886

A PORTABLE NATURAL LANGUAGE INTERFACE TO INGRES

642

DISSERTATION SUBMITTED BY

GURJEET SINGH KHANUJA

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF TECHNOLOGY

IN

COMPUTER SCIENCE AND TECHNOLOGY

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES

JAWAHARLAL NEHRU UNIVERSITY

NEW DELHI

DECEMBER 1990

CERTIFICATE

This is to certify that the dissertation entitled **"A Portable Natural Language Interface to INGRES"**, being submitted by me to Jawaharlal Nehru University in the partial fulfilment of the requirements for the award of the degree of **Master of Technology**, is a record of original work done by me under the supervision of **Dr. P. C. Saxena**, Associate Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University during the year 1990, Monsoon Semester.

The results reported in this dissertation have not been submitted in part or full to any other University or Institute for the award of any degree or diploma, etc.

Khanuja

GURJEET SINGH KHANUJA

[illegible]

Prof. N. P. Mukherjee
Dean,
School of Computer and
Systems Sciences,
J.N.U.,
New Delhi.

[illegible]

Dr. P. C. Saxena
Associate Professor,
School of Computer and
Systems Sciences,
J.N.U.,
New Delhi.

ACKNOWLEDGEMENT

I express my sincere gratitude to my supervisor **Dr. P. C. Saxena** for his uncompromising guidance, constant supervision and constructive criticism without which this work would not have been completed successfully.

I extend my sincere thanks to **Prof. N. P. Mukherjee**, Dean, School of Computer and Systems Sciences, Jawaharlal Nehru University for his encouragement and facilities provided for the completion of this work.

I also take this opportunity to thank all faculty and staff members and my friends who have been directly or indirectly helpful in eliminating a variety of problems encountered by me in the course of completing this dissertation.

GURJEET SINGH KHANUJA

CONTENTS

PREFACE

In this work, we are presenting a system which uses Tree Adjoining Grammar (TAG) to provide a natural language interface to the relational database system INGRES. Work was been done to determine what basic trees had to be used as input to the parser, what information had to be stored in the lexicon to support the translation of a parse tree into a database query and what algorithms were necessary to perform the translation process. A successful implementation in PROLOG has been done. Examples are given. Also, the interface is structured in such a way that much of it can be used in a front_end to database query languages other than INGRES.

Chapter 1 of this thesis covers the problems involved in understanding natural languages. In this chapter we will describe the different techniques used in computer world to understand natural languages. Chapter 2 describes natural language interfacing to databases, and their advantages. It also describes in brief the work which has been done on natural language interfacing to databases. Chapter 3 of the thesis deals with the general problem of parsing and understanding natural languages with emphasis on the database environment. Some parsing techniques other than those employing tree adjoining grammars are mentioned. Then tree adjoining grammars are discussed in detail. Chapter 4 describes the system implemented in brief. Chapter 5 describes the syntactic aspects of the system. It covers how a lexicon is set up for a given application along with parsing mechanism. The tree sets used by the system are also given in this chapter. Chapter 6 describes procedures used in translating the parse tree to QUEL queries. The details of translation with some examples are given in this chapter.

CH. 1 INTRODUCTION

In this chapter, we briefly describe the different issues regarding Natural Language Understanding (NLU), viz. meaning of NLU, what are the difficulties in understanding Natural Languages, how to interprete natural language input to a computer. There are many approaches to these problems which are highlighted in the second section of this chapter.

1.1 Natural Language Understanding :

Natural Language Understanding has been an important topic of interest within linguistics, Artificial Intelligence, Database Design and Information retrieval for quite some time. The goal of natural language understanding is not to have computers understand everything we say; after all, even people misunderstand each other occasionally. The understanding of natural language is very difficult because syntactic and semantic issues are very tightly coupled, vocabularies are very large, vagueness and ambiguities abound, and many of the syntactic constructs are very awkward. Problems in natural language understanding are _

- *Ambiguity* : Many of the things we say can be interpreted in more than one ways. Some of the factors that contribute to the ambiguity of natural language are as follows :-
  * Multiple Word Meaning : It is not uncommmon for a single word to have more than one meaning.

* Syntactic Ambiguity : Some of the ambiguity in English is caused by peculiarities in its syntax.

* Unclear Antecedents : We frequently use pronouns in place of previously used nouns.

_ *Imprecision* : People often express concepts with vague and inexact terminology. For example, how long is *a long time* ?

_ *Incompleteness* : We do not always say all of what we mean. Because we share many details without fear of being misunderstood; we assume that our listeners can "read between the lines."

_ *Inaccuracy* : It includes mistakes in any of the following areas,

* Spelling errors,
* Transposed words,
* Ungrammatical constructions,
* Incorrect syntax,
* Incomplete sentence and/or
* Improper punctuation.

Real world applications tend to concentrate on areas where at least one of the following simplifying assumptions apply :

(1) Complete understanding is not required, as in automatic document indexing.

(2) The vocabulary, syntax and semantics are very restricted, such as with database.

Our efforts here are focused on the second of the two simplifications. Specifically, we have developed a natural language interface, which is currently being used as a front_end to the general relational database system INGRES.

1.2 Techniques for Natural Language Interpretation :

Several techniques are available for interpreting natural language input to a computer. The objectives of all are probably similar; to extract semantic meaning from human input. A number of issues must be resolved before choosing an appropriate method. These include _

* Size of vocabulary.
* Use of non_grammatical input.
* The number of different users of the systems.
* Whether the principle user population is composed of regular or occasional users.
* The extent to which each individual sentence within that session will refer to others within that session.
* Whether user population will mature with time.

1.2.1 Traditional Approaches :

The simplest approach to interpreting the meaning of natural language is to seperate the analysis of syntax from the semantic. It appears, at first sight, reasonable that all grammatical tree structures using certain rules describe what costitutes a well formed sentence.

The semantic analysis (extracting meaning) then follows from the result of this syntactic analysis. The syntactic parse should indicate such features as whether the verb is passive or active, the subject or object of the sentence and so on. The meaning can therefore be easily drawn out.

The process has the advantage of simplicity. The approach is well-suited to simple sentences in tightly constrained domains. A-user is able to build a comprehensive set of rules and ensure that these are followed. New words can be added and defined in terms which the system understands.

However, there are disadvantages with this _

(1) A given sentence may be capable of parsed into more than one way.

(2) It is not possible to handle ungrammatical text in this fashion.

(3) Certain sentence types, whilst grammatical, pose problems.

1.2.2 Transition Networks :

Transition networks operate essentially by working through a sentence from left to right. For each word there are only a limited number of words which can grammatically follow it. For example, an adjective is only likely to be followed by another adjective or by a noun. The network which the possible routes form from a given start point to a completed noun or verb phrase form are described as a transition network.

This simple network is inadequate. A recursive facility enables the system to handle subordinate phrases. Also useful are registers which can be used to record information obtained in the analysis of one phrase which can affect other phrases. Such an enhanced system is known as an Augmented Transition Network (ATN).

1.2.3 Chart Parsing :

Chart parsing is an approach which facilitates the building up of structure from small blocks (a bottom_up approach). Previously described techniques use top_down approach. The disadvantage of the top_down approach is that it can be very time-consuming if an erroneous assumption is made early in the parsing process.

Another advantage is that it may allow the extraction of some information from ungrammatical input.

There is, however, a major efficiency problem. A large number of irrelevant "builbing blocks" may be created. Because of this problem, chart processing is generally used only in conjuction with some other approach to syntactic parsing.

1.2.4 Case Grammar :

This technique represents a departure from what has gone before, in that it uses some semantic information.

The case grammar ideas spring from the view of a sentence as a description of some underlying event. Hence, associated with a given verb, one can describe a case frame which has several slots or cases. Each case specifies a participant in the event. So, for example, the "go" will have a compulsory "Actor" case to indicate who or what is going and further optional cases to indicate where the Actor is going to, or perhaps what colour the Actor is going to.

CH. 2 NATURAL LANGUAGE INTERFACING

This chapter covers the advantages of Natural Language Interfacing (NLI), especially to relational databases. It deals with the portability and transportability of a natural language system in the context of relational databases. In the last section, we describe a brief review of commercially available software.

2.1 Natural Language Interfacing :

There are many arguments for providing a natural language front_end to a relational database system. One of the most common reason is to provide *naturalness*. It can also provide greater comprehensibility of obtaining information stored in the database by querying the system in a natural language. With a natural language as a means of communication with computer systems user can frame a question or statement in the way they normally think about the information being discussed, freeing them from having to know how the computer stores or accesses the information. Or it allows the user to make queries of a database without the need to understand the database's internal organisation. It helps the user to formulate queries and generating queries for the database (converting a task specification into package instructions).

2.2 Transportability & Portability :

Most existing natural language interface systems have been designed specifically to treat queries that are costrained in two ways -

(1) they are concerned with a single application domain and

(2) pertain to information in a single database.

Costruction of a system for a new domain or database requires a sizeable new effort, almost equal in magnitude to the original one.

Transportable natural language interfaces, i.e. those that can be easily adapted to new domains or databases, are potentially much more useful than domain or database specific systems.

A major challange in building natural language interfaces (NLIs) is to provide the information needed to bridge the gap between the way the user thinks about the domain of discourse and the way information about the domains is structured for computer processing.

The databases may employ different representations, or different encodings but an NLI should be able to handle queries for any of the encoding. Although the English query input to the NLI is the same in all cases, the NLI output (i.e. specific commands to a database system to retrieve the requested information) will be quite different for the different encodings. One of the main functions of the NLI is to make the necessary transformations and thus to insulate the user from the particularities of the database. To provide this insulation & bridge the gap between the user's view and the systems's data structures requires a combination of domain specific and general information. In particular, the system must have a model of the application domain's subject matter, including information about the objects in the domain, the properties they possess and their interrelationships and the words and phrases used to refer to each of these; the system must also know the connection between entities in that model and the information in the database. In constructing transportable systems it is therefore important to provide a means for acquiring domain specific information easily.

Our system can easily be adapted to a given user database. Also, much of this system can be used without change to provide an interface to systems other than INGRES.

This system processes a user's English question into a parse tree. Then this parse tree is translated to an intermediate code called a pseudo query and passes it to INGRES. The use of the intermediate pseudo query makes the system *portable*. Conversion for use with a different query language is accomplished by providing routines which convert the pseudo query into a query in new language.

2.3 Recent Work on Natural Language Processing and Databases :

The commercial systems are developed in various parts of the world. Few of them are described below.

2.3.1 LUNAR SYSTEM :

It is a Natural Language Interface to Moon Rocks Database by Woods,1973 at BBN.

The system uses a small vocabulary (3500 words) required for moon rock database. The LUNAR database uses encoding in the database query language. In this, there were seven data domains. Sets of elements that could be members of each domain were mutually exclusive. The system used a powerful ATN syntactic parser. It parsed sentence on to the semantic program for translation into a query. The resulting query was then executed. The semantic analyzer gathers information from verbs and their cases, nouns, noun modifiers and determiners to build the database query. The query is built in terms of conceptual primitives. The database uses rules to compare the syntactic structure of the question with a syntactic template. If they match, the syntactic part of the rule is added to the developing query.

The system can handle anaphoric references (pronoun reference to previous phrases). It could handle 90% of the questions posed to LUNAR by geologists. Its overall formulation is so clean and neat that it has been used for most parsing and language understanding systems.

Limitations : Utterance were limited to database queries. This was non portable and non extensible. It is no longer in use.

2.3.2 PLANES/JETS SYSTEM :

PLANES/JETS is a natural language interface to a large database developed by Waltx DL. in 1975 at MIT.

Database was created for the maintenance of flight recorder for all novel aircrafts. It ignores syntax and assumes that all inputs are in the form of requests that it turns into formal language query extensions. It uses a semantic grammar. It looks for semantic constituents by doing a left to right scan of the user's sentence. Semantic constituents include items which belong to PLANE TYPE, TIME PERIOD, MALFUNCTION, CODE, HOW MANY, ACTION etc. It uses an ATN parser. The top level calls various subnets to analyze the input for semantic constituents . It utilizes concept-case frames which are string of constituents of reasonable queries. After application of the concept-case frames the resulting syntactic costituents are passed along with the query generator.

It can handle ellipses and pronouns and also deals with nongrammatical sentences. System asks for a rephrase if it doesn't understand.

Limitations : It was relatively inefficient and relies too heavily on its particular world of discourse for eliminating problems of world since selection.

2.3.3 ROBOT INTELLECT SYSTEM :

Robot-Intellect is a database question answering system developed by Harris in 1977 at Dartmouth.

It was an ATN syntactic parser (with backtracking) followed by semantic analysis to produce a formal query language representation of the input sentence. It handles a large vocabulary by building an inverted file of data element names indicating the data domains in which each name occurs. In addition, the inverted file contains words and phrases that are interpreted as data element names. A dictionary of common English words is also included. If two meanings of the inquiry appear likely and only returns that one which is interpreted to be the appropriate one.

Intellect is one of (with the first natural language database query) the systems to be available commercially. It can handle idioms via special mechanisms. It can handle some pronouns and ellipses.

Limitations : It does not consular context except to disambiguate pronouns and ellipsis.

2.3.4 TEAM :

This system is one of the ealiest to have laid emphasis on transportability of the interface across different domains.

TEAM is designed to interact with two kinds of users -

_ a database expert and

_ an end_user.

The database is created through a system-directed acquisition dialogue. As a result of this dialogue the language processing and data access components are extended so that the end_user may query the new database in natural language.

The system has three major components _

(1) The acquisition component,

(2) The DIALOGUE language system and

(3) A data access component.

The translation of an English query into a database query takes place in two stages. First, the DIALOGUE system costructs a representation of the literal meaning or the logical form into a formal database query. Each of these steps requires a combination of information that is dependant on the domains and information that is not. To provide for transportability, TEAM carefully distinguishes between the two.

CH. 3 TREE ADJOINING GRAMMAR AND OTHER SYNTAX MODELS

We have used Tree Adjoining Grammar (TAG) to provide a natural language interface to the relational database system INGRES. Tree adjoining grammars are properly more powerful than context free grammars. They perform very well in a database query environment because they have reasonable efficient parsing algorithms and they support the concept of nested queries very well. In the first section of this chapter we discuss about the grammars used for natural language understanding in the light of Tree Adjoining Grammar. In the last section we deal with Tree Adjoining Grammar in detail.

3.1 Grammars for Natural Language Understanding :

Several formal models for the expression of the syntax and semantics of the English language have been tried. Context free grammars are popular within the realm of artificial languages, such as computer programming languages , because of their elegance, simplicity and the availability of automatic parser generators such as YACC. General context free grammars can be parsed in time $O(n^3)$ where "n" is the number of words in a sentence and sufficiently orderly grammars can be parsed in linear time. Unfortunately, context free grammars alone are not powerful enough for the syntactic and semantic analysis required for natural language understanding.

Several more powerful syntactic models for natural languages have been considered and studies have been made to see how they interact with semantic analysis. Although syntactic analysis for structures more general than context free grammar can be very slow, progress on the problem of automatic parser generation for context sensitive grammars has been made. Several extensions of context free languages have been applied in the natural language environment.

One important extension of context free grammars is the DIAGRAM grammar. DIAGRAM is a large phrase structure grammar with rule_procedure added to it. The rule procedures allow phrases to inherit attributes from their constituents and from surrounding, large phrases. Context sensitive constraints may be imposed which provide consistency conditions and information on dominance. DIAGRAM has been used as the basis for the portable natural language database interface TEAM. Research on TEAM investigated the problem of providing a natural language interface which can be adapted to new database by personnel that are not themselves natural language processing experts.

Perhaps the most widely used model for natural language syntax is the Augmented Transition Net (ATN). The ATN is the syntactic basis for the CO_OP database interface and the YANLI natural language front_end. A transition net is a collection of nodes and directed arcs called links which describe syntactic structures (sentence, noun phrase, prepositional phrase, etc.) and arcs have labels which can be word categories (noun, verb, preposition etc.) or syntactic structures defined by other subnets. For example an ATN can easily express the facts that a sentence can be a noun phrase followed by a verb phrase and that a noun phrase can be a determiner followed by any number of adjectives, followed by a noun. Sentences are parsed by the arcs appropriate to the word categories in the sentence. An augmented transition net is a transition net which can store information as it goes and use this information in making decisions on syntactic structure. This gives the ATN a powerful means of combining syntactic and semantic analysis.

The Wait-And-See Parser (WASP) is a more sophisticated kind of natural language parser. A WASP first defines noun phrase in the sentence and then proceeds to group the noun phrases into a parse tree using the other words in the sentence and a collection of rules as a guide. While building a parse tree for a sentence a WASP maintains three data structures : a node stack which contains nodes in the parse tree, a buffer containing words and noun phrases which have not yet been placed in the buffer. The list of rules tells the WASP when to perform the following operations : move words or phrases into the buffer, when to use buffer contents to create a node and push it on the node stack and when to reduce a contiguous set of nodes on the stack into a single node.

One of the chief difficulties in natural language understanding is the fact that there is no boundary on the distance seperating two related nodes in the parse tree for a sentence. For example, figure 3.1 is a tree diagram of the question "What does a printer weigh ?" The word "what" has a significant relationship to the empty leaf node,"E", which acts as the object of the verb "weighs". Specifically, "what" can be said to function as the object of the verb in place of the empty node.

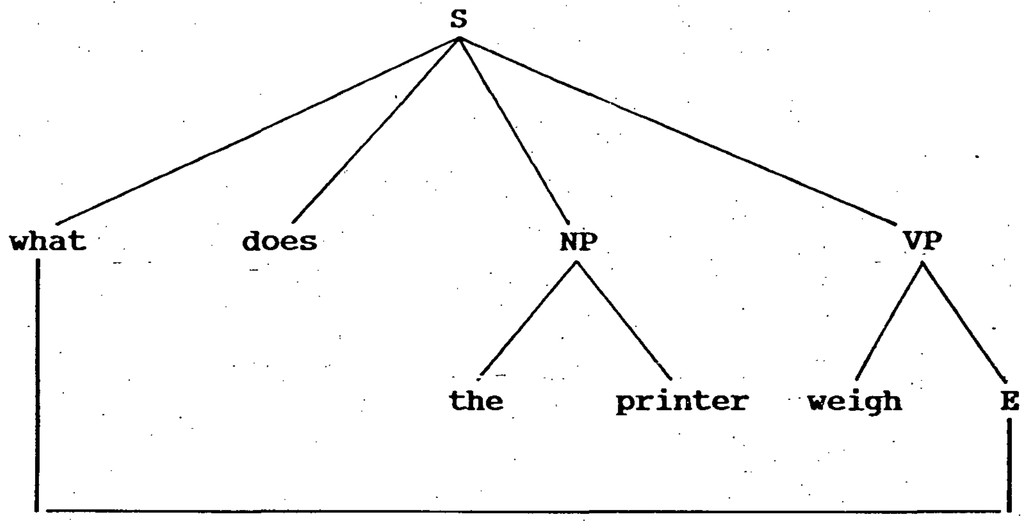


Figure 3.1

Transformational grammars attempt to model these relationships in terms of subtree movements called transformations. Informally, a transformational grammar is a context free grammar to which certain context sensitive rules have been added that allow for the rearrangement of subtrees in a parse tree. When the question of fig 3.1 is generated using an appropriate transformational grammar the word "what" starts out as the object of the verb and is moved to the beginning of the sentence by a transformation (as shown in figure 3.1 by dotted line). At some level of generation the sentence is "A pointer does weigh what ?" Two transformations are performed on this structure WH_MOVEMENT brings "what" to the front of the sentence and NP_AUX_INVERSION swiches the order of a "printer" and "does". The use of transformational grammar to model natural language has provided useful linguistic insights. It seems that transformational grammars have more generative power than is necessary for modelling natural languages.

