y,X,[gn2,W],V,U):-
        gnoun0(Z,Y,X,W,V,U).

/********** GNOUN0 **********/

gnoun0(singular,Z,f(Y,Z),[gn02,X],[X|W],W):-
        noun(X,Y).
gnoun0(plural,Z,f(Y,Z),[gn03,X],[X|W],W):-
        plural(V,X),
        noun(V,Y).
gnoun0(plural,Z,f(Y,Z),[gn04,X],[X|W],W):-
        name(X,V),
        append(U,[105,101,115],V),
        append(U,[121],T),
        name(S,T),
        noun(S,Y).
gnoun0(plural,Z,f(Y,Z),[gn05,X],[X|W],W):-
        name(X,V),
        append(U,[115],V),
        name(T,U),
        noun(T,Y).


/********** GADJECTIVE **********/

gadjective(Z,Y,f(X,Y),[gadj,W],[W|V],V):-
        adjective(W,X).

/********** GPROPER_NOUN **********/

gproper_noun(Z,Y,[X,Y],[who|W],W):-
        var(Y).
gproper_noun(plural,Z,[gpn2,Y],[Y|X],X):-
        nonvar(Y),
        plural(W,Y),
        proper_noun(W,Z),
        not noun(W,V).
```

```
gproper_noun(singular,Z,[gpn1,Y],[Y|X],X):-
        nonvar(Y),
        proper_noun(Y,Z),
        not noun(Y,W),
        not plural(V,Y),
        not pronoun(U,Y,T).

/********** TRANS_VERB **********/

trans_verb(singular,Z,Y,f(X,Z,Y),[tv1,X],[X|W],W):-
        verb_be(X,X).
trans_verb(plural,Z,Y,f(X,Z,Y),[tv1,X],[W|V],V):-
        verb_be(X,W),
        plural(W).
trans_verb(singular,Z,Y,f(X,Z,Y),[tv7,W],[W|V],V):-
        verb(W,X),
        transitive(W).
trans_verb(plural,Z,Y,f(X,Z,Y),[tv8,W],[W|V],V):-
        plural(U,W),
        verb(U,X),
        transitive(U).
trans_verb(plural,Z,Y,f(X,Z,Y),[tv9,W],[W|V],V):-
        name(W,U),
        append(U,[115],T),
        name(S,T),
        verb(S,X),
        transitive(S).

/********** INTRANS_VERB **********/

intrans_verb(singular,Z,f(Y,Z),[iv1,X],[X|W],W):-
        verb(X,Y),
        intransitive(X).
intrans_verb(plural,Z,f(Y,Z),[iv2,X],[X|W],W):-
        plural(V,X),
        verb(V,Y),
        intransitive(V).
intrans_verb(plural,Z,f(Y,Z),[iv3,X],[X|W],W):-
        name(X,V),
        append(V,[115],U),
        name(T,U),
        verb(T,Y),
        intransitive(T).


/********** QUAN_PRONOUN **********/
quan_pronoun(singular,Z,Y,exists(Z,indefinite(Z),f(W,Z)&Y),
                                        [qpr1,X],[X|V],V):-
        pronoun(existential,X,W).
quan_pronoun(singular,Z,Y,all(Z,indefinite(Z),f(W,Z)=>Y),
                                        [qpr2,X],[X|V],V):-
        pronoun(universal,X,W).
```

```
/********** CONJUNCTION **********/

conjunction(Z,Y,Z&Y,[conj1,and],[and|X],X):-
        true.
conjunction(Z,Y,Z#Y,[conj2,or],[or|X],X):-
        true.
conjunction(Z,Y,~(Z&Y),[conj3,nor],[nor|X],X):-
        true.

/********** CP_CLASS_NAME_NOT **********/

cp_class_name_not(Z,Y,~X,[ccnn0,W],V,U):-
        class_name_not(Z,Y,X,W,V,U).
cp_class_name_not(Z,Y,X,[ccnn1,W,[V,U],T],[S|R],Q):-
        conj_pair(S,U),
        class_name_not(Z,Y,P,W,R,O),
        conjunction(~P,N,X,[V,U],O,M),
        cp_class_name_not(Z,Y,N,T,M,Q).
cp_class_name_not(Z,Y,X,[ccnn2,W,V,U],T,S):-
        class_name_not(Z,Y,R,W,T,Q),
        conjunction(~R,P,X,V,Q,O),
        cp_class_name_not(Z,Y,P,U,O,S).

/********** CLASS_NAME_NOT **********/

class_name_not(singular,Z,Y,[cnn1|X],[a|W],V):-
        gnoun_not(singular,Z,Y,X,W,V).
class_name_not(singular,Z,Y,[cnn2|X],[an|W],V):-
        gnoun_not(singular,Z,Y,X,W,V).
class_name_not(plural,Z,Y,[cnn3|X],W,V):-
        gnoun_not(plural,Z,Y,X,W,V).
class_name_not(Z,Y,X,[cnn4|W],V,U):-
        gadjective(Z,Y,X,W,V,U).