3.2 Tree Adjoining Grammar :

We have used tree adjoining grammars for three reasons. First, they have parsing algorithms that are reasonably efficient. Second, they have general syntactic modelling power that is adequate for providing a natural language database interface. Finally, an interface based upon TAGs can be easily adapted to a variety of different user's databases with minimum amount of effort.

Tree adjoining grammars can be parsed in $O(n^4)$ and they have many liguistically significant features, such as having no boundary on the distance between related nodes.

A formal account of generative power of tree adjoining grammars and these results may be summarized as follows _

(1) For every context free grammar there is a TAG which defines the same set of sentences and the same set of parse trees.

(2) There exists a context free grammar ,G_{cfg}, and a TAG, G^{tag}, which defines more parse trees for certain sentences.

(3) There exists a TAG which defines a non context free language.

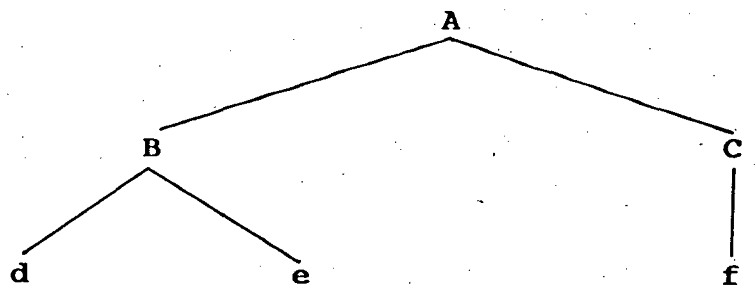
(4) Every language defined by a TAG is context sensitive.

(5) There exists a context sensitive language which is not defined by TAG.

Unlike other grammars, there are no production rules for TAG's. Rather TAG, G, consists of two sets of trees. Symbols in Tree adjoining grammar are defined as terminals or non_terminals and only non_terminals may appear as interior nodes in trees. The set of trees which are defined by the grammar G are all of those found in the initial tree set plus all of those which can be created from the initial trees through a process called adjoining.

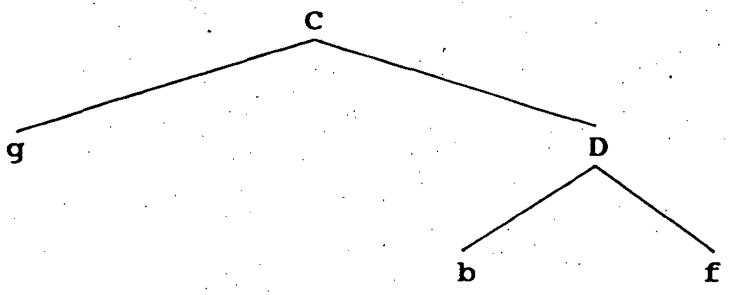
Adjoining : In the adjoining process, a node within a tree is removed along with all of its descendant nodes. In its place, an auxiliary tree whose root is the same as the removed node is inserted. Additional adjoining can occur any where in this large tree.

Let us consider a tree adjoining grammar, G having following sets of trees. The initial tree set has only one tree _

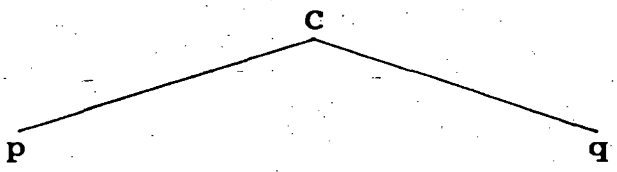


(Initial tree)

Auxiliary tree set has two trees shown below _

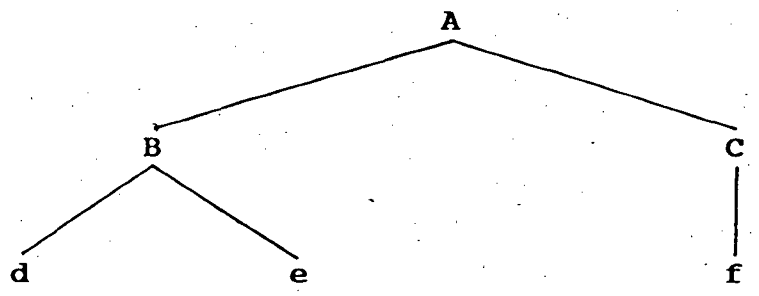


(tree no. 1)

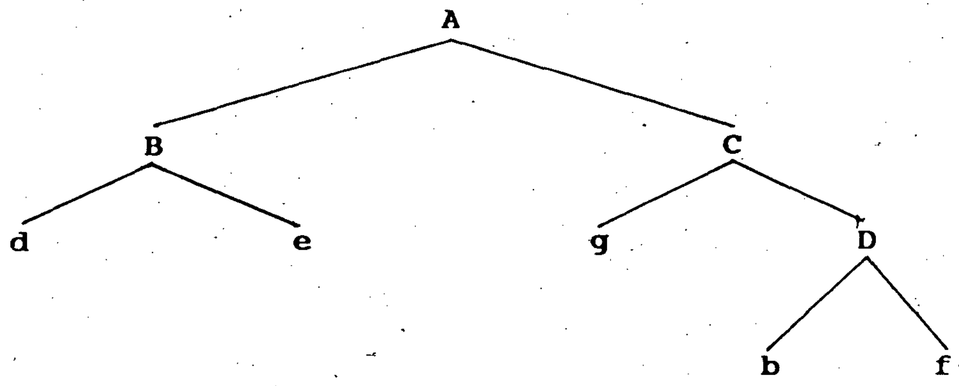


(tree no.2)

L(G), the language defined by G, that is set of all sentences defined by parse trees generated by G. Therefore L(G) contains following sentences _ "def", "degbf", "depq". The corresponding trees are shown below. For each adjoining operation, the newly inserted auxiliary tree is surrounded by dotted lines.

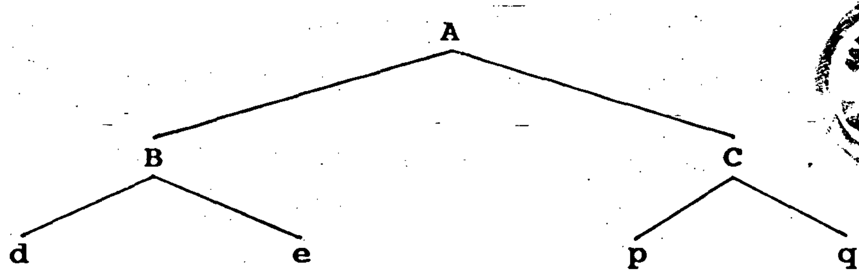


(for the sentence "deg")



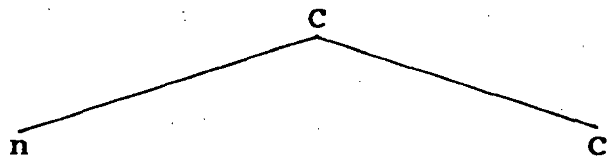
(for sentence "degbf")

TH-3655



(for sentence "depq")

Nested structures can also be handled very easily. Suppose, there is one more auxiliary tree shown below in the auxiliary tree set.



(tree no.3)

Then language defined by this grammar, G, is -

[def, de(n)*gbf, de(n)*pq] where "comma" represents union and (n)* is the closure of "n".

Diss
681.3.06
K527
Po.

CH. 4 SYSTEM OVERVIEW

In this chapter, first we briefly describe the system which has been implemented and then the different modules of the system in short. The detailed discussion will be in later chapters.

4.1 Brief Description of the System :

We have considered the application of the system on a given user database DBF (Given in the next section). The interface is composed of several active components : parser, translator, interpreter and another components which act as read-only files during the processing of a question and are known as lexicon files. The parser, translator, interpreter and tree adjoining grammar tree sets will remain the same for any user database for which the interface will be used.

A database administrator who wishes to use this system for a given application will perform the following tasks.

(1) Create the data definition of the relations.

(2) Create a lexicon entry for each additional word he wishes to be understood by the interface by specifying the part of speech of the word and a function which describes the condition that the word represents in the database.

(3) Load the database with data and maintain the data properly.

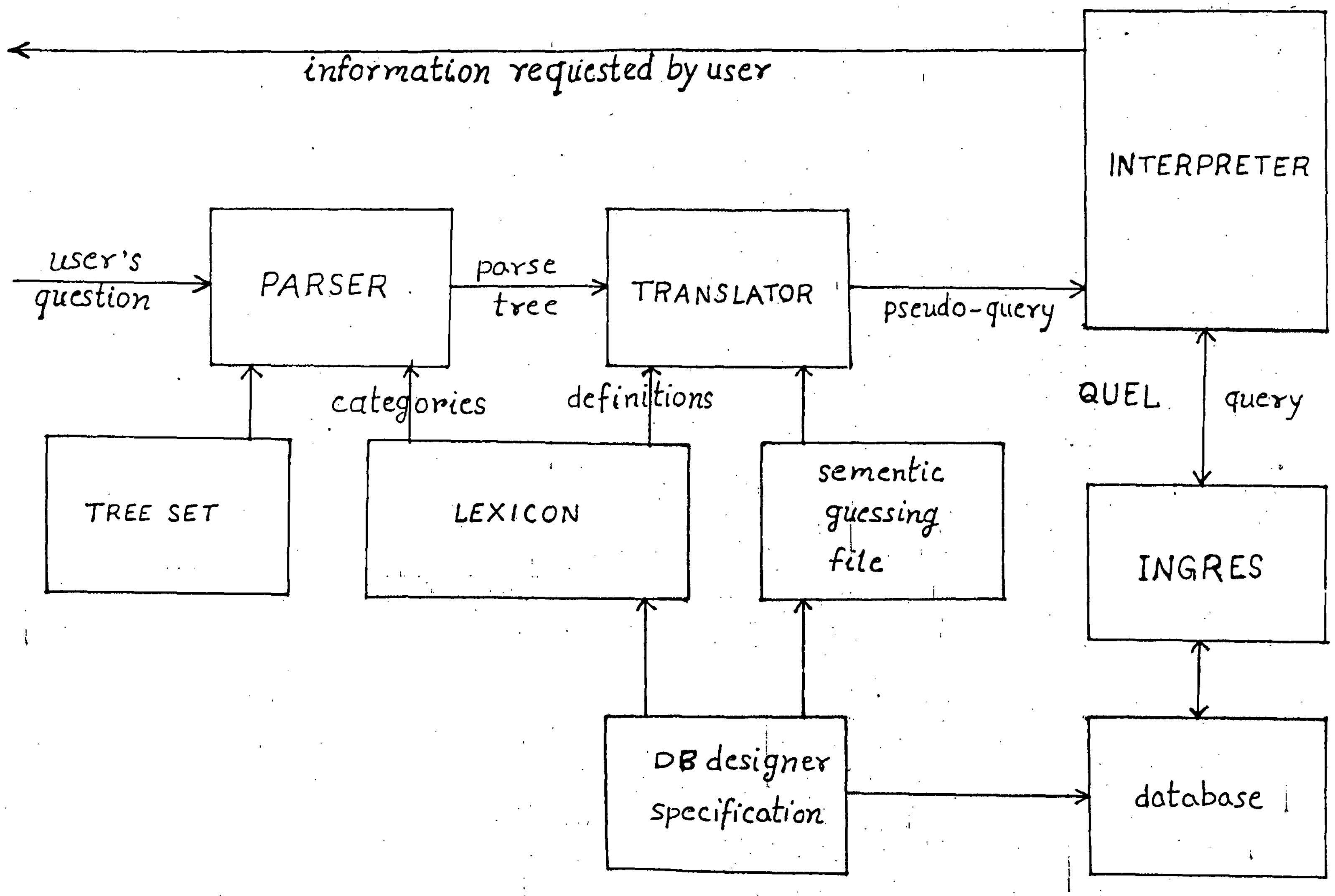

information requested by user
INTERPRETER
user's question
PARSER
parse tree
TRANSLATOR
pseudo-query
categories
definitions
QUEL
query
TREE SET
LEXICON
sementic guessing file
INGRES
DB designer specification
database

figure 4.1

Once the system is set up, users ask English questions and receive data in response. The system processes the question as follows.

First, they are sent to the parser. The parser accesses the tree set and the word category part of the lexicon to produce a tree adjoining grammar parse tree which represents the question. The parse tree is then sent to the translator which uses definitions of words in the lexicon to construct a pseudo query. It is then the job of the interpreter to phrase the pseudo query as a QUEL query, the language supported by INGRES and the results are returned to the user. Figure 4.1 illustrates the entire structure of the system.

4.2 A Sample Relation :

We have taken a sample relation to describe the working of our system. The table which we have created using INGRES is named DBF. It is a database containing information about the students of a school. This table has five attributes. The attributes name and their corresponding data formats are given below -

Column Name	Data Format
NAME	TEXT(25)
AGE	INTEGER(3)
ADDRESS	TEXT(40)
CLASS	TEXT(15)
GRADE	TEXT(1)

Steps involved in creating this relation using INGRES are -

(1) Invoke INGRES/MENU.

(2) Select TABLES option from the main menu.

(3) Select CREATE to create a new table.

(4) Enter the name of the table as DBF.

(5) Move the cursor to the table field, in the column labelled *Column Name*. Type in the name of the first field. Tab to column labeled *Data Format*. Enter the data format. Enter all the attributes' name in the same fashion.

(6) Select the *Save* menu item to save the table and its columns in the database.

Thus, the sample relation DBF has been created. Examples used in this dissertation and lexicon shown in Appendix (A) are based on this sample database. The system can be adapted to other databases, for that the only thing required is to change the lexicon.

4.3 MODULES OF THE SYSTEM :

The process of converting a natural language query into its equivalent QUEL query is divided into small processes called modules of the system. These modules are described, in brief, in this section.

4.3.1 Lexicon and Pseudo Query :

The first task of natural language is understanding each of the words in the sentence given. It can be achieved by maintaining a dictionary, also called a *lexicon*, which contains an entry for each word giving the target representation of the meaning of the word. Unfortunately, many words have several meanings and it may not be possible to choose the correct one just by looking at the word itself. So, it is the job of database designer to make entry for each possible meaning of a word and give preferences to the meanings.

For example, the word *diamond* might have the following set of meanings -

* A geometrical shape with four equal sides.
* A base ball field.
* An extremely hard and valuable gemstone.

If a database designer is writing a database for a base ball game than he should give more weightage to the second meaning of the word *diamond* than the other two.

The lexicon acts as a mapping between the user database and the English language. The sample of lexicon is given in figure 4.2. The first field in the lexicon is the word, i.e. *terminal*. The second field is the word category, i.e the category to which that word belongs to. "WH" category contains the question word, for example - *what, where, who, how* etc. The third and last field of lexicon is the *definition* field. It can be a function name or a definition given in the form of pseudo code.

WORD	CATEGORY	DEFINITION	
Whose	WH	FROM	DBF
		ID	DBF.NAME
		PRT	DBF.NAME
		WHERE	NIL
age	NOUN	FROM	DBF
		ID	DBF
		PRT	NIL
		WHERE	NIL
is	VERB	FUNCTION(INSTANCE)	

Figure (4.2)

The pseudo query data structure (see figure 4.3) contains a list of relations on which operations are being performed (the FROM part), a list of database fields indicating the information being required (the PRT part) and a list of conditions which must hold for each instance of relation (the WHERE part). These three fields are close in function to FROM, SELECT, and WHERE in SQL and RANGE, RETRIEVE and WHERE in QUEL. It makes the job of the interpreter easier. Pseudo query includes a fourth field ID which is a sentence (pertaining to meaning) component neccessary for the translation process.

PSEUDO QUERY			
FROM	ID	WHERE	PRT

SQL		
FROM	SELECT	WHERE

QUEL		
RANGE	RETRIEVE	WHERE

Figure 4.3

4.3.2 The Parser :

Parsing is a method of seperating a sentence into its component parts, which is the computer's equivalent of diagramming a sentence into a **parse tree**. Parsing takes advantage of inherent regularities in natural language to ensure that the computer understands the precise function of each word in a sentence, as well as its relationship with each of the other words.

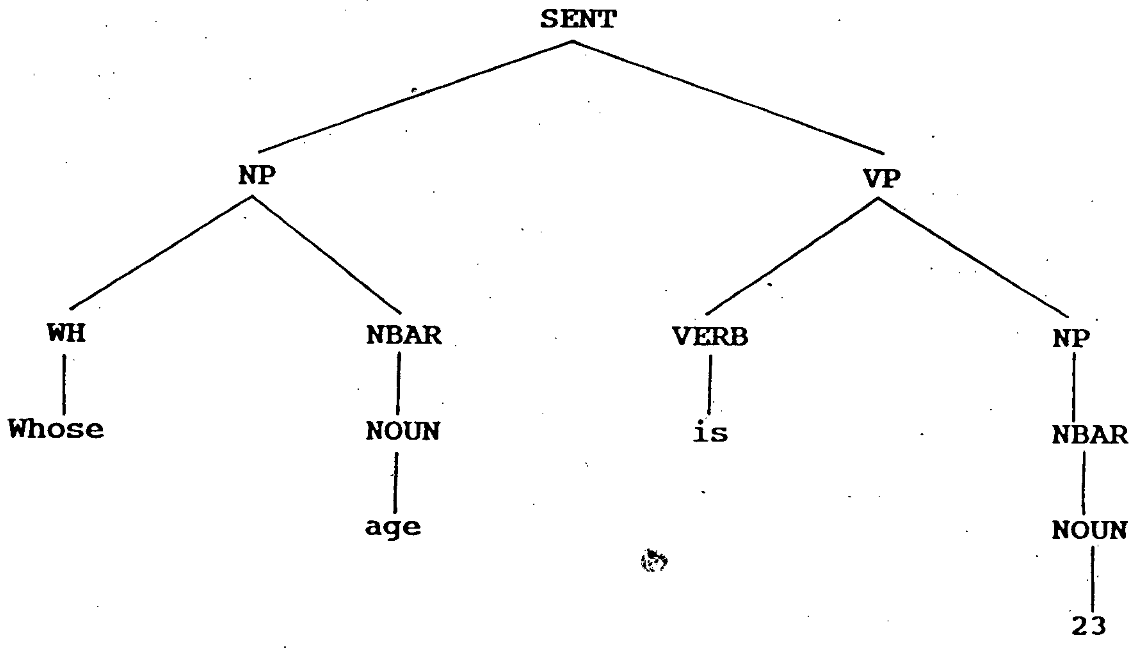
The input to the parser of our system is a English query, i.e. a string of words (terminals). There is a restriction that the query must start with a word which belongs to "WH" category or a question word. The parser generates an output which is actually the post-order traversal of a tree. With each node except the terminal and pre_terminal nodes, a number is attached. The purpose of this numeral is to show number of children attached to the node. For example a node *NP2* shows that it has two children attached to it. The pre_terminals can have one child only. Given the post order traversal and number of childern attached to each node, one can uniquely construct a parse tree. Let us consider a English query -

Whose age is 23 ?

The parser output would be -

Whose, WH, age, NOUN, NBAR1, NP2, is, VERB, 23, NOUN NBAR1, NP1, VP2, SENT2.

The diagrammatical representation of this parser output is shown in figure 4.5. Note that the post-order traversal of the tree is the same as the parser output.



figure(4.5)

4.3.3 Translator :

The translator and the interpreter of our system (see figure 4.1) are integrated into a single module. This module accepts the parse tree, i.e. the output of the parser and translates it into an intermediate structure called Pseudo query. Finally, a part of this module (*interpreter*) maps this pseudo query into an INGRES query and passes it to INGRES. The idea of first generating a pseudo query is to make the system ***Portable***. The conversion for use with a different query language is accomplished by providing routines which convert the pseudo query into a query in the new language.

Translation process is a bottom_up sequence of transformation of the parse tree. Each interior node i, the tree corresponds to a subroutine. We will discuss the function of these subroutines in Chapter 6.

The translator maintains a stack. It reads the inputted string, i.e. the parser output. If the word fetched is a terminal, it pushes it on to the stack. If it is a pre_terminal, (since it has one child in the tree), it pops the top of the stack. It should be a terminal, so it brings the definition of the word from lexicon and pushes it on to the stack. Since each interior node in the parse tree represents a subroutine and it operates on its predecessors. If the next symbol is a non_terminal then pop as many data from top of stack as the number attached to it. Then jump to the subroutine represented by the non_terminal and pass the data popped as parameters to that routine. The result of that subroutine is pushed onto the stack. This procedure is continued until the whole input string gets exhausted.

Let us consider the example taken in previous sub section again -

Whose age is 23?

The output of the parser was _

Whose, WH, age, NOUN, NBAR1, NP2, is, VERB, 23, NOUN, NBAR1, NP1, VP2, SENT2.

Applying the procedure described above for the translation, the output of the translator will be (the part of lexicon used for the translation porcess is shown in figure 4.3) -

FROM	*DBF*
ID	*DBF.NAME*
PRT	*DBF.NAME*
WHERE	*DBF.AGE = 23*

This is the equivalent pseudo query of the English question. This is then converted into QUEL query. This conversion is easy because the fields of pseudo query are close in function to fields of QUEL query. It is covered in Chapter 6.

CH. 5 SYNTACTIC ANALYSIS

The lexical and syntax analysis of our system is being covered in this chapter. First we discuss the design and implementation of the lexicon used by our system. Then we describe the parsing mechanism in detail. The tree sets or tree adjoining grammar used by this system are also given.

5.1 Design and Implementation of a Lexicon :

In this section, we will look at the design and implementation details of the lexicon used by the system.

5.1.1 Introduction :

A parser, for parsing a sentence, needs a dictionary for getting syntactic information about the words in the language. The collection of words along with the syntactic information constitute the *lexicon* of a parsing system. The information needed in the lexicon depends on the application. For natural language interface to a database the lexicon would differ from a conventional one (which gives only syntactic information), for this application it provides addition information regarding those words which have a special or restricted meaning in the domain of the database.

5.1.2 Design of the Lexicon :

We have divided the lexicon logically into two parts -

(A) Core lexicon and

(B) Database specific lexicon.

(A) *Core Lexicon* :

This part of the lexicon contains those words whose usage hardly ever changes across different domains. Example of such words are "WH" category word *what, who* or pronouns like *this*—only the syntactic information, which includes -

- the lexical category and
- feature dimensions.

The lexical category of a word is what we call *part of speech* in English grammar, like for the word *boy* the lexical category is *noun*. It is possible that a word might have one of several lexical categories depending on the way it is used. For instance, the word *play* in the sentence - "Let us *play* tennis" has the lexical category *verb* and in the sentence - "It was a good *play*", it has the lexical category *noun*.

(B) ***Database Specific Lexicon*** :

This constitutes of those words which have specific meaning with respect to the domain. For instance, the word *offer* in the domain of a *university* database would invariably mean the act of offering a course, by departments or the teachers in the departments. We see that the word has its meaning restricted. This information will not be used during the parsing stage, but will be pulled out of the lexicon and placed in the output frame of the parser (along with the word). It will be used by the subsequent module.

The words are the terminals in the grammar. Each words in the sentence is attached to its corresponding category. For this reason, the categories are often regarded as being *pre_terminals*. For each word, there is a definition entry in the lexicon. The definition entry could be either a subroutine name or a definition. In the lexicon nouns and adjectives always have definitions which are in the form of pseudo queries. Such words are called **object_words**. For example, the definition of the word *age* is a pseudo query which says that *age* corresponds to an attribute of the relation *DBF* called *DBF.age*. The *WHERE* and *PRT* parts of the pseudo query are *nil* because a mere reference to *age* does not imply the selection of any specific tuples in *DBF* or the request of any information in the lexicon as the name of a subroutine; the function of this subroutine is to copy the relation/field pair in the *ID* field into the *PRT* field. The role of these subroutines are explained in Ch. 6.

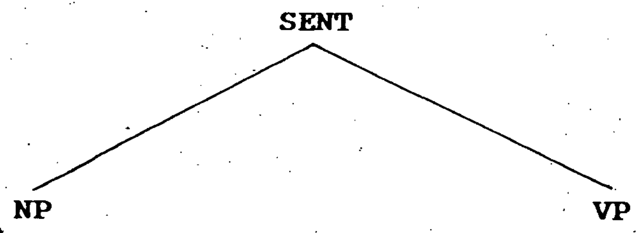
In the following section, we will describe the parser used by the system.

5.2 THE PARSER :

The syntactic analysis requires some kind of parsing techniques (a method of carving sentence into its component parts) which is the computer's equivalent of diagramming a sentence. Parsing takes advantage of inherent regularities in natural language to ensure that the computer understands the precise function of each word in a sentence as well as its relationship to each other word.

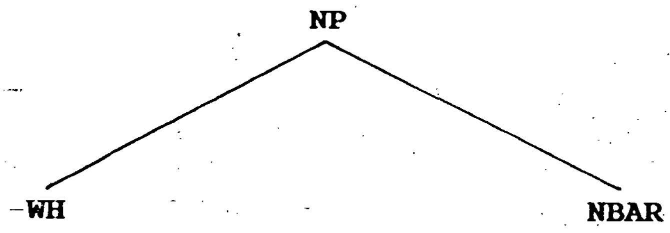
5.2.1 Tree Sets :

We employ a Tree Adjoining Grammar (TAG) parser for parse tree construction. Unlike other grammars, there are no production rules for TAG's. Rather a TAG consists of two sets of trees, the initial trees and auxiliary trees. Symbols in a TAG are identified as terminals or non_terminals and only non_terminals may appear as interior nodes in a tree. The set of trees defined by the grammar *G* are all those found in the set of initial trees plus all those which can be created from the initial ones by adding auxiliary trees through adjoining (already discussed in chapter 3 Sec. 3.2). The TAG which we used as part of our database interface is shown below; it consists of one initial tree and eleven auxiliary trees.

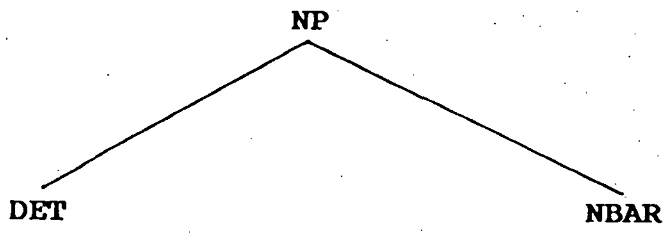


(Initial Tree)

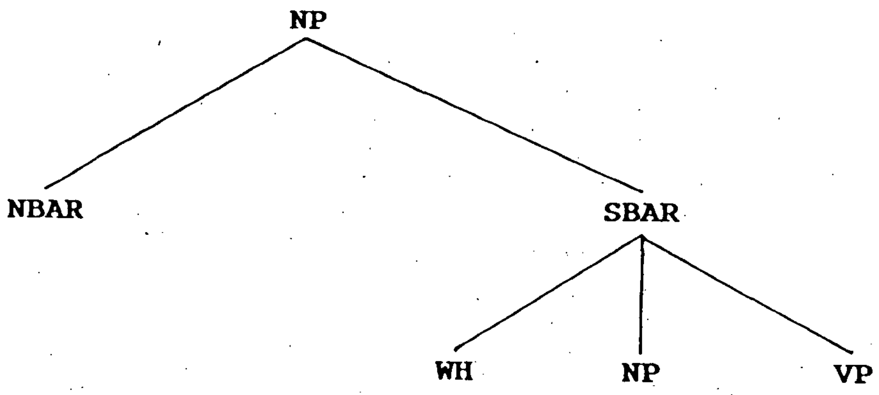
Auxiliary Tree set



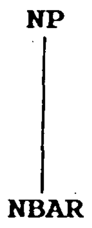
Aux. Tree (1)



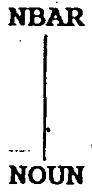
Aux. Tree (2)



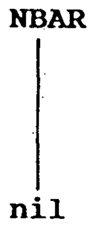
Aux. Tree (3)



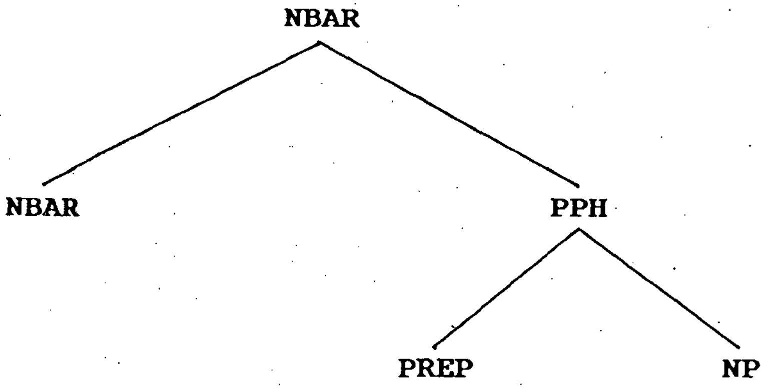
Aux. Tree (4)



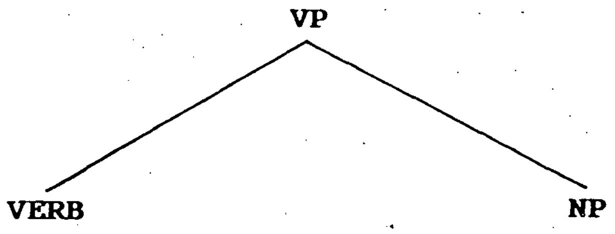
Aux. Tree (5)



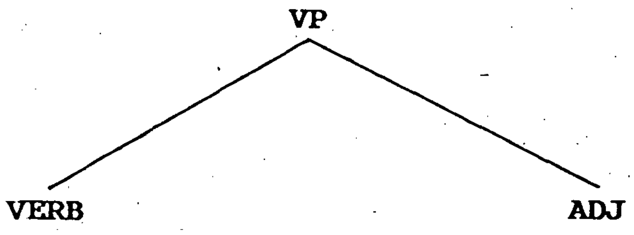
Aux. Tree (6)



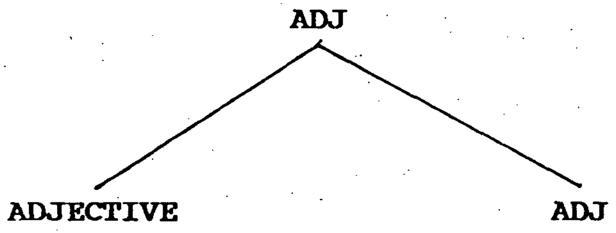
Aux. Tree (7)



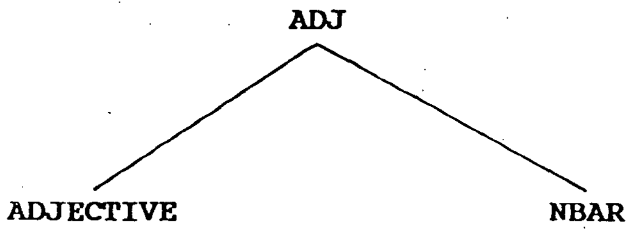
Aux. Tree (8)



Aux. Tree (9)



Aux. Tree (10)



Aux. Tree (11)

5.2.2 The Parsing Mechanism :

The parsing of the input string starts with the initial tree. This initial tree grows up using the process of adjoining, i.e. replacing a node and its corresponding sub_tree with another auxiliary tree whose root node is the same as the removed node. This process of adjoining has already been discussed in Chapter 3. Now, the tree thus obtained is traversed in an in_order manner. During traversal, if the leaf node is found to be a pre_terminal, i.e. of the word category such as noun, verb, preposition etc., then the next word of the input string is looked whether it is of the same category as the preterminal. For looking this, two fields of the lexicon (discussed in previous section) namely the ***word*** and ***category*** are used. The lexicon, in fact, tells about the category the word from the input string belongs to. However, the definition field is not used from the lexicon until the translation phase. If the word is found to be of the same category than this word, i.e. the terminal is removed from the input string and is attached to the pre_terminal in the tree.

The above process is continued until the whole input string gets exhausted. At this point, we have got the parse tree for the given input string.

The following sub_section describes the parsing mechanism in more detail. It also covers the programming constraint like "how" and "why". It is implemented in PROLOG.

5.2.3 Details of the Parsing Process :

We will consider the processing of an English question -

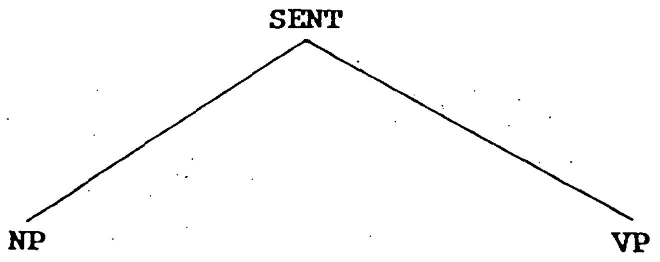
What is the age of Gurjeet?

The part of lexicon used by the parser to parse this query is shown in figure 5.2

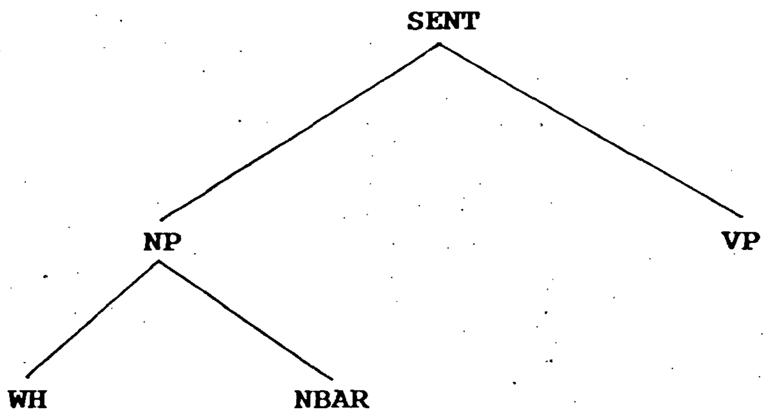
WORD	CATEGORY
What	WH
is	VERB
the	DET
of	PREP
Gurjeet	NOUN
age	NOUN

Figure 5.2

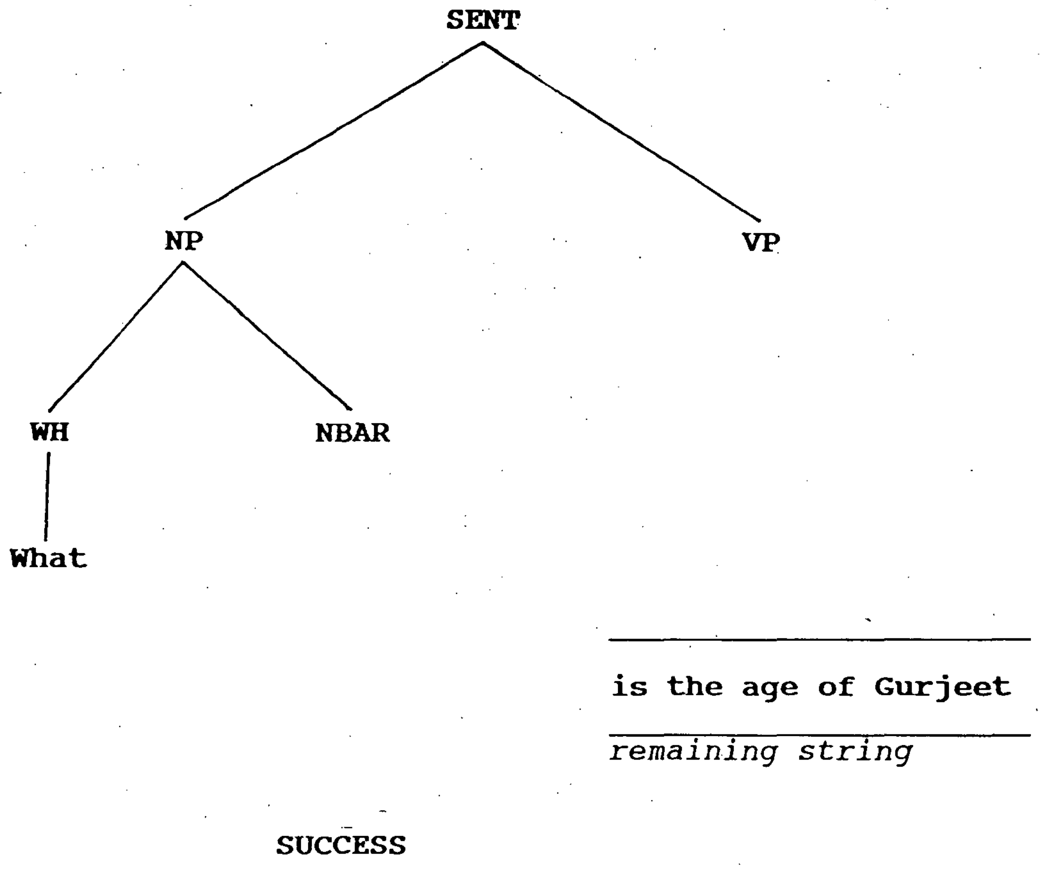
The parser always starts with initial tree. The auxiliary trees are written in the previous sub_section. At the beginning -



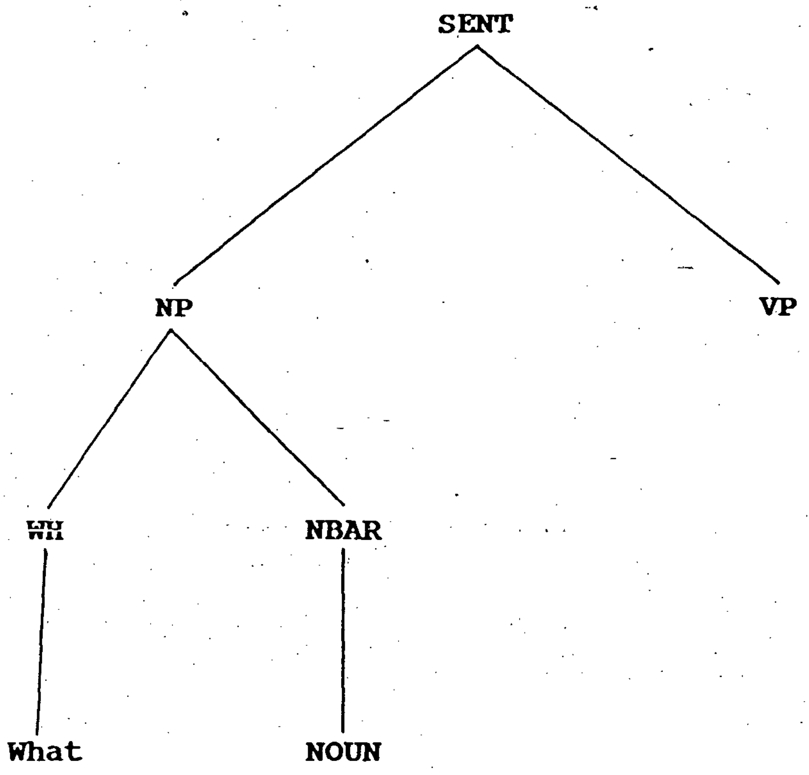
Now "NP" is replaced by first tree (see figure 5.1) of auxiliary tree set



"WH" is the pre_terminal and the first symbol in the input string *What* belongs to this word category. So, the word *What* is removed from the input string and attached to the pre_terminal in the above tree. The replacement is announced as *SUCCESSFUL*. The tree left is -



Now, "NBAR" is replaced by the first auxiliary tree with the root node "NBAR", i.e tree no. 4.