/********** NOUN_PHRASE_NOT **********/

noun_phrase_not(Z,Y,X,W,[npn1,V,U,T],S,R):-
        gdeterminer_no(Z,Y,Q,P,W,V,S,O),
        gnoun(Z,Y,Q,U,O,N),
        rel_clause(Z,Y,M,T,N,R),
        incorporate2(X,M,P).
noun_phrase_not(Z,Y,~X,W,[npn2,V,U],T,S):-
        gdeterminer_no(Z,Y,R,~X,W,V,T,Q),
        gnoun(Z,Y,R,U,Q,S).
noun_phrase_not(Z,Y,X,W,[npn3,V,U],T,S):-
        gdeterminer_no(Z,Y,R,X,W,V,T,Q),
        gnoun(Z,Y,R,U,Q,S).

/********** CP_VERB_PHRASE_NOT **********/

cp_verb_phrase_not(Z,Y,X,[cvpn0,W],V,U):-
        verb_phrase_not(Z,Y,X,W,V,U).
```

```
cp_verb_phrase_not(Z,Y,X,[cvpn1,W,V,U],T,S):-
        verb_phrase_not(Z,Y,R,W,T,Q),
        conjunction(R,P,X,V,Q,O),
        cp_verb_phrase_not(Z,Y,P,U,O,S).

/********** VERB_PHRASE_NOT **********/

verb_phrase_not(Z,Y,X,[vpn4,W,not,V],U,T):-
        trans_verb(Z,Y,S,f(is,Y,S),W,U,R),
        negatif(R,Q),
        cp_class_name(Z,Y,X,V,Q,T).
verb_phrase_not(Z,Y,X,[vpn5,W,not,V,U],T,S):-
        auxiliary(Z,W,T,R),
        negatif(R,Q),
        trans_verb(plural,Y,P,O,V,Q,N),
        cp_noun_phrase(M,P,O,X,U,N,S).
verb_phrase_not(Z,Y,X,[vpn6,W,not,V],U,T):-
        auxiliary(Z,W,U,S),
        negatif(S,R),
        intrans_verb(plural,Y,X,V,R,T).
verb_phrase_not(Z,Y,X&W,[vpn7,V,not,U,T],S,R):-
        auxiliary(Z,V,S,Q),
        negatif(Q,P),
        intrans_verb(plural,Y,X,U,P,O),
        glocator(Y,W,T,O,R).
verb_phrase_not(Z,Y,X,[vpn01,W,V,U],T,S):-
        auxiliary(Z,W,T,R),
        trans_verb(plural,Y,Q,P,V,R,O),
        cp_noun_phrase(N,Q,~P,X,U,O,S).
verb_phrase_not(Z,Y,~X,[vpn02,W,V],U,T):-
        auxiliary(Z,W,U,S),
        intrans_verb(plural,Y,X,V,S,T).
verb_phrase_not(Z,Y,~(X&W),[vpn03,V,U,T],S,R):-
        auxiliary(Z,V,S,Q),
        intrans_verb(plural,Y,X,U,Q,P),
        glocator(Y,W,T,P,R).
verb_phrase_not(Z,Y,X,[vpn0,W,V],U,T):-
        trans_verb(Z,Y,S,f(is,Y,S),W,U,R),
        cp_class_name_not(Z,Y,X,V,R,T).
verb_phrase_not(Z,Y,X,[vpn1,W,V],U,T):-
        trans_verb(Z,Y,S,R,W,U,Q),
        cp_noun_phrase(P,S,~R,X,V,Q,T).
verb_phrase_not(Z,Y,~X,[vpn2,W],V,U):-
        intrans_verb(Z,Y,X,W,V,U).
verb_phrase_not(Z,Y,~(X&W),[vpn3,V,U],T,S):-
        intrans_verb(Z,Y,X,V,T,R),
        glocator(Y,W,U,R,S).

/********** NEGATIF **********/

negatif([not|Z],Z):-
        true.

/********** AUXILIARY **********/
auxiliary(plural,[aux1,do],[do|Z],Z):-
        true.
```

```
auxiliary(singular,[aux2,does],[does|Z],Z):-
        true.

/********** GDETERMINER_NO **********/

gdeterminer_no(singular,Z,Y,X,all(Z,indefinite(Z),Y=>X),
                                    [gdetn1],[no|W],W):-
        true.


/********** GADJECTIVE_NOT **********/

gadjective_not(Z,Y,X#W,[gadjn0,V,U],T,S):-
        gadjective_not0(Z,Y,X,V,T,R),
        gadjective_not(Z,Y,W,U,R,S).
gadjective_not(Z,Y,X,[gadjn1,W],V,U):-
        gadjective_not0(Z,Y,X,W,V,U).

/********** GADJECTIVE_NOT0 **********/

gadjective_not0(Z,Y,f(X,Y),[gadj,W],[W|V],V):-
        adjective(W,X).

/********** QUAN_PRONOUN_NOT **********/

quan_pronoun_not(singular,Z,Y,
        exists(Z,indefinite(Z),f(W,Z)&Y),[qpr1,X],[X|V],V):-
                        pronoun(existential,X,W).
quan_pronoun_not(singular,Z,Y,
        all(Z,indefinite(Z),f(W,Z)=>Y),[qpr2,X],[X|V],V):-
                        pronoun(universal,X,W).

/********** GNOUN_NOT **********/

gnoun_not(Z,Y,~X#W,[gnn1,V,U],T,S):-
        gadjective_not(Z,Y,W,V,T,R),
        gnoun_not(Z,Y,X,U,R,S).
gnoun_not(Z,Y,X,[gnn2,W],V,U):-
        gnoun0(Z,Y,X,W,V,U).
```

```
/* the  WORDING TECHNIQUE */
/********** SENTENCE **********/

sentence(Z,Y,X):-
        statement_phrase(Z,Y,X).
sentence(Z,Y,X):-
        question_phrase(Z,Y,X).