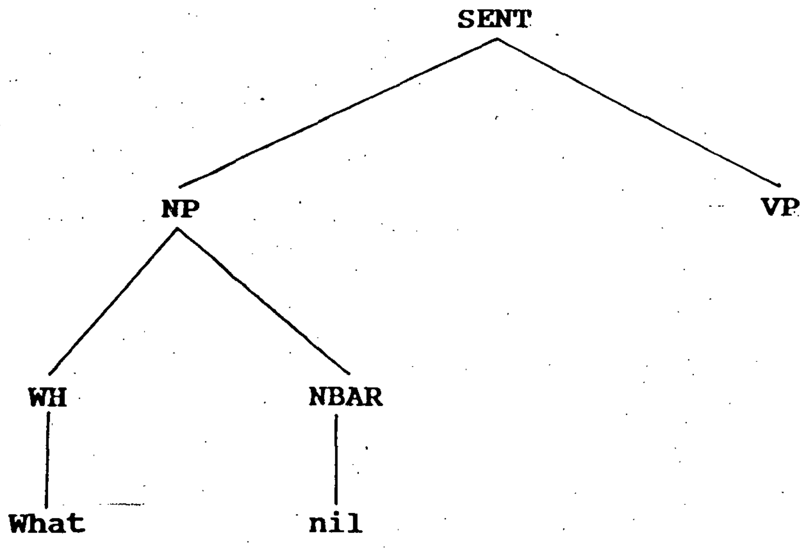
is the age of Gurjeet

remaining string

FAIL

but the next word in the input string, i.e. *"is"* does not belong to word category *"NOUN"*. Hence, the adjoining fails here. Other trees with root node NBAR are tried; this is called *backtracking.*

So, the next tree with root node NBAR is tree number 6 (see figure 5.1)



is the age of Gurjeet

remaining string

SUCCESS

nil is a terminal and can be attached to any non_terminal, this replacement is always true. Now the next non_terminal in the tree is VP when looked in in_order manner. "VP" can be replaced by tree number 8, i.e. the first tree with root node "VP" in the auxiliary tree set as shown in the figure (a) on the next page. Next symbol to be processed is the pre_terminal "VERB". The first terminal of the remaining input string is attached to the pre_terminal if it belongs to that pre_terminal category and SUCCESS is announced.

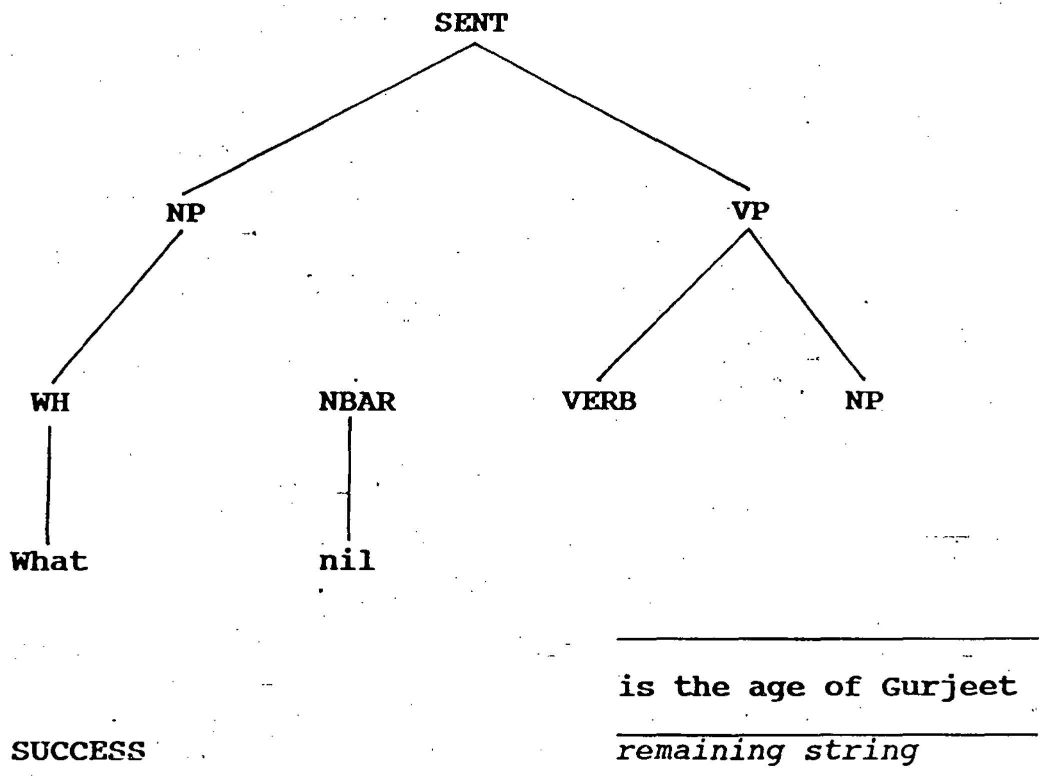


Figure (a)

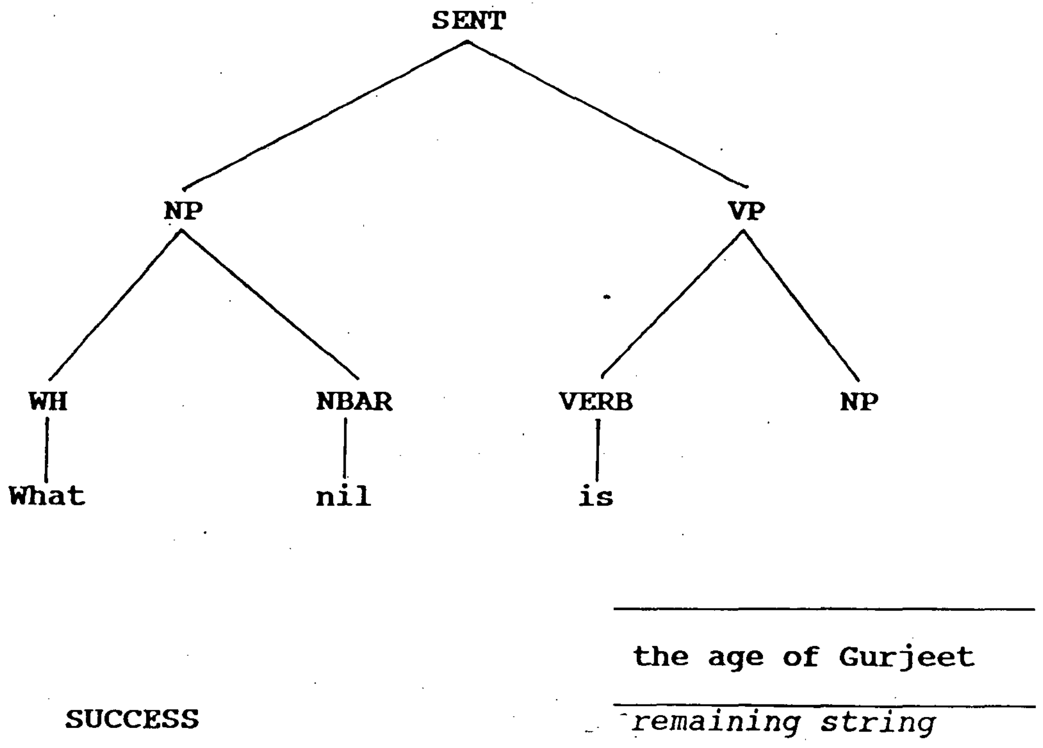
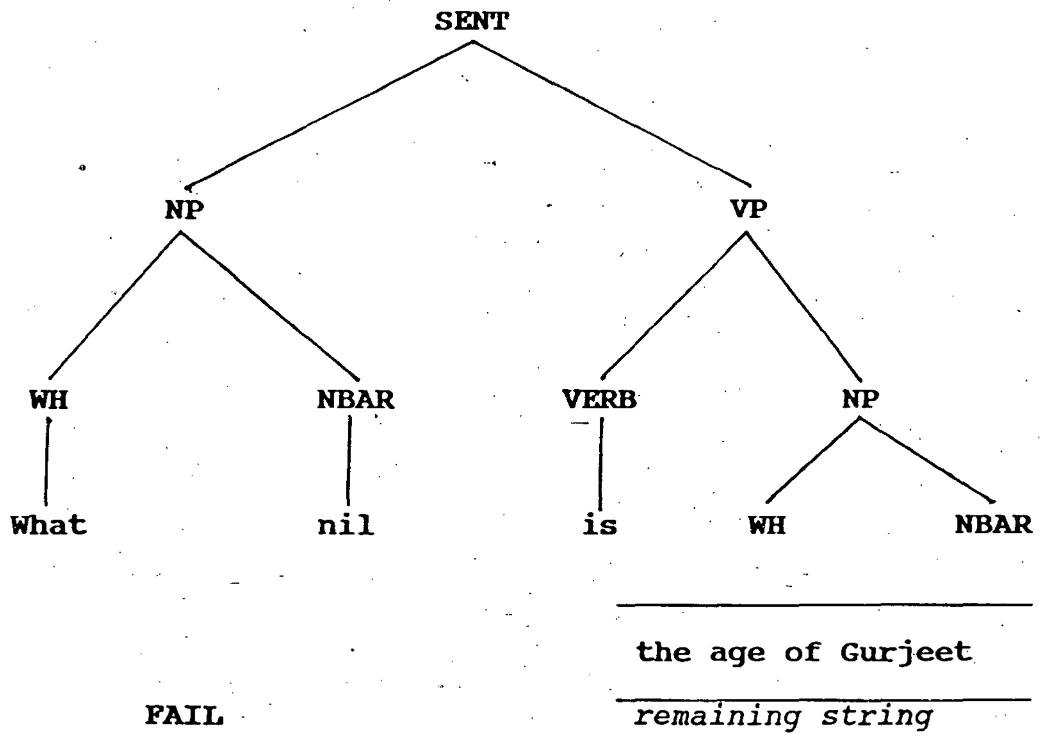
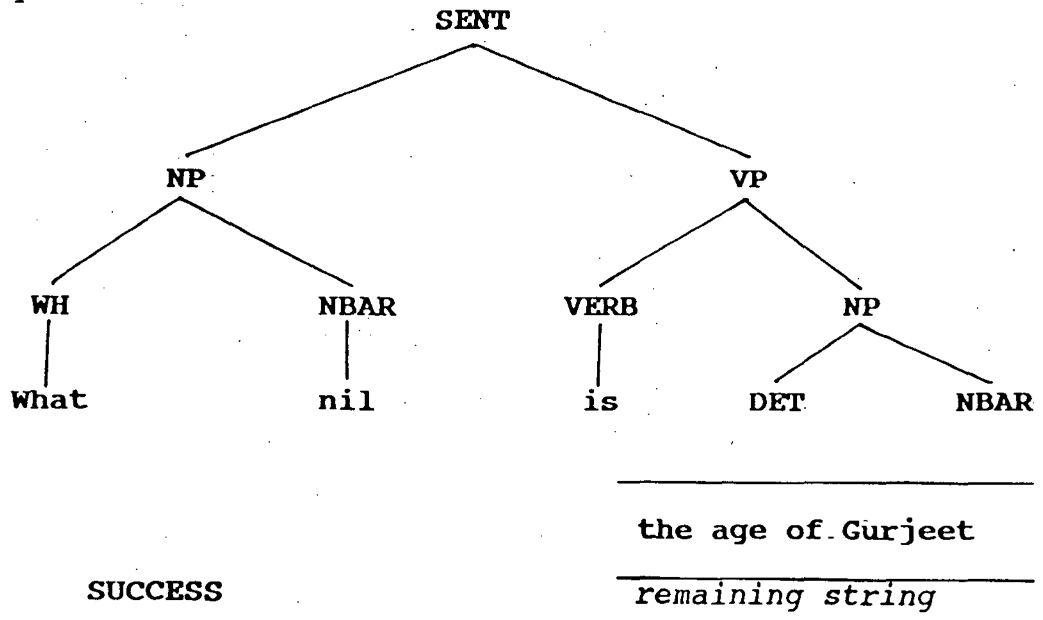


Figure (b)

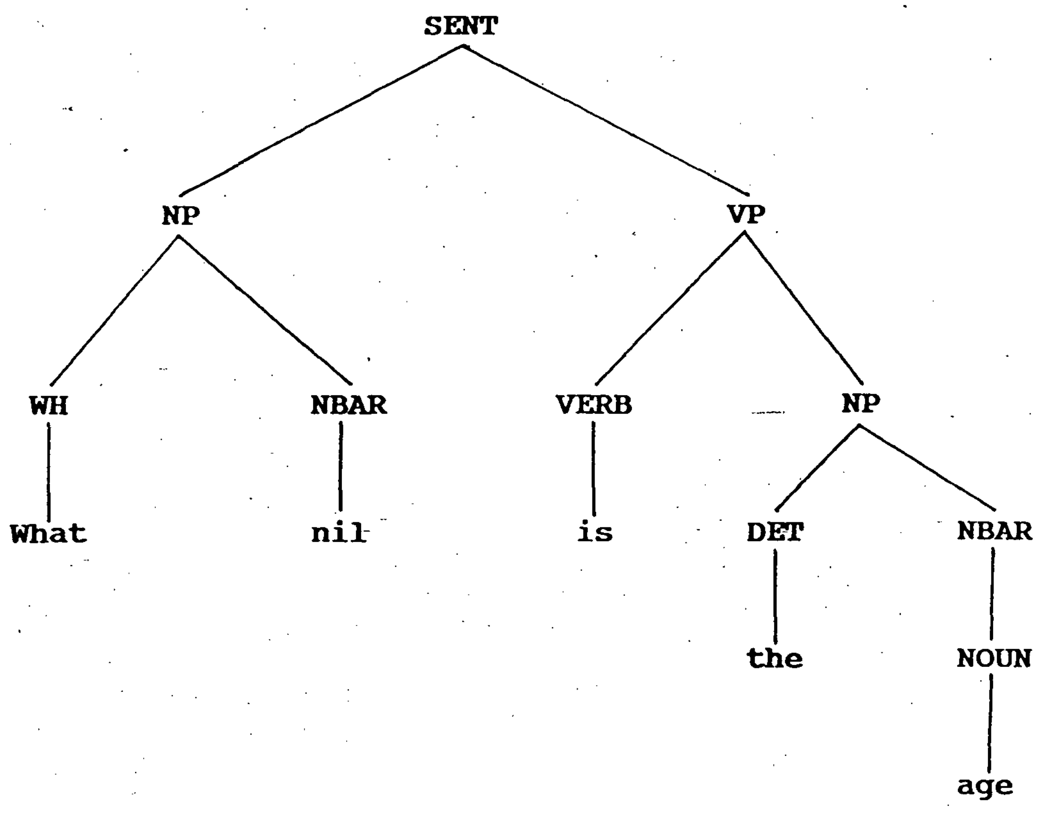
"NP" is replaced by the auxiliary tree number 1.