/********** STATEMENT_PHRASE **********/

statement_phrase(Z,Y,X):-
        noun_phrase(W,V,U,Z,Y,T),
        cp_verb_phrase(W,V,U,T,X).
statement_phrase(Z,Y,X):-
        noun_phrase_not(W,V,U,Z,Y,T),
        cp_verb_phrase_not(W,V,U,T,X).


/********** CP_CLASS_NAME **********/

cp_class_name(Z,Y,X,W,V):-
        class_name(Z,Y,X,W,V).


/********** CLASS_NAME **********/

class_name(singular,Z,Y,[a|X],W):-
        gnoun(singular,Z,Y,X,W).
class_name(singular,Z,Y,[an|X],W):-
        gnoun(singular,Z,Y,X,W).
class_name(plural,Z,Y,X,W):-
        gnoun(plural,Z,Y,X,W).
class_name(Z,Y,X,W,V):-
        gadjective(Z,Y,X,W,V).


/********** CP_NOUN_PHRASE **********/

cp_noun_phrase(Z,Y,X,W,V,U):-
        noun_phrase(Z,Y,X,W,V,U).
cp_noun_phrase(Z,Y,~X,W,V,U):-
        noun_phrase_not(Z,Y,X,W,V,U).
cp_noun_phrase(Z,Y,X̄,W,V,U):-
        noun_phrase_not(Z,Y,~X,W,V,U).


/********** NOUN_PHRASE **********/

noun_phrase(Z,Y,X,exists(Y,R,W),V,U):-
        gproper_noun(Z,Y,R,V,T),
        relative_clause(Z,Y,X,W,T,U).
noun_phrase(Z,Y,X̄,exists(Y,U,X),W,V):-
        gproper_noun(Z,Y,U,W,V).
noun_phrase(Z,Y,X,W,V,U):-
        gdeterminer(Z,Y,T,S,W,V,R),
        gnoun(Z,Y,T,R,Q),
```

```
        relative_clause(Z,Y,X,S,Q,U).
noun_phrase(Z,Y,X̄,W,V,U):-
        gdeterminer(Z,Y,T,X,W,V,S),
        gnoun(Z,Y,T,S,U).
noun_phrase(Z,Y,X,X,W,V):-
        gnoun(Z,Y,X,W,V).
noun_phrase(Z,Y,X,W,V,U):-
        quan_pronoun(Z,Y,T,W,V,S),
        relative_clause(Z,Y,X,T,S,U).
noun_phrase(Z,Y,X̄,W,V,U):-
        quan_pronoun(Z,Y,X,W,V,U).


/********** CP_VERB_PHRASE **********/

cp_verb_phrase(Z,Y,X,W,V):-
        verb_phrase(Z,Y,X,W,V).


/********** VERB_PHRASE **********/

verb_phrase(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        cp_class_name(Z,Y,X,T,V).
verb_phrase(Z,Y,X̄,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        glocator(Y,X,T,V).
verb_phrase(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,T,W,S),
        cp_noun_phrase(R,U,T,X,S,V).
verb_phrase(Z,Y,X̄,W,V):-
        intrans_verb(Z,Y,X,W,V).
verb_phrase(Z,Y,X&W,V,U):-
        intrans_verb(Z,Y,X,V,T),
        glocator(Y,W,T,U).
verb_phrase(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        negatif(T,S),
        cp_class_name_not(Z,Y,X,S,V).
verb_phrase(Z,Y,X̄,W,V̄):-
        auxiliary(Z,W,U),
        negatif(U,T),
        trans_verb(plural,Y,S,R,T,Q),
        cp_noun_phrase(P,S,~R,X,Q,V).
verb_phrase(Z,Y,~X,W,V):-
        auxiliary(Z,W,U),
        negatif(U,T),
        intrans_verb(plural,Y,X,T,V).
verb_phrase(Z,Y,~(X&W),V,U):-
        auxiliary(Z,V,T),
        negatif(T,S),
        intrans_verb(plural,Y,X,S,R),
        glocator(Y,W,R,U).
verb_phrase(Z,Y,X,W,V):-
        auxiliary(Z,W,U),
        trans_verb(plural,Y,T,S,U,R),
        cp_noun_phrase(Q,T,S,X,R,V).
```

```
verb_phrase(Z,Y,X,W,V):-
        auxiliary(Z,W,U),
        intrans_verb(plural,Y,X,U,V).
verb_phrase(Z,Y,X&W,V,U):-
        auxiliary(Z,V,T),
        intrans_verb(plural,Y,X,T,S),
        glocator(Y,W,S,U).


/********** GLOCATOR **********/

glocator(Z,Y,[X|W],V):-
        word_used(X),
        locator(X,U),
        noun_phrase(T,S,f(U,Z,S),Y,W,V).


/********** GPLACE **********/

gplace(Z,Y,X,W,V):-
        noun_phrase(Z,Y,X,X,W,V).


/********** REL_CLAUSE **********/

relative_clause(Z,Y,T,T&X,[W|V],U):-
        word_used(W),
        rel_clause(W),
        verb_phrase(Z,Y,X,V,U).


/********** GDETERMINER **********/
gdeterminer(Z,Y,X,W,all(Y,indefinite(Y),X=>W),[V|U],U):-
        word_used(V),
        determiner(Z,universal,V).
gdeterminer(Z,Y,X,W,exists(Y,indefinite(Y),X&W),[V|U],U):-
        word_used(V),
        determiner(Z,existential,V).
gdeterminer(singular,Z,Y,X,exists(Z,definite(Z),Y&X),
                                                [the|W],W):-
        true.
gdeterminer(plural,Z,Y,X,all(Z,definite(Z),Y&X),[the|W],W):-
        true.
gdeterminer(plural,Z,Y,X,all(Z,indefinite(Z),Y=>X),W,W):-
        true.


/********** GNOUN **********/

gnoun(Z,Y,X,W,V):-
        gnoun0(Z,Y,X,W,V).
```

```
/********** GNOUN0 **********/
gnoun0(singular,Z,f(Y,Z),[X|W],W):-
        word_used(X),
        noun(X,Y).
gnoun0(plural,Z,f(Y,Z),[X|W],W):-
        word_used(X),
        plural(V,X),
        noun(V,Y).
gnoun0(plural,Z,f(Y,Z),[X|W],W):-
        word_used(X),
        name(X,V),
        append(U,[105,101,115],V),
        append(U,[121],T),
        name(S,T),
        noun(S,Y).
gnoun0(plural,Z,f(Y,Z),[X|W],W):-
        word_used(X),
        name(X,V),
        append(U,[115],V),
        name(T,U),
        noun(T,Y).


/********** GADJECTIVE **********/

gadjective(Z,Y,f(X,Y),[W|V],V):-
        word_used(W),
        adjective(W,X).


/********** GPROPER_NOUN **********/

gproper_noun(_,Z,proper_noun(Z),[who|X],X):-
        var(Z).
gproper_noun(singular,Z,proper_noun(Z),[Y|X],X):-
        proper_noun(Y,Z),
        word_used(Y).
gproper_noun(plural,Z,proper_noun(Z),[Y|X],X):-
        word_used(Y),
        plural(W,Y),
        proper_noun(W,Z),
        not noun(W,V).


/********** TRANS_VERB **********/

trans_verb(singular,Z,Y,f(W,Z,Y),[W|X],X):-
        /*word_used(W),*/
        verb_be(W,W).
trans_verb(plural,Z,Y,f(V,Z,Y),[W|X],X):-
        /*word_used(W),*/
        verb_be(V,W),
        plural(W).
trans_verb(singular,Z,Y,f(X,Z,Y),[W|V],V):-
        word_used(W),
        verb(W,X),
        transitive(W).
```

```
trans_verb(plural,Z,Y,f(X,Z,Y),[W|V],V):-
        word_used(W),
        plural(U,W),
        verb(U,X),
        transitive(U).
trans_verb(plural,Z,Y,f(X,Z,Y),[W|V],V):-
        word_used(W),
        name(W,U),
        append(U,[115],T),
        name(S,T),
        verb(S,X),
        transitive(S).


/********** INTRANS_VERB **********/

intrans_verb(singular,Z,f(Y,Z),[X|W],W):-
        word_used(X),
        verb(X,Y),
        intransitive(X).
intrans_verb(plural,Z,f(Y,Z),[X|W],W):-
        word_used(X),
        plural(V,X),
        verb(V,Y),
        intransitive(V).
intrans_verb(plural,Z,f(Y,Z),[X|W],W):-
        word_used(X),
        name(X,V),
        append(V,[115],U),
        name(T,U),
        verb(T,Y),
        intransitive(T).


/********** QUAN_PRONOUN **********/
quan_pronoun(singular,Z,Y,exists(Z,indefinite(Z),f(W,Z)&Y),
                                                    [X|V],V):-
        word_used(X),
        pronoun(existential,X,W).
quan_pronoun(singular,Z,Y,all(Z,indefinite(Z),f(W,Z)=>Y),
                                                    [X|V],V):-
        word_used(X),
        pronoun(universal,X,W).

/********** CONJUNCTION **********/

conjunction(Z,Y,Z&Y,[and],[and|X],X):-
        true.
conjunction(Z,Y,Z#Y,[or],[or|X],X):-
        true.
conjunction(Z,Y,~(Z&Y),[nor],[nor|X],X):-
        true.
```

```
/********** CP_CLASS_NAME_NOT **********/
cp_class_name_not(Z,Y,~X,W,V):-
        class_name_not(Z,Y,X,W,V).


/********** CLASS_NAME_NOT **********/

class_name_not(singular,Z,Y,[a|X],W):-
        gnoun_not(singular,Z,Y,X,W).
class_name_not(singular,Z,Y,[an|X],W):-
        gnoun_not(singular,Z,Y,X,W).
class_name_not(plural,Z,Y,X,W):-
        gnoun_not(plural,Z,Y,X,W).
class_name_not(Z,Y,X,W,V):-
        gadjective(Z,Y,X,W,V).


/********** NOUN_PHRASE_NOT **********/

noun_phrase_not(Z,Y,X,W,V,U):-
        gdeterminer_no(Z,Y,T,S,W,V,R),
        gnoun(Z,Y,T,R,Q),
        relative_clause(Z,Y,X,S,Q,U).
noun_phrase_not(Z,Y,~X,W,V,U):-
        gdeterminer_no(Z,Y,T,~X,W,V,S),
        gnoun(Z,Y,T,S,U).
noun_phrase_not(Z,Y,X,W,V,U):-
        gdeterminer_no(Z,Y,T,X,W,V,S),
        gnoun(Z,Y,T,S,U).