Since the next word in input string, i.e. *"the"* does not belong to "WH" category. Hence the replacement fails and parser has to backtrack and try other alternatives. "NP" is replaced by tree number 2 -



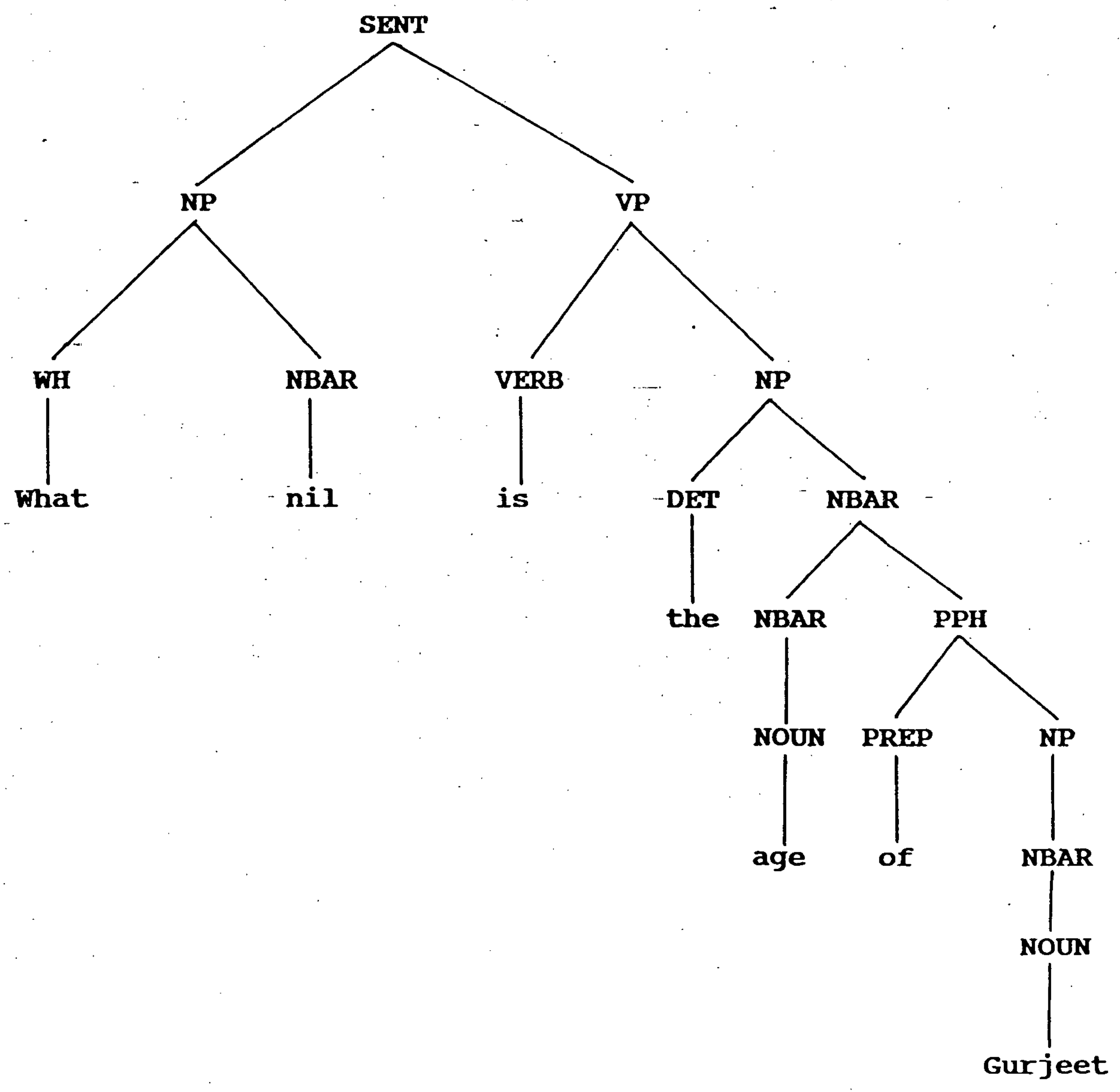
The terminal *"the"* is attached to the pre_terminal "DET". Next NBAR is replaced and "age" is attached to NOUN -



of Gurjeet

FAIL *remaining string*

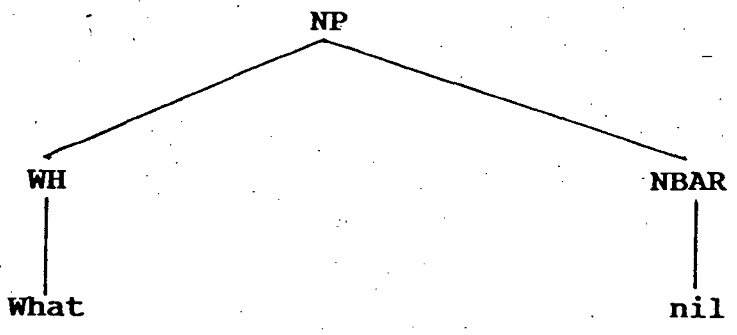
The adjoining done in the tree given above is announced "failed". Of course, the adjoining of tree number 2 for the node NBAR was correct, since the next terminal from the remaining input string, i.e. *"age"* belongs to the word category NOUN. But after doing this replacement the tree has got no non_terminal free and remaining input string is not empty at this point. Our parsing is successful only when all leaves are terminals and the input string is completely exhausted. Next, tree number 6 is tried -



SUCCESS

remaining string

The input string is exhausted and only terminals are at the leaves of the tree generated, hence parsing is successful. It is the diagrammatic representation of the parser output, actually the output of the parser is a list of terminals/non_terminals. This list is a post_order traversal of the parse tree. Whenever a sub_tree is completed, i.e. all its leaves are terminals only, and the recent adjoining is flagged as successful, then the post_order traversal of that sub_tree is stored in the list. Each non_terminal in the list is also having a numeral attached to it, indicating the number of children it has in the parse tree. For example, the sub_tree -



When "nil" is attached to NBAR, the sub_tree with NP as root node is completed, hence its post order traversal is stored in the list as - *what, WH, nil, NBAR1, NP2.*

So, the output of the parse tree for the query *"What is the age of Gurjeet" ?* is -

What, WH, nil, NBAR1, NP2, is, VERB, the, DET, age, NOUN, NBAR1, of, PREP, Gurjeet, NOUN, NBAR1, NP1, PPH2, NBAR2, NP2, VP2, SENT2

Auxiliary tree set can easily be extended. This makes it possible for the parser to handle more complicated queries.

CH. 6 SEMANTIC ANALYSIS

In this chapter, we will discuss the translation process of our system. The translation of parse tree to pseudo query and then its interpretation to QUEL query has been described in detail with examples. Algorithms have been given in pseudo code for the procedures used in the translation.

6.1 Procedures for the Translation Process :

The act of translating a parse tree to a pseudo query is a bottom_up operation. The leaves of the tree contain words and each interanl node contains a function which acts on the values returned by its offspring and returns a pseudo query. Pseudo queries are themselves functions and the lexicon entry for each object_word is a pseudo query which describes the role of that word in database access. The WH words act as signals in the translation process. When they appear before NOUN, it is assumed that the NOUN is a class of items and that the values of corresponding fields in the database relation are requested by the user. When a WH word appears in place of an object word, its location in relationship to the surrounding syntactic structures is used to determine what information is requested. WH words are represented in the lexicon as the name of a subroutine; the function of this subroutine is to copy the relation/field pairs in the ID into the PRT field.

The relationship between a sentence's subject and the object of its verb can be determined in part by the number of objects of the verb. Thus, it is reasonable to propose that the semantic portion of the verb will include a means of determining how many objects follow it. It would also be helpful if there was a way of establishing whether these objects fall into recognizable categories.

When the translator performs its bottom_up evaluation of the parse tree, it starts with the individual word (terminal node) definitions and moves up through progressively higher structures until the entire question has been translated. It is, therefore, very natural for the translator to build pseudo queries with embedded sub_queries.

The uppermost nodes in the parse tree process information that is synthesized from a combination of the lower nodes. The procedure which creates these combinations is called HAS_A function and it takes as parameters two pseudo queries.

The pseudo queries in the parameter list of HAS_A representing the modifying sub_structure (or constituents) is called the inferior parameter. The parameter representing the constituent being modified is called superior parameter (Sup).

The ID fields of pseudo queries have a specific relationship to each other. There can be many combinations of pseudo queries with different ID's. For combinational purposes, pairs of pseudo queries have been broken up into three classes _

(1) the two IDs have the same value.

(2) the relation in the ID of the inferior pseudo

query is in the FROM list of the superior pseudo query (inferior modifies the superior).

(3) all other possibilities.

If case(1) holds for the pair of pseudo queries then only the result of case(1) combination is returned. Likewise, if case(1) is not applicable for the pair, then result of case(2) combination is returned ; otherwise, the result of case(3) combination is returned. The algorithm of HAS_A function is given below.

```
Function HAS_A (sup,inf);
begin
    if sup.ID = inf.ID then
      /* case(1) */
    begin
        result.ID = sup.ID
        result.FROM = sup.FROM
        result.PRT = sup.PRT
        if sup.WHERE = nil then
                  result.WHERE = inf.WHERE
        else if inf.WHERE = nil then
                  result.WHERE = sup.WHERE
             else
                   result.WHERE = sup.WHERE AND
                                  inf.WHERE
    end      /* end of case(1) */
```

```
else   /* beginning of case(2) */
     if sup.ID.relation E sup.FROM and
        (inf.PRT <> nil or  inf.WHERE <> nil)
     then begin
               result.ID = sup.ID
               if sup.FROM = inf.FROM then
                  result.FROM = sup.FROM
               else result.FROM = sup.FROM AND
                                       inf.FROM
               if sup.WHERE = nil then
                  result.WHERE = inf.WHERE
               else if inf.WHERE = nil then
                        result.WHERE = sup.WHERE
               else
                    result.WHERE = sup.WHERE AND
                                        inf.WHERE
end   /* end of case(2) */
else  /* beginning of case(3) */
begin
     result.ID = sup.ID
     result.FROM = sup.FROM
     result.PRT = sup.PRT
     if sup.WHERE = nil then
        result.WHERE = inf.WHERE
     else if inf.WHERE = nil then
               result.WHERE = sup.WHERE
```

```
                 else
                       result.WHERE = sup.WHERE AND
                                      inf.WHERE
            end     /* end of case(3) */
            return (result)
        end;    /* end of function HAS_A */
```

HAS_A is a very general all purpose procedure. Any pair of pseudo queries can be combined using HAS_A. However, there are points in the translation process when functions other than mere combining must be performed on a pseudo query list. The INSTANCE procedure of our program is a generalization of HAS_A function and contains the code needed to correctly handle cases where a function name is passed to a higher node. The pseudo code for INSTANCE is given below -

```
Function INSTANCE (sup,inf);
/* 'sup' parameter is the left sublink of the node
and 'inf' parameter is the right one in the parse
tree */
begin
    if sup = nil then
         return (inf)
    if inf = nil then
         return (sup)
    if (sup is a function name) then
       if sup = 'cp_id_to_prt' then
```

```
            begin
                  inf.PRT = inf.ID
                  return (inf)
            end
            else
                return (sup.function, inf)
         else
            if (inf is a function name) then
               if inf = 'cp_id_to_prt' then
               begin
                  sup.PRT = sup.ID
                  return (sup)
                end
                else
                    return (inf.function, sup)
            else
                return (HAS_A (sup,inf))
        end /* end of function INSTANCE */
```

The use of INSTANCE will be illustrated in the examples of the next section.

6.2 Details of the Translation Process :

We have already seen (in Ch. 4) the definition for the NOUNS, VERBS, PREPOSITIONS, WH_word etc. for our sample DBF database. The complete lexicon given is in the Appendix A.

Let us examine each step of the translation for the sentence -

"What is the age of Gurjeet" ?

The lexicon used for the parsing and translation of this query is shown in figure 6.1. The parse tree is shown in figure 6.2.

WORD	CATEGORY	DEFINITION	
What	WH	function (cp_id_to_prt)	
is	VERB	function (instance)	
the	DET	nil	
age	NOUN	FROM	DBF
		ID	DBF.AGE
		PRT	nil
		WHERE	nil
of	PREP	function (instance)	
Gurjeet	NOUN	FROM	DBF
		ID	DBF.NAME
		PRT	nil
		WHERE	DBF.NAME = "Gurjeet"

Figure (6.1)

We will illustrate the translation process as a bottom_up sequence of transformation of the parse tree. Each interior node in the tree corresponds to a function and the first function evaluated are those corresponding to pre_terminal nodes : WH, NOUN, VERB etc. The functions associated with a pre_terminal node has the same name as the pre_terminal node and merely returns the definition of the word attached to it. When the function NBAR is called with only one parameter, it returns that parameter unchanged. When NBAR has more than one parameter, it returns a list of parameter functions and first parameter and second parameter as the parameter of this new function. The pseudo code for NBAR is -

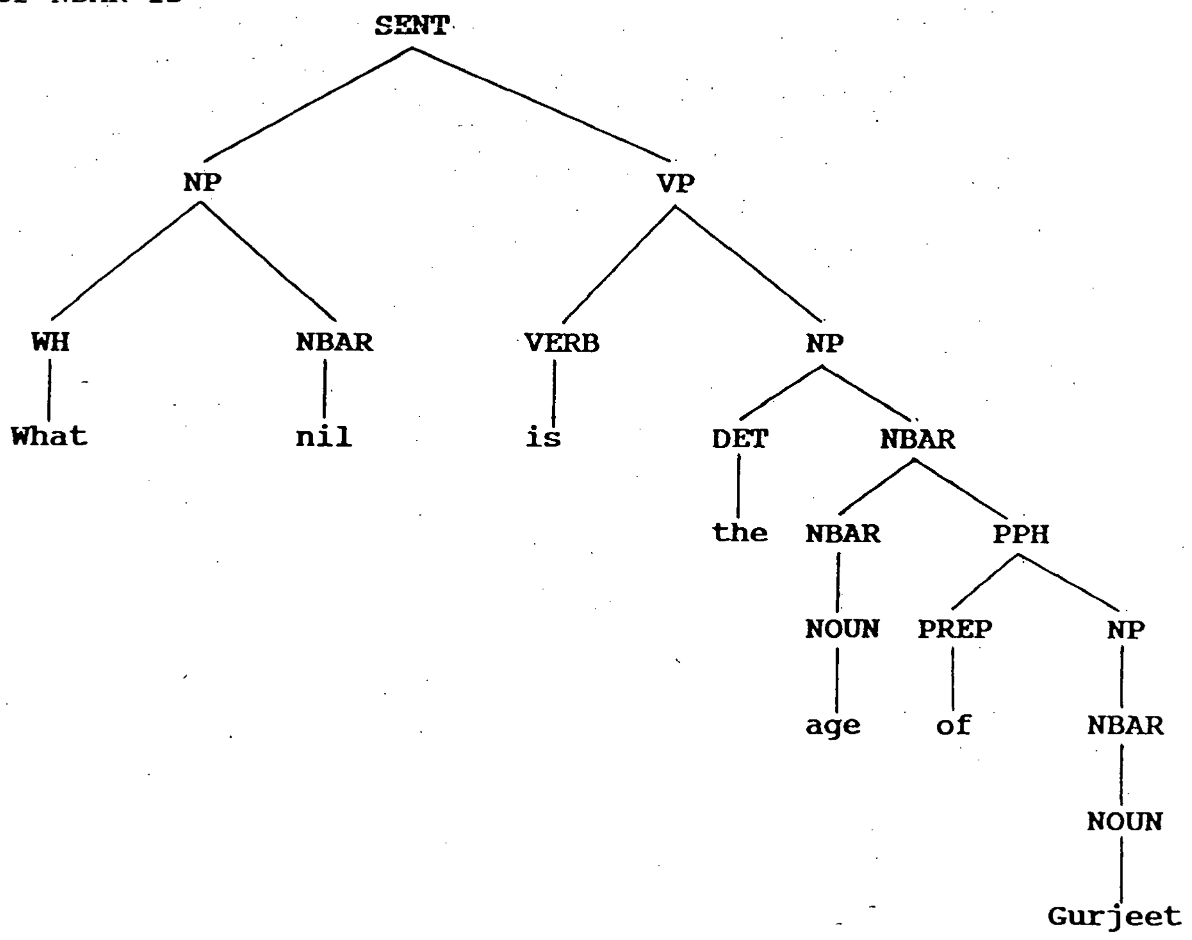


Figure (6.2)

```
Function NBAR (parameter #1, [parameter #2]);
begin
    if (there is only one parameter) then
        return (parameter #1)
    else
        return (parameter #2.function (parameter #1
                 ,parameter #2.para_list))
end;  /* end of function NBAR */
```

Thus, after evaluating the pre_terminal function and the NBAR, we have the tree shown in figure 6.3.

The function NP is given in pseudo code below. The parameter listed in square bracket is optional.

```
Function NP (parameter #1, [parameter#2])
begin
    if (there is only one parameter) then
      return (parameter #1)
    else
        return ( INSTANCE (para#1, para#1))
end  /* end of function NP */
```

The function PPH is called with two parameters representing a preposition and its object. The preposition will be represented by an appropriate function name. The code for PPH can be summarized as follows -

```
Function PPH (prep, object)
begin
    return (a list of prep and object)
end  /* end of function PPH */
```

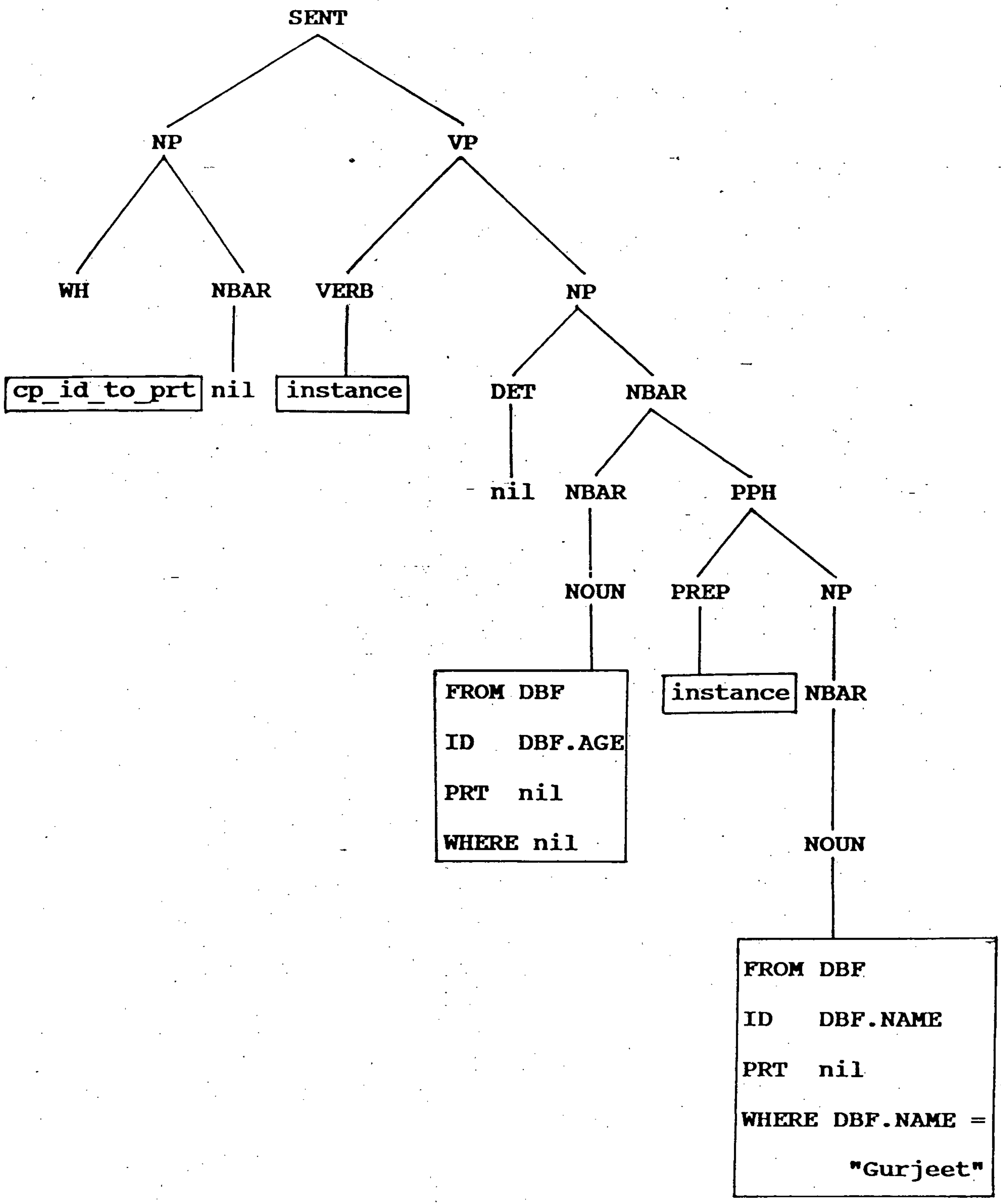

SENT
NP
VP
WH
NBAR
VERB
NP
cp_id_to_prt
nil
instance
DET
NBAR
nil
NBAR
PPH
NOUN
PREP
NP
FROM DBF
ID DBF.AGE
PRT nil
WHERE nil
instance
NBAR
NOUN
FROM DBF
ID DBF.NAME
PRT nil
WHERE DBF.NAME = "Gurjeet"

Figure (6.3)

After the PPH and NP has been applied, we have the t as shown in figure 6.4.