/********** CP_VERB_PHRASE_NOT **********/

cp_verb_phrase_not(Z,Y,X,W,V):-
        verb_phrase_not(Z,Y,X,W,V).


/********** VERB_PHRASE_NOT **********/

verb_phrase_not(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        negatif(T,S),
        cp_class_name(Z,Y,X,S,V).
verb_phrase_not(Z,Y,X,W,V):-
        auxiliary(Z,W,U),
        negatif(U,T),
        trans_verb(plural,Y,S,R,T,Q),
        cp_noun_phrase(P,S,R,X,Q,V).
verb_phrase_not(Z,Y,X,W,V):-
        auxiliary(Z,W,U),
        negatif(U,T),
        intrans_verb(plural,Y,X,T,V).
verb_phrase_not(Z,Y,X&W,V,U):-
        auxiliary(Z,V,T),
        negatif(T,S),
        intrans_verb(plural,Y,X,S,R),
        glocator(Y,W,R,U).
```

```
verb_phrase_not(Z,Y,X,W,V):-
        auxiliary(Z,W,U),
        trans_verb(plural,Y,T,S,U,R),
        cp_noun_phrase(Q,T,~S,X,R,V).
verb_phrase_not(Z,Y,~X,W,V):-
        auxiliary(Z,W,U),
        intrans_verb(plural,Y,X,U,V).
verb_phrase_not(Z,Y,~(X&W),V,U):-
        auxiliary(Z,V,T),
        intrans_verb(plural,Y,X,T,S),
        glocator(Y,W,S,U).
verb_phrase_not(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        cp_class_name_not(Z,Y,X,T,V).
verb_phrase_not(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,T,W,S),
        cp_noun_phrase(R,U,~T,X,S,V).
verb_phrase_not(Z,Y,~X,W,V):-
        intrans_verb(Z,Y,X,W,V).
verb_phrase_not(Z,Y,~(X&W),V,U):-
        intrans_verb(Z,Y,X,V,T),
        glocator(Y,W,T,U).


/********** NEGATIF **********/

negatif([not|Z],Z):-
        true.


/********** AUXILIARY **********/

auxiliary(plural,[do|Z],Z):-
        true.
auxiliary(singular,[does|Z],Z):-
        true.


/********** GDETERMINER_NO **********/
gdeterminer_no(singular,Z,Y,X,all(Z,indefinite(Z),Y=>X),
                                        [no|W],W):-
        true.


/********** GADJECTIVE_NOT **********/

gadjective_not(Z,Y,X,W,V):-
        gadjective_not0(Z,Y,X,W,V).


/********** GADJECTIVE_NOT0 **********/

gadjective_not0(Z,Y,f(X,Y),[W|V],V):-
        word_used(W),
        adjective(W,X).
```

```
/********** QUAN_PRONOUN_NOT **********/
quan_pronoun_not(singular,Z,Y,
                 exists(Z,indefinite(Z),f(W,Z)&Y),[X|V],V):-
        word_used(X),
        pronoun(existential,X,W).
quan_pronoun_not(singular,Z,Y,
                 all(Z,indefinite(Z),f(W,Z)=>Y),[X|V],V):-
        word_used(X),
        pronoun(universal,X,W).


/********** GNOUN_NOT **********/
gnoun_not(Z,Y,X,W,V):-
        gnoun0(Z,Y,X,W,V).
```

```
/********** SENTENCE **********/

sentence(Z,Y,X):-
        statement_phrase(Z,Y,X).
sentence(Z,Y,X):-
        question_phrase(Z,Y,X).


/********** STATEMENT_PHRASE **********/

statement_phrase(Z,Y,X):-
        noun_phrase(W,V,U,Z,Y,T),
        cp_verb_phrase(W,V,U,T,X).
statement_phrase(Z,Y,X):-
        noun_phrase_not(W,V,U,Z,Y,T),
        cp_verb_phrase_not(W,V,U,T,X).


/********** CP_CLASS_NAME **********/

cp_class_name(Z,Y,X,W,V):-
        class_name(Z,Y,X,W,V).



/********** CLASS_NAME **********/

class_name(singular,Z,Y,[a|X],W):-
        gnoun(singular,Z,Y,X,W).
class_name(singular,Z,Y,[an|X],W):-
        gnoun(singular,Z,Y,X,W).
class_name(plural,Z,Y,X,W):-
        gnoun(plural,Z,Y,X,W).
class_name(Z,Y,X,W,V):-
        gadjective(Z,Y,X,W,V).


/********** CP_NOUN_PHRASE **********/

cp_noun_phrase(Z,Y,X,W,V,U):-
        noun_phrase(Z,Y,X,W,V,U).


cp_noun_phrase(Z,Y,~X,W,V,U):-
        noun_phrase_not(Z,Y,X,W,V,U).
cp_noun_phrase(Z,Y,X,W,V,U):-
        noun_phrase_not(Z,Y,~X,W,V,U).