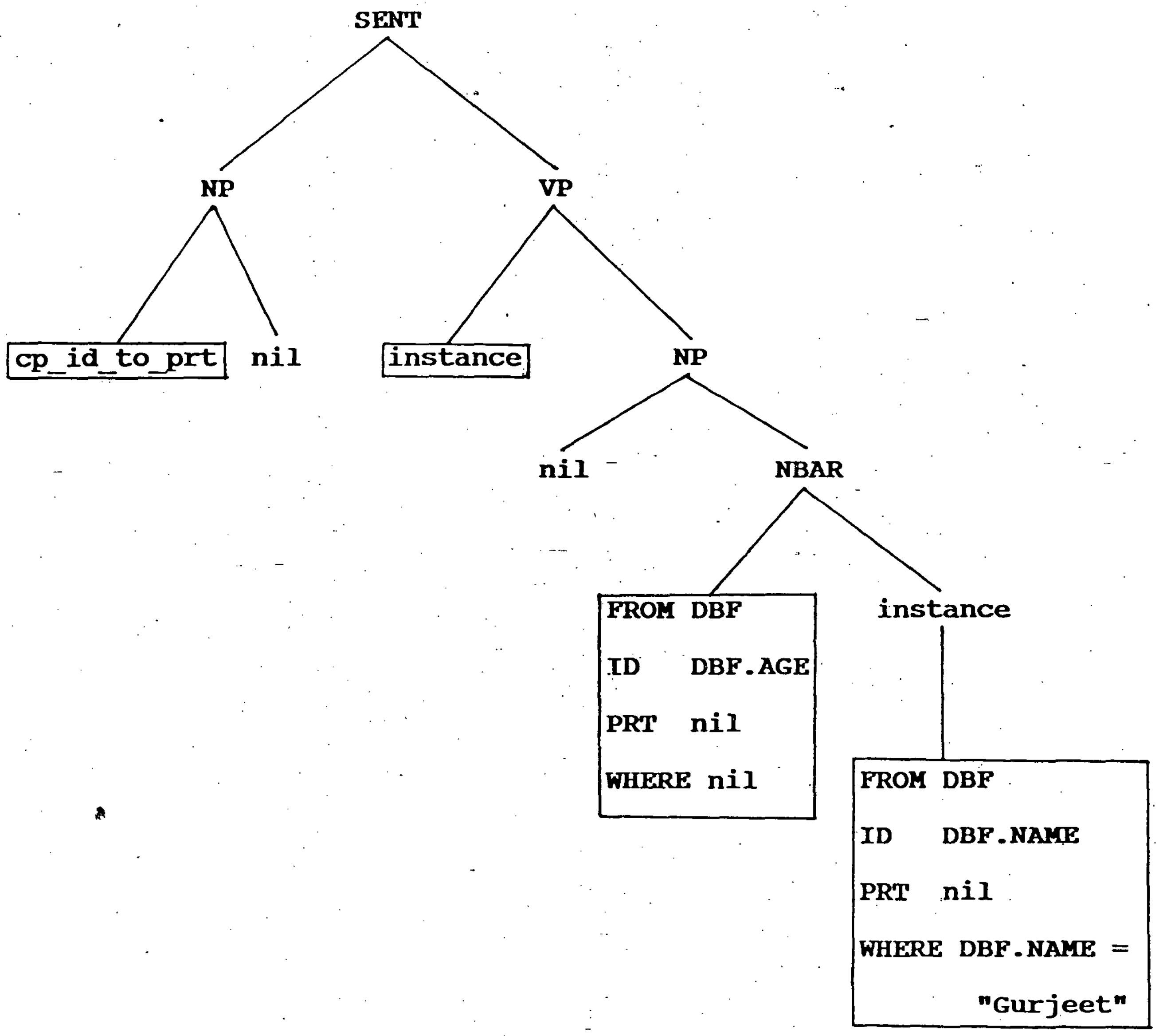


Figure (6.4)

When INSTANCE is called with only one parameter, it returns that parameter unchanged. Function NP also returns the non_nil parameter unchanged when one of them is 'nil'. So, the tree left is shown in figure 6.5.

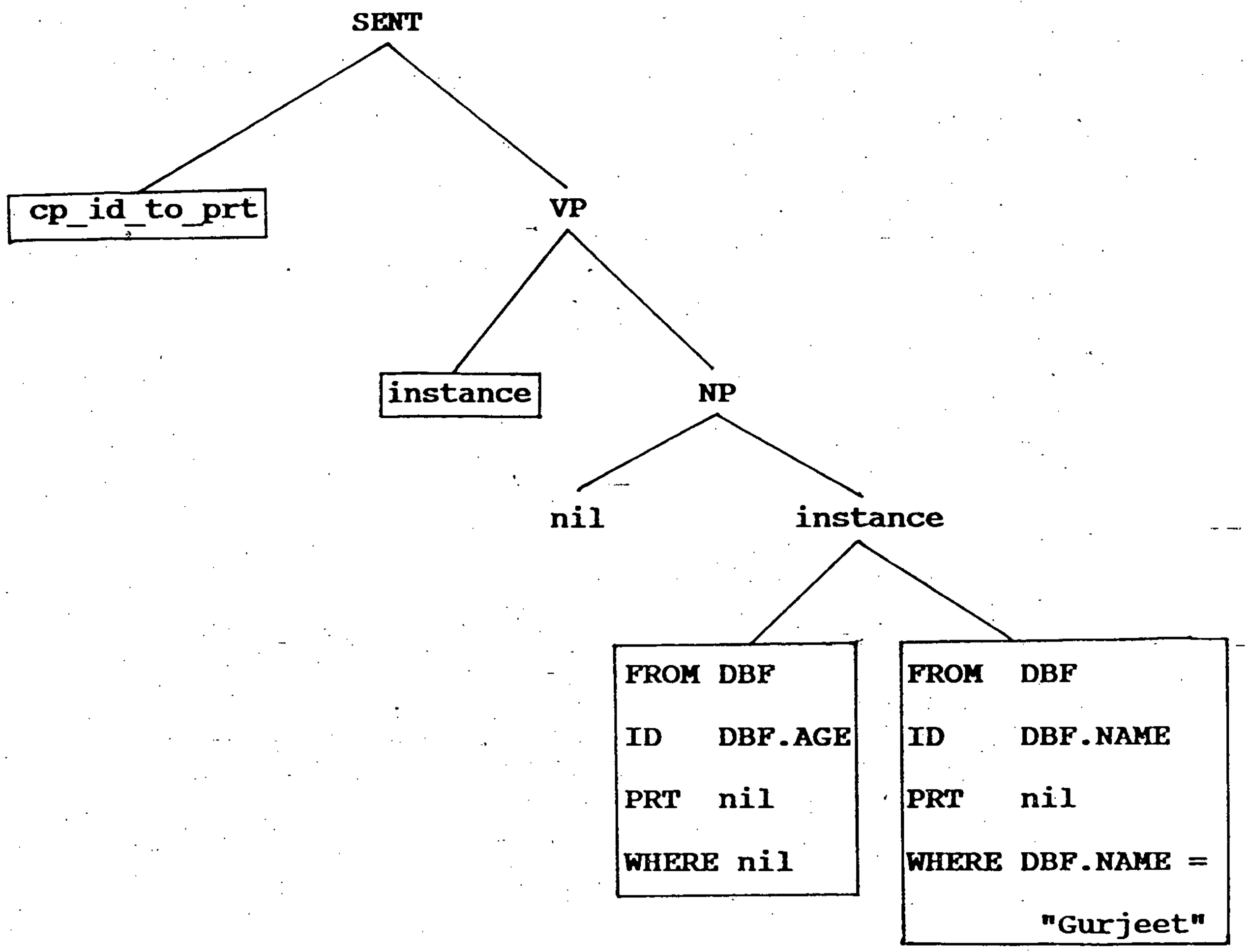


Figure (6.5)

The function INSTANCE has two parameters, shown in figure 6.5, since none of them is a function name. So, the INSTANCE function, in turn, calls the HAS_A function. Case (2) of HAS_A function is applicable, since the ID fields of its parameters are not equal. After evaluating this node, we are left with the tree shown in figure 6.6.

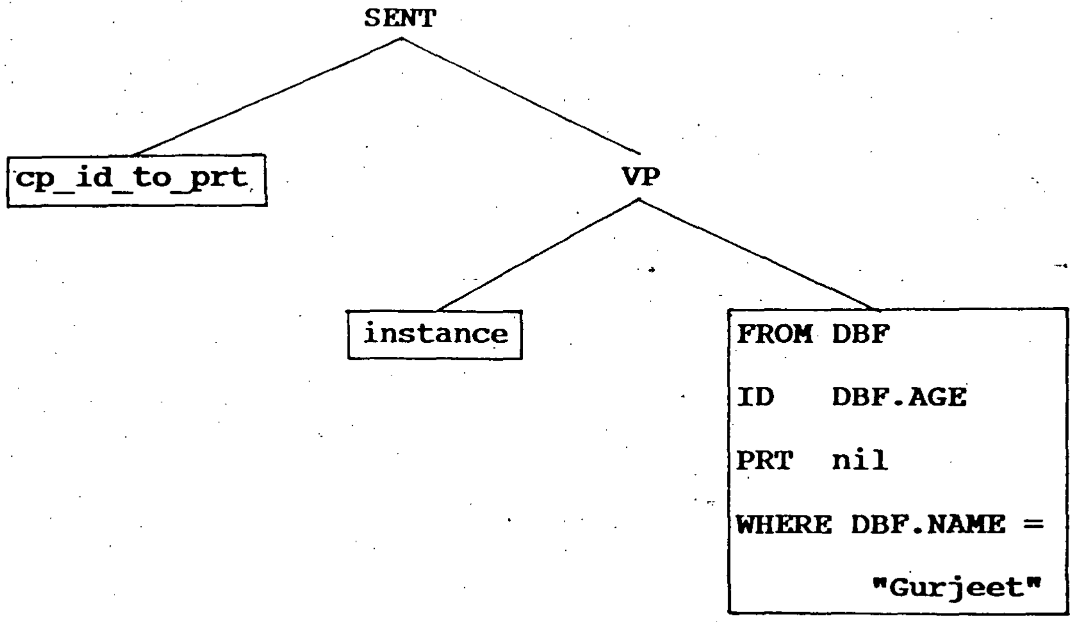


Figure (6.6)

Next, the node VP is evaluated. It has two parameters in the tree of fig. 6.6. The function of VP is very simple and can be summarized as follows -

```
Function VP (parameter #1, parameter #2)
begin
    return (a list of para #1 and para #2)
end  /* end of function VP */
```

Tree, left after solving the VP node, is shown in figure 6.7.

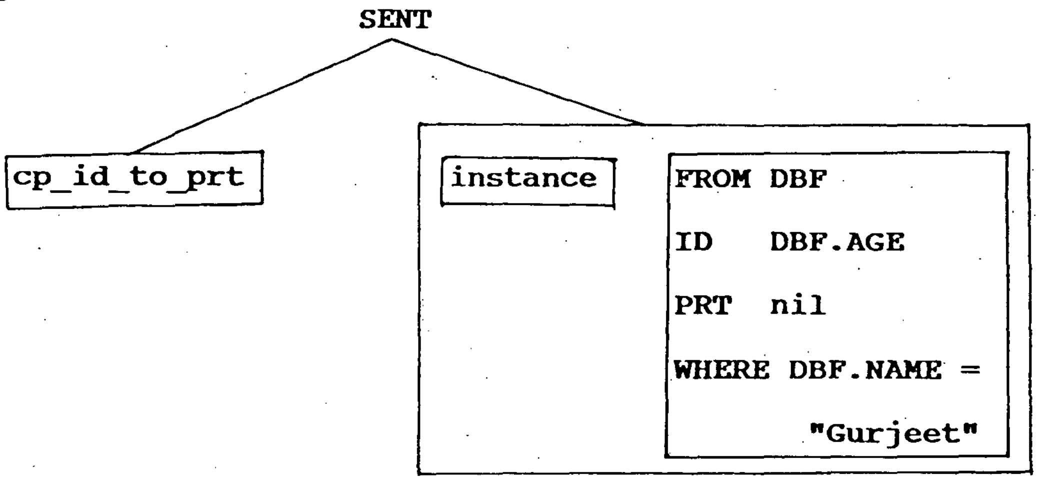


Figure (6.7)

Finally, the function SENT has two parameters corresponding to the noun phrase and the verb phrase. The first element of the verb phrase is a function. So, the notation for a function and a parameter list given with the definition of INSTANCE may be used. Then SENT may be represented in pseudo code as follows -

```
Function SENT (np, vp)
begin
    return (vp.function (np, vp.para_list))
end   /* end of function SENT */
```

For the tree in figure 6.7, VP.function is INSTANCE. So, the result of evaluating SENT is the result of evaluating the tree in figure 6.8.

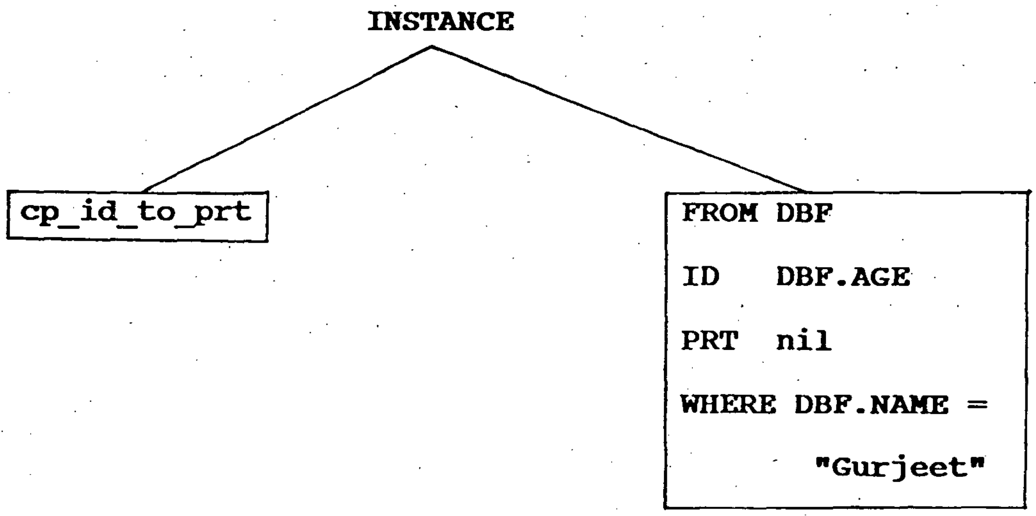


Figure (6.8)

The function cp_id_to_prt copies the ID field of its parameter into the PRT field. So, the result of evaluating this function will give the pseudo query -

```
FROM     DBF
ID       DBF.AGE
PRT      DBF.AGE
WHERE    DBF.NAME = "Gurjeet"
```

Figure (6.9)

The structure of pseudo query is so chosen that it can directly be mapped to the QUEL query used by INGRES. The FROM field of pseudo code is RANGE of QUEL. The PRT field of pseudo code is RETRIEVE of QUEL and WHERE of pseudo code is same as the WHERE of QUEL. Hence, the QUEL query thus obtained is -

```
RANGE     DBF
RETRIEVE  DBF.AGE
WHERE     DBF.NAME =
            "Gurjeet"
```

for our English query.

Let us consider another English query which has nested structure -

What is the name of person whose age is 23 ?

The parser output for this natural language query is -

What, WH, nil, NBAR1, NP2, is, VERB, the, DET, name, NOUN, NBAR1, of, PREP, person, NOUN, NBAR1, whose, WH, age, NOUN, NBAR1, NP1, is, VERB, 23, NOUN, NBAR1, NP1, VP2, SBAR3, NP2, PPH2, NBAR2, NP2, VP2, SENT2.

This is the post_order traversal of the parse tree and number attached to each internal node is the number of siblings of that node. We are taking a part of the parse tree whose root node is SBAR. We have seen the function of all internal nodes except *SBAR* in the previous example. So, let us carefully examine the evaluation of SBAR structure in the parse tree of the above natural language query. The sub_tree is given in figure 6.10.

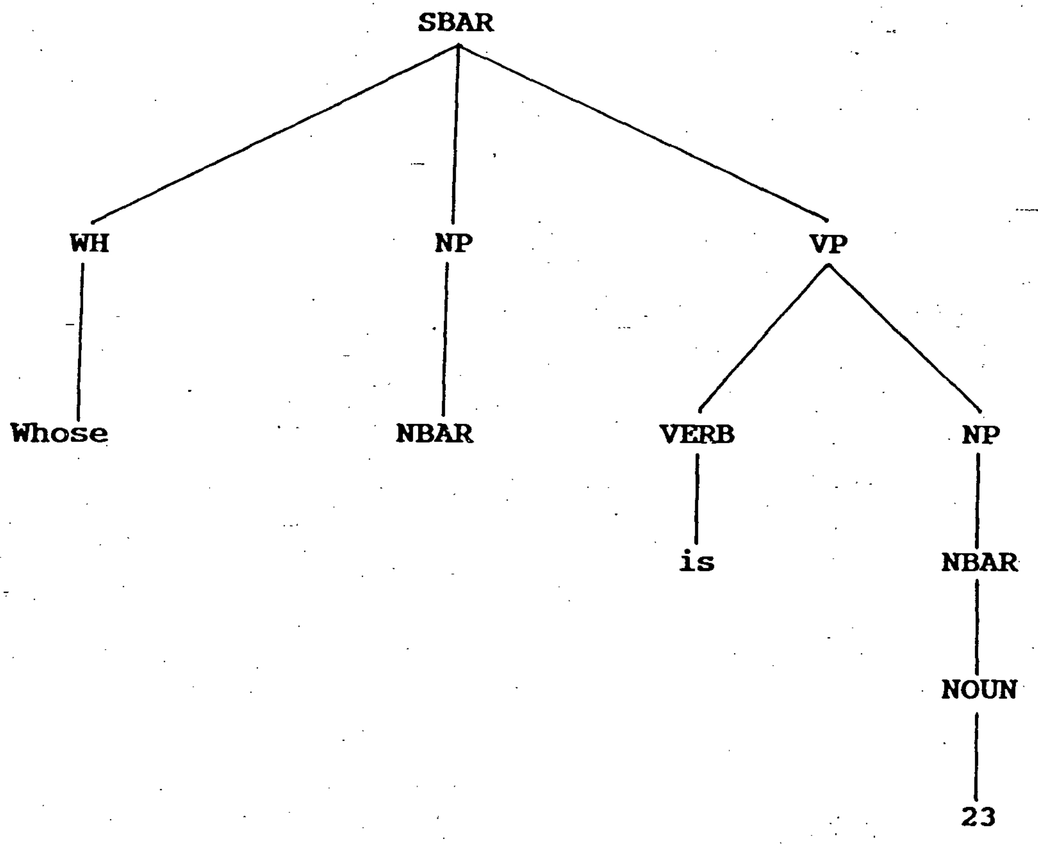


Figure (6.10)

Again, the functions at pre_terminal nodes return the definitions of the words associated with them and the functions NBAR and NP called with only one parameter return that parameter unchanged. After the evaluation of these functions, we have the tree shown in figure 6.11.