/********** NOUN_PHRASE **********/


noun_phrase(Z,Y,X,exists(Y,R,W),V,U):-
        gproper_noun(Z,Y,R,V,T),
        relative_clause(Z,Y,X,W,T,U).
```

```
noun_phrase(Z,Y,X,exists(Y,U,X),W,V):-
        gproper_noun(Z,Y,U,W,V).
noun_phrase(Z,Y,X,W,V,U):-
        gdeterminer(Z,Y,T,S,W,V,R),
        gnoun(Z,Y,T,R,Q),
        relative_clause(Z,Y,X,S,Q,U).
noun_phrase(Z,Y,X,W,V,U):-
        gdeterminer(Z,Y,T,X,W,V,S),
        gnoun(Z,Y,T,S,U).
noun_phrase(Z,Y,X,X,W,V):-
        gnoun(Z,Y,X,W,V).
noun_phrase(Z,Y,X,W,V,U):-
        quan_pronoun(Z,Y,T,W,V,S),
        relative_clause(Z,Y,X,T,S,U).
noun_phrase(Z,Y,X,W,V,U):-
        quan_pronoun(Z,Y,X,W,V,U).


/********** CP_VERB_PHRASE **********/

cp_verb_phrase(Z,Y,X,W,V):-
        verb_phrase(Z,Y,X,W,V).

/********** VERB_PHRASE **********/
verb_phrase(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        cp_class_name(Z,Y,X,T,V).
verb_phrase(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        glocator(Y,X,T,V).
verb_phrase(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,T,W,S),
        cp_noun_phrase(R,U,T,X,S,V).
verb_phrase(Z,Y,X,W,V):-
        intrans_verb(Z,Y,X,W,V).
verb_phrase(Z,Y,X&W,V,U):-
        intrans_verb(Z,Y,X,V,T),
        glocator(Y,W,T,U).
verb_phrase(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        negatif(T,S),
        cp_class_name_not(Z,Y,X,S,V).
verb_phrase(Z,Y,X,W,V):-
        auxiliary(Z,W,U),
        negatif(U,T),
        trans_verb(plural,Y,S,R,T,Q),
        R\=f(is,_,_),  /* to prevent 'do <not> is ' */
        cp_noun_phrase(P,S,~R,X,Q,V).
verb_phrase(Z,Y,~X,W,V):-
        auxiliary(Z,W,U),
        negatif(U,T),
        intrans_verb(plural,Y,X,T,V).
verb_phrase(Z,Y,~(X&W),V,U):-
        auxiliary(Z,V,T),
        negatif(T,S),
        intrans_verb(plural,Y,X,S,R),
        glocator(Y,W,R,U).
```

```
/********** GLOCATOR **********/
glocator(Z,Y,[X|W],V):-
        locator(X,U),
        noun_phrase(T,S,f(U,Z,S),Y,W,V).


/********** GPLACE **********/

gplace(Z,Y,X,W,V):-
        noun_phrase(Z,Y,X,X,W,V).

/********** REL_CLAUSE **********/
relative_clause(Z,Y,T,T&X,[W|V],U):-
        rel_clause(W),
        verb_phrase(Z,Y,X,V,U).

/********** GDETERMINER **********/
gdeterminer(Z,Y,X,W,
   all(Y,indefinite(Y,coverage(Coverage)),X=>W),[V|U],U):-
        nonvar(V),
        determiner(Z,universal,V,Coverage).
gdeterminer(Z,Y,X,W,
   exists(Y,indefinite(Y,coverage(Coverage)),X&W),[V|U],U):-
        nonvar(V),
        determiner(Z,existential,V,Coverage).
gdeterminer(Z,Y,X,W,
   all(Y,indefinite(Y,coverage(Coverage)),X=>W),[V|U],U):-
        var(V),
        determiner_coverage(Z,universal,V,Coverage).
gdeterminer(Z,Y,X,W,
   exists(Y,indefinite(Y,coverage(Coverage)),X&W),[V|U],U):-
        var(V),
        determiner_coverage(Z,existential,V,Coverage).
gdeterminer(singular,Z,Y,X,
   exists(Z,definite(Z,coverage(10)),Y&X),[the|W],W).
gdeterminer(plural,Z,Y,X,
   all(Z,definite(Z,coverage(10)),Y&X),[the|W],W).
gdeterminer(plural,Z,Y,X,
   all(Z,indefinite(Z,coverage(70)),Y=>X),[V|W],[V|W]):-
        V\==who.

determiner_coverage(Z,Quantifier,V,Coverage):-
        integer(Coverage),
        determiner(Z,Quantifier,V,Coverage1),
        Coverage1=Coverage.
determiner_coverage(Z,Quantifier,V,Coverage):-
        integer(Coverage),
        determiner(Z,Quantifier,V,Coverage1),
        Coverage1<Coverage,!.
determiner_coverage(Z,Quantifier,V,Coverage):-
        nonvar(Coverage),
        determiner(Z,Quantifier,V,Coverage1),
        Coverage1=<50.
```

```
/********** GNOUN **********/
gnoun(Z,Y,X,W,V):-
        gnoun0(Z,Y,X,W,V).