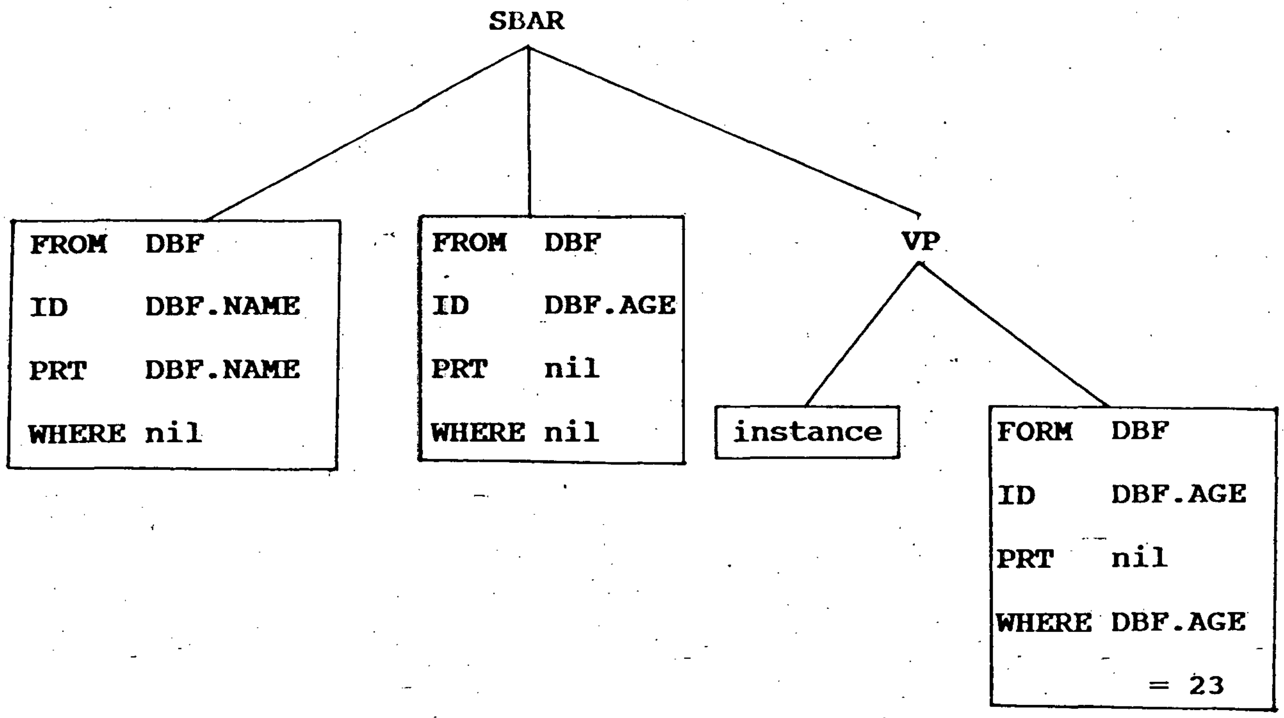


Figure (6.11)

Referring to the definition of VP, given previously, we see that evaluation of VP gives the tree shown in figure 6.12.

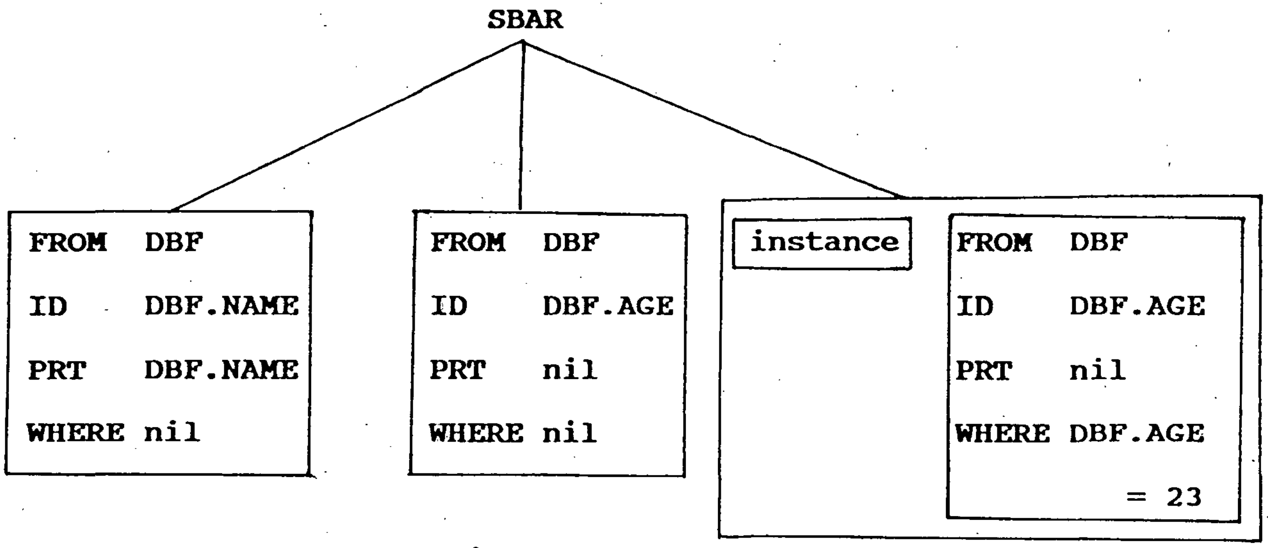


Figure (6.12)

The function of SBAR may be used in different environments from the one we see in this example. When SBAR is called with tree parameters, however, the situation is always one in which the left offspring is a WH word which denotes possession. When it is called with only two parameters, the first parameter is a pronoun, whose antecedent is a sibling of the SBAR and parameter#2 will be linked to this antecedent later in the translation process. Thus, SBAR may be given in pseudo code as follows -

```
Function SBAR (parameter#1, parameter#2, [parameter#3])
begin
      if there are tree parameter then
         return (para#3.function (para#1, para#2))
      else
         return (parameter#2)
end;
```

So, evaluation of SBAR means evaluation of INSTANCE in our example, as shown in figure 6.13.

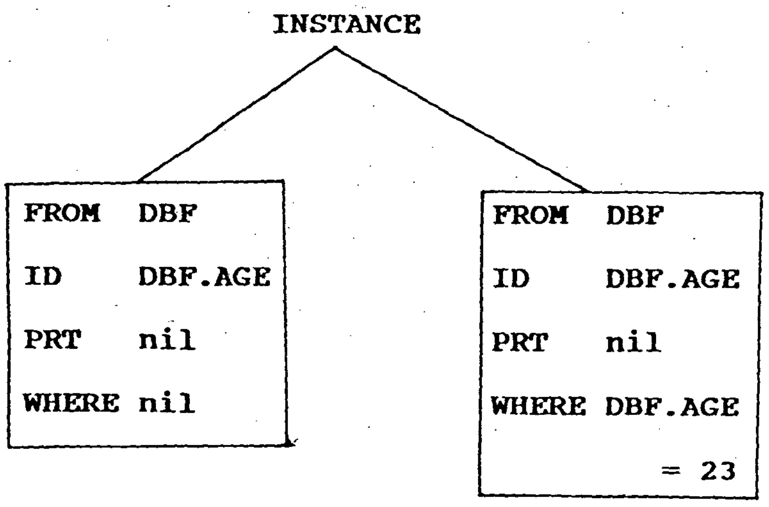


figure (6.13)

After evaluating INSTANCE the pseudo query obtained for the sub_tree is -

```
FROM  DBF
ID    DBF.AGE
PRT   nil
WHERE DBF.AGE
          = 23
```

This is equivalent to the definition of the noun "23". The evaluation of rest of the tree is same as given in the previous example. The pseudo query obtained finally is -

```
FROM  DBF
ID    DBF.NAME
PRT   DBF.NAME
WHERE DBF.AGE
          = 23
```

This pseudo query is then converted to equivalent QUEL query. Conversion is simple because the function of FROM, PRT and WHERE of pseudo query fields are the same as the function of RANGE, RETRIEVE and WHERE fields respectively of the QUEL query. Thus, the QUEL query obtained is -

```
RANGE      DBF
RETRIEVE   DBF.NAME
WHERE      DBF.AGE
               = 23
```

CONCLUSION

Before concluding the dissertation, we discuss the class of questions accepted by our system, the linguistic phenomena that are being tackled and, of course, the limitations of the system.

In this system, only those questions which start with "Wh", i. e. the questions starting with a word belonging to the "Wh" category such as Who, What, Which, When, Where, etc., are accepted. The questions beginning with a "Why" are not inclusive in the list as such type of questions lead to a problem of reasoning which, clearly, is beyond the scope of our system.

The parser that we have implemented is capable of including any unknown word in the parse tree. The parser treats an unknown word as a "noun", allowing it to take place in the parse tree but identifying it with a special pre-terminal symbol. Numbers are treated in the same way, but identifying it with a different pre-terminal symbol to distinguish them from string values. If the parser is not able to create a parse tree, it backtracks and asks the user for the category of the unknown word. Its definition will be asked during the translation phase. Still, if no exact parse tree is found, the user is signalled and the translator is not invoked.

Of the auxiliary verbs, modals such as can, could, will, would, shall, must, may, might, etc. are not included. Questions involving comparators such as "more than", "less than", "equal to", etc. are also not allowed. Besides, some sentences which are not gramatically correct may also be able to get themselves parsed which is not desireable.

This system has been designed to handle the nested queries also very efficiently. But it does not support connectors like and, or, etc.

This system can easily be adapted to a given user database, which makes the system "transportable". The only change required is in the part of the lexicon. The core lexicon remains the same. Definitions and the categories, if required, of the word according to their probable use in the query have to be changed by the database engineer in the data specific lexicon. Rest of the system remains unchanged.

The use of the intermediate pseudo-query makes the system portable; conversion for use with a different query language is accomplished by providing routines which convert the pseudo-query into a query in a new language.

In an effort to improve the TAG approach, the parser will have to be redesigned to allow the use of real links. Such an improvement would allow accurate translations of sentences which are presently untranslatable.

Another improvement can be made by altering the structure of the pseudo-query. Presently, a pseudo-query is a hierarchical structure, consisting of a primary query with embedded sub-queries. This approach is adequate for translating a wide variety of sentences. However, consider the sentence "What boy, whose dog's nose is broken, has a bicycle ?". The antecedent of "whose" is "boy". When the NP dominating "boy" and the NBAR (Whose dog's nose is broken) is processed, the resulting pseudo-query must incorporate the fact that "boy" possesses the "dog". However, because the translation of parse trees is processed in a bottom-up fashion. The primary query in the pseudo-query resulting in the pseudo-query resulting from translating the SBAR would represent the "nose". This represents a problem as it will seem to the translator that it is the boy whose nose is broken.

If the pseudo-queries were restructured to be directed graphs rather than hierarchical structures, this problem would be solved. When the phrase "whose dog's nose" is translated, the presence of "whose" would cause a special marker to be placed on the processor of "dog", and then translation would continue as previously described. Later, the SBAR function would look for this special marker. After finding it the SBAR function would direct the next higher level of translation to use the processor of "dog" as the antecedent of the sibling node of SBAR. Thus, "boy" would be established as the owner of "dog".

The TAGs have many unique properties such as links and local constraints which make them useful in processing NL. These properties can be used to devise a wide range of applications including commercial packages and research tools for linguists.

APPENDIX 1

```
/*****************************************************************/
/*     Database file lexicon.pro.                                */
/*     This file should be in drive <B> of your system.          */
/*****************************************************************/

predicates
          lexicon(s,s,either)

clauses

/*****************************************************************/
/*  This part of lexicon remain same, whatever may be the        */
/*  domain of your database.                                     */
/*****************************************************************/

lexicon(a,det,function(m)).
lexicon(an,det,function(m)).
lexicon(the,det,function(m)).
lexicon(in,prep,function(instance)).
lexicon(to,prep,function(instance)).
lexicon(is,verb,function(instance)).
lexicon(of,prep,function(instance)).
lexicon(what,wh,function(cp_id_to_prt)).
lexicon(who,wh,definition(dbf,"dbf.name","dbf.name",nil)).
lexicon(whose,wh,definition(dbf,"dbf.name","dbf.name",nil)).

/*****************************************************************/
/*  This is data specific part of lexicon and may have to be     */
/*  changed according to the domain of the database.             */
/*****************************************************************/

lexicon(got,verb,function(instance)).
lexicon(lives,verb,function(instance)).
lexicon(belongs,verb,function(instance)).
lexicon(gurjeet,noun,definition(dbf,"dbf.name",nil,
                                "name = gurjeet")).
lexicon(age,noun,definition(dbf,"dbf.age",nil,nil)).
lexicon(indore,noun,definition(dbf,"dbf.address",nil,
                                "address = indore")).
lexicon(name,noun,definition(dbf,"dbf.name",nil,nil)).
lexicon("m.tech.",noun,definition(dbf,"dbf.class",nil,
                                  "class = m.tech.")).
lexicon(person,noun,definition(dbf,"dbf.name",nil,nil)).
lexicon(class,noun,definition(dbf,"dbf.class",nil,nil)).
lexicon(grade,noun,definition(dbf,"dbf.grade",nil,nil)).
lexicon("152 kaveri",noun,definition(dbf,"dbf.address",nil,
                                  "address = 152 kaveri")).
lexicon("23",noun,definition(dbf,"dbf.age",nil,"age = 23")).
lexicon(address,noun,definition(dbf,"dbf.address",nil,nil)).
lexicon("A",noun,definition(dbf,"dbf.grade",nil,"grade = a")).
lexicon(mukul,noun,definition(dbf,"dbf.name",nil,"name = mukul")).
```

APPENDIX 2.

AN IMPLIMENTATION OF NATURAL LANGUAGE INTERFACING TO INGERS.