/********** GNOUN0 **********/
gnoun0(singular,Z,f(Y,Z),[X|W],W):-
        nonvar(X),
        noun(X,Y).
gnoun0(singular,Z,f(Y,Z),[X|W],W):-
        var(X),
        noun(X,Y),
        X==Y. /* to prevent X={what,which} */
gnoun0(plural,Z,f(Y,Z),[X|W],W):-
        nonvar(X),
        plural(V,X),
        noun(V,Y).
gnoun0(plural,Z,f(Y,Z),[X|W],W):-
        var(X),
        noun(V,Y),
        V==Y,
        plural(V,X).
gnoun0(plural,Z,f(Y,Z),[X|W],W):-
        nonvar(X),
        name(X,V),
        append(U,[105,101,115],V),
        append(U,[121],T),
        name(S,T),
        noun(S,Y).
gnoun0(plural,Z,f(Y,Z),[X|W],W):-
        var(X),
        noun(V,Y),
        V==Y,
        name(V,U),
        append(T,[121],U),
        append(T,[105,101,115],S),
        name(X,S).
gnoun0(plural,Z,f(Y,Z),[X|W],W):-
        nonvar(X),
        name(X,V),
        append(U,[115],V),
        name(T,U),
        noun(T,Y).
gnoun0(plural,Z,f(Y,Z),[X|W],W):-
        var(X),
        noun(V,Y),
        V==Y,
        not(plural(V,V1)),
        name(V,U),
        append(U,[115],T),
        name(X,T).
```

```
/********** GADJECTIVE **********/

gadjective(Z,Y,f(X,Y),[W|V],V):-
        adjective(W,X).


/********** GPROPER_NOUN **********/

gproper_noun(Arbitarary,Z,proper_noun(Z),[who|X],X):-
        var(Z).
gproper_noun(singular,Z,proper_noun(Z),[Y|X],X):-
        proper_noun(Y,Z).
gproper_noun(plural,Z,proper_noun(Z),[Y|X],X):-
        nonvar(Y),
        plural(W,Y),
        proper_noun(W,Z),
        not (noun(W,V)).
gproper_noun(plural,Z,proper_noun(Z),[Y|X],X):-
        var(Y),
        proper_noun(W,Z),
        plural(W,Y),
        not (noun(W,V)).


/********** TRANS_VERB **********/

trans_verb(singular,Z,Y,f(W,Z,Y),[W|X],X):-
        verb_be(W,W).
trans_verb(plural,Z,Y,f(V,Z,Y),[W|X],X):-
        verb_be(V,W),
        plural(W).

trans_verb(singular,Z,Y,f(X,Z,Y),[W|V],V):-
        verb(W,X),
        transitive(W).
trans_verb(plural,Z,Y,f(X,Z,Y),[W|V],V):-
        plural(U,W),
        verb(U,X),
        transitive(U).
trans_verb(plural,Z,Y,f(X,Z,Y),[W|V],V):-
        nonvar(W),
        name(W,U),
        append(U,[115],T),
        name(S,T),
        verb(S,X),
        transitive(S).
trans_verb(plural,Z,Y,f(X,Z,Y),[W|V],V):-
        var(W),
        verb(U,X),
        transitive(U),
        name(U,T),
        append(S,[115],T),
        name(W,S).
```

```
/********** INTRANS_VERB **********/
intrans_verb(singular,Z,f(Y,Z),[X|W],W):-
        verb(X,Y),
        intransitive(X).
intrans_verb(plural,Z,f(Y,Z),[X|W],W):-
        plural(V,X),
        verb(V,Y),
        intransitive(V).
intrans_verb(plural,Z,f(Y,Z),[X|W],W):-
        nonvar(X),
        name(X,V),
        append(V,[115],U),
        name(T,U),
        verb(T,Y),
        intransitive(T).
intrans_verb(plural,Z,f(Y,Z),[X|W],W):-
        var(X),
        verb(V,Y),
        intransitive(V),
        name(V,U),
        append(T,[115],U),
        name(X,T).


/********** QUAN_PRONOUN **********/
quan_pronoun(singular,Z,Y,
 exists(Z,indefinite(Z,coverage(Coverage)),f(W,Z)&Y),[X|V],V):-
        nonvar(X),
        pronoun(existential,X,W,Coverage).
quan_pronoun(singular,Z,Y,
    all(Z,indefinite(Z,coverage(Coverage)),f(W,Z)=>Y),[X|V],V):-
        nonvar(X),
        pronoun(universal,X,W,Coverage).
quan_pronoun(singular,Z,Y,
 exists(Z,indefinite(Z,coverage(Coverage)),f(W,Z)&Y),[X|V],V):-
        var(X),
        pronoun_coverage(existential,X,W,Coverage).
quan_pronoun(singular,Z,Y,
    all(Z,indefinite(Z,coverage(Coverage)),f(W,Z)=>Y),[X|V],V):-
        var(X),
        pronoun_coverage(universal,X,W,Coverage).

pronoun_coverage(Quantifier,X,W,Coverage):-
        integer(Coverage),
        pronoun(Quantifier,X,W,Coverage1),
        Coverage1=Coverage.
pronoun_coverage(Quantifier,X,W,Coverage):-
        integer(Coverage),
        pronoun(Quantifier,X,W,Coverage1),
        Coverage1<Coverage,!.
pronoun_coverage(Quantifier,X,W,Coverage):-
        nonvar(Coverage),
        pronoun(Quantifier,X,W,Coverage1),
        Coverage1=<50.
```

```
/********** CONJUNCTION **********/
conjunction(Z,Y,Z&Y,[and],[and|X],X):-
        true.
conjunction(Z,Y,Z#Y,[or],[or|X],X):-
        true.
conjunction(Z,Y,~(Z&Y),[nor],[nor|X],X):-
        true.


/********** CP_CLASS_NAME_NOT **********/

cp_class_name_not(Z,Y,~X,W,V):-
        class_name_not(Z,Y,X,W,V).



/********** CLASS_NAME_NOT **********/

class_name_not(singular,Z,Y,[a|X],W):-
        gnoun_not(singular,Z,Y,X,W).
class_name_not(singular,Z,Y,[an|X],W):-
        gnoun_not(singular,Z,Y,X,W).
class_name_not(plural,Z,Y,X,W):-
        gnoun_not(plural,Z,Y,X,W).
class_name_not(Z,Y,X,W,V):-
        gadjective(Z,Y,X,W,V).


/********** NOUN_PHRASE_NOT **********/

noun_phrase_not(Z,Y,X,W,V,U):-
        gdeterminer_no(Z,Y,T,S,W,V,R),
        gnoun(Z,Y,T,R,Q),
        relative_clause(Z,Y,X,S,Q,U).
noun_phrase_not(Z,Y,~X,W,V,U):-
        gdeterminer_no(Z,Y,T,~X,W,V,S),
        gnoun(Z,Y,T,S,U).
noun_phrase_not(Z,Y,X,W,V,U):-
        gdeterminer_no(Z,Y,T,X,W,V,S),
        gnoun(Z,Y,T,S,U).

/********** CP_VERB_PHRASE_NOT **********/
cp_verb_phrase_not(Z,Y,X,W,V):-
        verb_phrase_not(Z,Y,X,W,V).

/********** VERB_PHRASE_NOT **********/
verb_phrase_not(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        negatif(T,S),
verb_phrase_not(Z,Y,X,W,V):-
        auxiliary(Z,W,U),
        negatif(U,T),
        trans_verb(plural,Y,S,R,T,Q),
        R\=f(is,_,_),
        cp_noun_phrase(P,S,R,X,Q,V).
        cp_class_name(Z,Y,X,S,V).
```

```
verb_phrase_not(Z,Y,X,W,V):-
        auxiliary(Z,W,U),
        negatif(U,T),
        intrans_verb(plural,Y,X,T,V).
verb_phrase_not(Z,Y,X&W,V,U):-
        auxiliary(Z,V,T),
        negatif(T,S),
        intrans_verb(plural,Y,X,S,R),
        glocator(Y,W,R,U).
verb_phrase_not(Z,Y,X,W,V):-
        auxiliary(Z,W,U),
        trans_verb(plural,Y,T,S,U,R),
        S\=f(is,_,_),
        cp_noun_phrase(Q,T,~S,X,R,V).
verb_phrase_not(Z,Y,~X,W,V):-
        auxiliary(Z,W,U),
        intrans_verb(plural,Y,X,U,V).
verb_phrase_not(Z,Y,~(X&W),V,U):-
        auxiliary(Z,V,T),
        intrans_verb(plural,Y,X,T,S),
        glocator(Y,W,S,U).
verb_phrase_not(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,f(is,Y,U),W,T),
        cp_class_name_not(Z,Y,X,T,V).
verb_phrase_not(Z,Y,X,W,V):-
        trans_verb(Z,Y,U,T,W,S),
        cp_noun_phrase(R,U,~T,X,S,V).
verb_phrase_not(Z,Y,~X,W,V):-
        intrans_verb(Z,Y,X,W,V).
verb_phrase_not(Z,Y,~(X&W),V,U):-
        intrans_verb(Z,Y,X,V,T),
        glocator(Y,W,T,U).


/********** NEGATIF **********/
negatif([not|Z],Z):-
        true.

/********** AUXILIARY **********/
auxiliary(plural,[do|Z],Z):-
        true.
auxiliary(singular,[does|Z],Z):-
        true.

/********** GDETERMINER_NO **********/
gdeterminer_no(singular,Z,Y,X,
    all(Z,indefinite(Z,coverage(50)),Y=>X),[no|W],W).
gdeterminer_no(singular,Z,Y,X,
    all(Z,indefinite(Z,coverage(Coverage)),Y=>X),[No|W],W):-
        var(No),
        Coverage=<50,
        No=no.
```

```
/********** GADJECTIVE_NOT **********/
gadjective_not(Z,Y,X,W,V):-
        gadjective_not0(Z,Y,X,W,V).


/********** GADJECTIVE_NOT0 **********/
gadjective_not0(Z,Y,f(X,Y),[W|V],V):-
        adjective(W,X).


/********** QUAN_PRONOUN_NOT **********/

quan_pronoun_not(singular,Z,Y,
 exists(Z,indefinite(Z,coverage(Coverage)),f(W,Z)&Y),[X|V],V):-
        nonvar(X),
        pronoun(existential,X,W,Coverage).
quan_pronoun_not(singular,Z,Y,
   all(Z,indefinite(Z,coverage(Coverage)),f(W,Z)=>Y),[X|V],V):-
        nonvar(X),
        pronoun(universal,X,W,Coverage).
quan_pronoun_not(singular,Z,Y,
 exists(Z,indefinite(Z,coverage(Coverage)),f(W,Z)&Y),[X|V],V):-
        var(X),
        pronoun_coverage(existential,X,W,Coverage).
quan_pronoun_not(singular,Z,Y,
  all(Z,indefinite(Z,coverage(Coverage)),f(W,Z)=>Y),[X|V],V):-
        var(X),
        pronoun_coverage(universal,X,W,Coverage).



/********** GNOUN_NOT **********/

gnoun_not(Z,Y,X,W,V):-
        gnoun0(Z,Y,X,W,V).
```