WRITTEN IN

Turbo PROLOG version 2.0

BY

Gurjeet Singh Khanuja

```
code = 3000

/* set the stack size to 3000 using setup of main menu */

domains

        s      = symbol
        ls     = s*
        either = word(s);
                 function(s);
                 definition(s,s,s,s)

database

        node(either)

        /* node database is used as stack during the translation
           phase. If translation is successful then it will cont_
           ain only one data element. That data element is the
           pseudo query equivalent of the question asked.
         */

        temp_store(either)
        unknown_word(either)

        /* A word is said to be unknown if it is not found in the
           lexicon of the system i.e lexicon.pro file. These unk_
           nown words found in the natural language query are st_
           ored in unknown_word database. temp_store database is
           used to handle these unknown words. We will store the
           unknown words, their category, and thier definition in
           this database.
                        System will refer this database whenever
           it fails to find a word in the lexicon.
        */
```

```
include "b:lexicon.pro"

/* The lexicon is being stored in a seperate file <lexicon.pro>
   which can be changed or updated time to time by the database
   engineer. The lexicon.pro file should be in the drive [B:]
   of your computer system.
*/


/******************** P R E D I C A T E S **********************/

predicates

          go
          read_sentence(ls)
          reverse(ls,ls,ls)
          reverse_lst(ls,ls)

      /* PREDICATES USED FOR  PARSING A QUERY */

          parse_sentence(ls,ls,ls)
          nounphrase(ls,ls,ls,ls)
          verbphrase(ls,ls,ls,ls)
          adjective(ls,ls,ls,ls)
          nbar(ls,ls,ls,ls)
          sbar(ls,ls,ls,ls)
          parser_output(ls)
          pph(ls,ls,ls,ls)
          refer(s,s)

      /* PREDICATES USED FOR TRANSLATION */

          find_def(s,either)
          empty_temp_store
          entered(s,s,either)
          definition_or_function(s,s)
          enter_def_or_fun(char,s,s)
          translate(ls)
          check(s)
          category
          nbar1
          np1
          pph2
          nbar2
          np2(either,either)
          vp2
          sent2
          inst
          instance(either,either)
          function_id_prt(either)
          has_a(either,either)
          cases(s,s,s,s,s,s,s,s)
          part_of_case2(s,s,s,s,s,s,s,s)
          part_of_case21(s,s,s,s,s,s,s,s)
          part_of_case1(s,s,s,s,s,s,s,s)
          union(s,s,s)
          write_the_result

      goal
          go.

/************************ C L A U S E S *************************/

clauses

        go :-
            read_sentence([]).


        /* read_sentence, reads the sentence inputted in a list
           L until the question mark <?> appears in the input
           sequence.
        */

        read_sentence(L) :-
                          write("Enter -> "),
                          readln(Word),
                          Word <> "?",!,
                          read_sentence([Word | L]).


        /* Since the list works in LIFO fashion therefore it is
           first reversed before sending it to the parser.
        */

        read_sentence(L) :-
                          !,
                          reverse_lst(L,LR),

                          /* parser is called here */

                          parse_sentence(LR,[],TL1),

                          /* TL1 is the parser output, it is post
                             order traversal of a tree called
                             parse tree. It is first reversed
                             before sending to the translator.
                          */

                          reverse_lst(TL1,TL2),
                          makewindow(1,7,7,"",0,0,25,80),
                          TL3 = TL2,
                          write("Parsed tree ........."),nl,
                    /* parser_output, writes the output of
                       the parser on the crt.
                    */

                    parser_output(TL2),nl,

                    /* 'translate' is the first rule of the
                        translator phase.
                    */

                    translate(TL3).


/* reverse_lst, reverse the order of contents of list L.
   The reversed list is in LR. Note the content and order
   of elements in list L remain unaltered.
*/

reverse_lst(L,LR) :-
                    reverse(L,[],LR).

reverse([],L,L).

reverse([H|T],L1,L2) :-
                      reverse(T,[H|L1],L2).


/* The following rules writes the output of the parser */

parser_output([]) :-
                    readchar(_).

parser_output([H|T]) :-
                      write(H,"  "),
                      parser_output(T).


/* The following rule is called after the translation
   phase. At the end of translation process there should
   be only one data element in the 'node' database. This
   data element is the QUEL query which must be euivalent
   to the natural language query.
*/

write_the_result :-
                   retract(node(definition(M,_,O,P))),!,
                   write("FROM     : ",M),nl,
                   write("RETRIVE : ",O),nl,
                   write("WHERE    : ",P),nl.
```

```
/************************ P A R S E R **************************/


        /* There are four atoms in most of the parser rule, e.g.
           nounphrase(ls,ls,ls,ls). All the atoms are of type
           list <ls> <list of symbols>. The first atom represents
           the remaining list, i.e. input string yet to be procc_
           ess. The second atom denotes, the remaining list which
           is to be returned after the completion of that rule.
           The third and fourth atom represent the parser output.
           The third one is the parser output passed to this rule
           by last rule executed, means the current status of the
           parser output. The fourth, will be the parser output
           after the completion of this rule.
        */


        /* Following is the Initial tree of our Tree Adjoining
           Grammar.
        */

        parse_sentence(L,TL,TL1) :-
                                  nounphrase(L,RL,TL,TL2),
                                  verbphrase(RL,RL1,TL2,TL3),
                                  TL1 = [sent2|TL3],
                                  RL1 = [].


        /* Following rule represent the first auxiliary tree of
           auxiliary tree set
        */

        nounphrase([H|T],RL,TL1,TL2) :-
                                      refer(H,C),
                                      C = wh,
                                      TL3 = [H|TL1],
                                      TL4 = [cat|TL3],
                                      nbar(T,RL1,TL4,TL5),
                                      TL2 = [np2|TL5],

                                      /* 'np2'  is stored  in the
                                         output   of  the  parser
                                         because  this  sub  tree
                                         with node nounphrase has
                                         got two siblings.
                                      */

                                      RL = RL1.
```

```
/* It is the second auxiliary tree */

nounphrase([H|T],RL,TL1,TL2) :-
                              refer(H,C),
                              C = det,
                              TL3 = [H|TL1],
                              TL4 = [det|TL3],
                              nbar(T,RL1,TL4,TL5),
                              TL2 = [np2|TL5],

                              /* This sub tree also have
                                 two siblings, one is de_
                                 terminer and other is
                                 nbar.
                              */

                              RL = RL1.


/* This is the third auxiliary tree */

nounphrase(L,RL,TL1,TL2) :-
                          nbar(L,RL1,TL1,TL3),
                          sbar(RL1,RL2,TL3,TL4),
                          TL2 = [np2 | TL4],
                          RL = RL2.


/* Fourth auxiliary tree */

nounphrase(L,RL,TL1,TL2) :-
                          nbar(L,RL,TL1,TL3),
                          TL2 = [np1|TL3].


/* Following is the part of third auxiliary tree */

sbar([H|T],RL,TL1,TL2) :-
                        refer(H,C),
                        C = wh,
                        TL3 = [H | TL1],
                        TL4 = [cat | TL3],
                        nounphrase(T,RL1,TL4,TL5),
                        verbphrase(RL1,RL2,TL5,TL6),
                        RL = RL2,
                        TL2 = [sbar3 | TL6].

                        /* This sub_tree has three sibli_
                           gs.
                        */
```

```
/* This is the fifth auxiliary tree and first of those
   auxiliary tree set which have 'nbar' as root node.
*/

nbar([H|T],RL,TL1,TL2) :-
                          refer(H,C),
                          C = noun,
                          TL3 = [H|TL1],
                          TL4 = [cat|TL3],
                          RL = T,
                          TL2 = [nbar1|TL4].


/* Sixth auxiliary tree */

nbar(L,RL,TL1,TL2) :-
                      TL3 = [nil|TL1],
                      TL2 = [nbar1|TL3],
                      RL = L.


/* Seventh auxiliary tree */

nbar(L,RL,TL1,TL2) :-
                      nbar(L,RL1,TL1,TL3),!,
                      pph (RL1,RL2,TL3,TL4),
                      TL2 = [nbar2|TL4],
                      RL = RL2.


/* This is the part of seventh auxiliary tree */

pph([H|T],RL,TL1,TL2) :-
                         refer(H,C),
                         C = prep,
                         TL3 = [H|TL1],
                         TL4 = [cat|TL3],
                         nounphrase(T,RL1,TL4,TL5),
                         TL2 = [pph2|TL5],
                         RL = RL1.


/* Eight auxiliary tree */

verbphrase([H|T],RL,TL1,TL2) :-
                                refer(H,C),
                                C = verb,
                                TL3 = [H|TL1],
                                TL4 = [cat|TL3],
                                nounphrase(T,RL1,TL4,TL5),
                                TL2 = [vp2|TL5],
                                RL = RL1.
```

```
/* Nineth auxiliary tree */

verbphrase([H|T],RL,TL1,TL2) :-
                              refer(H,C),
                              C = verb,
                              TL3 = [H|TL1],
                              TL4 = [cat|TL3],
                              adjective(T,RL1,TL4,TL5),
                              TL2 = [vp2|TL5],
                              RL = RL1.


 /* Tenth auxiliary tree */

adjective([H|T],RL,TL1,TL2) :-
                             refer(H,C),
                             C = adjective,
                             TL3 = [H|TL1],
                             TL4 = [cat|TL3],
                             adjective(T,RL1,TL4,TL5),
                             TL2 = [adj2|TL5],
                             RL = RL1.


/* Eleventh auxiliary tree */

adjective([H|T],RL,TL1,TL2) :-
                             refer(H,C),
                             C = adjective,
                             TL3 = [H|TL1],
                             TL4 = [cat|TL3],
                             nbar(T,RL1,TL4,TL5),
                             TL2 = [adj2|TL5],
                             RL = RL1.


/* Refer rule first check the word <W> in the lexicon, if
   it found in the lexicon then it returns its category,
   otherwise, the parser assume that this word belongs to
   'noun' category and try to parse the sentence. This
   word along with its category is stored in a database
   'unknown_word'. A definition whose first element necce
   ssarily 'nil' is also stored. This first element is
   used later to identify that the definition of this un_
   known word is not given by the user. And ask user to
   enter the definition.
*/

refer(W,C) :-
            lexicon(W,C,_).
```

```
     refer(W,C) :-
                  !,C = noun,
                  asserta(unknown_word(word(W))),
                  asserta(unknown_word(word(noun))),
                  asserta(unknown_word(definition(nil,nil,nil,
                                                          nil))).

  /* refer(W,C) :-
                  retract(unknown_word(_)),
                  retract(unknown_word(_)),
                  write("What is the category of word <",W,">
                                                            : "),
                  readln(C),!,
                  asserta(unknown_word(word(C))),
                  asserta(unknown_word(definition(nil,nil,nil,nil)
  */


/******************** T R A N S L A T O R **********************/


     /* The translator takes the first symbol of the parser
        output and do some processing then take another one,
        this process is continue until the whole string get
        exhausted.
     */


     translate([]) :-
                     write_the_result.


     translate([H|T]) :-
                       !,check(H),
                       translate(T).


     /* The symbols are checked here, each internal node in the
        parse tree represent some function.
     */

     /* 'nbar', this rule return the list of its sublinks,
        hence do nothing.
     */

     check(nbar2) :-
                   nbar2.
```

```
/* if 'nbar' has only one parameter then it returns that
   parameter unaltered.
*/

check(nbar1) :-
              nbar1.


/* 'np' with one parameter returns the parameter unchang_
   ed.
*/

check(np1) :-
            np1.


/* 'np' with two parameter calls instance rule, swaps its
   parameter and pass them to instance rule.
*/

check(np2) :-
            retract(node(D1)),!,
            retract(node(D2)),!,
            np2(D1,D2).


/* Since, 'pph'  simply returns a list of its parameter
   therefore, it is represented as a fact.
*/

check(pph2) :-
             pph2.

check(vp2) :-
            vp2.

check(sent2) :-
              sent2.

check(cat) :-
            category.


/* 'sbar' with three parameters, it calls instance subro_
   utine and passes its first and third parameter to this
   routine.
 */

check(sbar3) :-
              retract(node(Def1)),!,
              retract(node(Def2)),!,
              retract(node(Def3)),!,
                    retract(node(_)),!,
                    Def2 = function(instance),
                    asserta(node(Def3)),
                    asserta(node(Def1)),
                    inst,!.


/* Determiner is ignored by our system. It puts the word
   'nil' instead of the determiner.
*/

check(det) :-
            retract(node(word(_))),!,
            asserta(node(word(nil))).


/* Words are stored in the stack 'node' */

check(W) :-
          asserta(node(word(W))).


/* When the symbol encountered, the translator pops the
   word just stored in the stack <node>, and find the de_
   finition of the word and push it on to the stack. This
   is what the function of pre_terminals in the parse
   tree.
*/


category :-
          retract(node(word(Word))),
          find_def(Word,Def),
          asserta(node(Def)).


/* It finds the definition of the word. If the word is
   not found in the lexicon it ask the user to enter its
   definition or the function name and store it in the
   unknown_word database.
*/

find_def(Word,Def) :-
                    lexicon(Word,_,Def).

find_def(Word,Def) :-
                    retract(unknown_word(Data)),
                    asserta(temp_store(Data)),
                    Data = word(Word),
                    retract(temp_store(W)),
                    asserta(unknown_word(W)),
                    retract(temp_store(word(Category))),
                    asserta(unknown_word(word(Category))),
                    retract(temp_store(Definition)),
                    entered(Word,Category,Definition),
                    retract(unknown_word(Def)),
                    asserta(unknown_word(Def)),
                    empty_temp_store.


/* This rule is used to maintain the unknown_word data_
   base.
*/

empty_temp_store :-
                  retract(temp_store(Data)),
                  asserta(unknown_word(Data)),
                  fail.

empty_temp_store.


/* This rule checks, whether the definition of unknown
   word <Word> has been entered already or not.
*/

entered(Word,Category,
 definition(From,_,_,_)) :-
                        From = nil,
                        definition_or_function(Word,
                                                Category).


/* This rule ask the user whether the unknown word has a
   definition or is a subroutine name and route the con_
   trol accordingly.
*/

definition_or_function
      (Word,Category) :-
                       write("I assumed that the word <"
                             ,Word,">"),nl,
                       write("  belongs to <",noun,
                             "> category"),nl,nl,
                       write("The word <",Word,
                       "> has a definition or a function"),
                       nl,
                       write("Press < d / f > : "),
                       readchar(Ch),nl,
                       enter_def_or_fun(Ch,Word,
                                        Category).
```

```
/* It accepts the definition for the unknown word */

enter_def_or_fun
 ('d',Word,Category) :-
                      write("Enter the definition of <"
                            ,Word,">"),nl,
                      write(" whose category is <",
                            Category,"> : "),nl,nl,
                      write("Enter the FROM part -> "),
                      readln(From),
                      write("Enetr the ID part -> "),
                      readln(Id),
                      write("Enter the PRT part -> "),
                      readln(Prt),
                      write("Enter the WHERE part -> "),
                      readln(Where),
                      asserta(unknown_word(definition
                                 (From,Id,Prt,Where))).


/* It accepts the function name of the unknown word */

enter_def_or_fun
 ('f',Word,Category) :-
                      write("Enter the function name of "
                           ,"the word <",Word,">"),nl,
                      write(" whose category is <",
                            Category,"> : "),nl,nl,
                      readln(Function_name),
                      asserta(unknown_word(function
                                     (Function_name))).


/* 'nbar'and 'np' with one parameter, returns their
   parameter unchanged.
*/

nbar1.

np1.


/* 'pph' with two parameter returns a list of parameter,
   therefore always true.
*/

pph2.


/* 'nbar' with two parameters calls the instance rule and
    pass its siblings to it as parameters.
*/
```

```
nbar2 :-
       retract(node(Def1)),
       retract(node(Def2)),
       Def2 = function(instance),
       asserta(node(Def1)),
       inst.


/* 'np' with two parameter, if one of the parameter is
   'nil' then it returns the other parameter unchanged.
   If both the parameter are non 'nil' then it swaps the
   two parameter and calls instance rule.
*/

np2(D1,D2) :-
             D1 = word(nil),
             asserta(node(D2)),
             np1.

np2(D1,D2) :-
             D2 = word(nil),
             asserta(node(D1)),
             np1.

np2(D1,D2) :-
             asserta(node(D1)),
             asserta(node(D2)),
             inst.


/* 'vp' with two parameter returns a list of parameter,
   therefore always true.
*/

vp2.


/* 'sent' it calls parameter #2.function and pass its
   parameters to this function.
*/

sent2 :-
       retract(node(Def)),
       retract(node(Fun)),
       Fun = function(instance),
       asserta(node(Def)),
       inst.


/* It pops the top of stack <node> two times and get two
   data. It then calls 'instance' rule and pass these
   data as parameter to it.
*/
```

```
inst :-
     retract(node(Inf)),!,
     retract(node(Sup)),!,
     instance(Sup,Inf).


/* 'instance', if either of two parameter is 'nil' it
   returns the other parameter unaltered. If either of
   the parameter is a subroutine name, it calls that
   routine and passes the other as its parameter.
   If both the parameter are definitions then it calls
   HAS_A function instead.
*/


instance(Sup,Inf) :-
                     Sup = word(nil),
                     asserta(node(Inf)).

instance(Sup,Inf) :-
                     Inf = word(nil),
                     asserta(node(Sup)).

instance(Sup,Inf) :-
                     Sup = function(N),
                     N = cp_id_to_prt,
                     function_id_prt(Inf).

instance(Sup,Inf) :-
                     Inf = function(N),
                     N = cp_id_to_prt,
                     function_id_prt(Sup).

instance(Sup,Inf) :-
                     has_a(Sup,Inf).


function_id_prt
(definition(M,N,_,P)) :-
                         asserta(node(definition(M,N,N,P))).


/* This routine has been divided into three cases */

has_a(definition(M,N,O,P),
     definition(W,X,Y,Z)) :-
                              cases(M,N,O,P,W,X,Y,Z).


/* This is case(1). It is applicable when the ID fields,
   i.e. the second element of the definitions are equal.
*/
```

```
cases(M,N,O,P,W,X,Y,Z) :-
                                N = X,
                                part_of_case1(M,N,O,P,W,X,Y,Z).


/* This is case(2), whan case(1) does not satisfy has_a
   function try this case.
*/

cases(M,N,O,P,W,X,Y,Z) :-
                                Y <> "nil",
                                union(M,W,Mw),
                                union(O,Y,Oy),
                                part_of_case2(Mw,N,Oy,P,W,X,Y,Z).

/* case(2) */

cases(M,N,O,P,W,X,Y,Z) :-
                                Z <> "nil",
                                union(M,W,Mw),
                                union(O,Y,Oy),
                                part_of_case2(Mw,N,Oy,P,W,X,Y,Z).


/* case(3), it is executed only when case(1) and case(2)
   are failed.
*/

cases(M,N,O,P,_,_,_,Z) :-
                                union(P,Z,Pz),
                                asserta(node(definition
                                                    (M,N,O,Pz))).


part_of_case2(M,N,O,P,W,X,Y,Z) :-
                                          Z = "nil",
part_of_case21(M,N,O,P,W,X,Y,Z).

part_of_case2(M,N,O,P,_,_,_,Z) :-
                                          P <> "nil",
                                          concat(" and ",Z,K),
                                          concat(P,K,H),
                                          asserta(node(definition
                                                          (M,N,O,H))).


part_of_case2(M,N,O,_,_,_,_,Z) :-
                                          asserta(node(definition
                                                          (M,N,O,Z))).

part_of_case21(M,N,O,P,_,_,_,_) :-
                                           asserta(node(definition
                                                           (M,N,O,P)))
```

```
        part_of_case1(M,N,O,P,_,_,_,Z) :-
                                        P = "nil",
                                        asserta(node(definition
                                                      (M,N,O,Z))).

        part_of_case1(M,N,O,P,_,_,_,Z) :-
                                        Z = "nil",
                                        asserta(node(definition
                                                      (M,N,O,P))).


        /* Union rule is used to concat two strings. If either of
           the string is 'nil' then 'union' returns the other
           string without concatenation. If both the strings are
           same then it returns either one.
        */

        union(X,Y,Z) :-
                       X = Y,
                       Z = X.

        union(X,Y,Z) :-
                       X = "nil",
                       Z = Y.

        union(X,Y,Z) :-
                       Y = "nil",
                       Z = X.

        union(X,Y,Z) :-
                       concat(X," U ",Zz),
                       concat(Zz,Y,Z).

/*****************************************************************/
```

REFERENCES

[1] BATEMAN,R. F. 1983. "A translation to encourage user modifiable man-machine dialogue". In Designing for Human-Computer Communication, Academic Press.

[2] CLEAL, D. M. and HEATON, M. O. . Knowledge-based Systems : Implication for Human-Computer Interfaces.

[3] CROFT, B. and LEWIS, D. 1987."An approach to Natural Language Processing for document retrieval". In Proceedings of the 10th Annual ACM SIGIR Conference on Research and Development in Information Retrieval, New Orleans.

[4] GAZDAV, K., et. al. . Generalised Phrase Structure Grammar, Basil Blackwell.

[5] GROSZ, B. J., et. al. 1986. Natural Language Processing, Morgan-Kaufman, Palo Alto, CA.

[6] GROSZ, B. J., et. al. 1987. "TEAM : An experiment in the design of transportable Natural-Language Interfaces". In Artificial Intelligence, Vol. 32, pp. 173-2434.

[7] HARRIS, L. A. 1977. "User-oriented database query with ROBOT natural language query system". In International Journal of Machine Studies.

[8] JONES, L. P., et. al. . "INDEX : The statistical basis for an investigation". In The Journal of the American Society for Information Science.

[9] JOSHI, A. K. 1985."Tree Adjoining Grammars : How much context-sensitivity is required to provide reasonable structural descriptions ?". In Natural Language Parsing - psychological, computation and theoretical perspectives, Cambridge University Press, pp. 206-150.

[10] JOSHI, A. K. 1985. An Introduction to Tree Adjoining Grammars, Deptt. of Computer and Information Science, Moore School, Univ. of Pennsylvania, Philadelphia PA.

[11] JOSHI, A. K. and VIJAY-SHANKER, K. 1985. "Some computational properties of Tree Adjoining Grammars". In 23rd Annual Meeting of the Association for Computational Languages, pp.8-12.

[12] JOSHI, A. K., et. al. 1975. "Tree Adjunct Grammars". In Journal of Computer and System Sciences, Vol. 10, pp. 136-163.

[13] JOSHI, A. K. and YOKOMORI, T., "Parsing of Tree Adjoining Grammars". In Technical Report, Department of Computer and Information Science, Univ. of Pennsylvania.

[14] KAPLAN, S. J. 1979. "Co-operative response from a portable natural language query system". In Technical Report, Deptt. of Computer and Information Science, Univ. of Pennsylvania.

[15] RICH, E. 1986. Artificial Intelligence, McGraw Hill, New York.

[16] ROBINSON, J. J. 1982. "Diagram : A grammar for dialogues". In Commnications of the ACM, Vol. 25, No. 1, pp. 27-47.

[17] SAGALOWIEZ, D. and SLOCUM, J. . "Developing NLP to complex data". SRI iNternational.

[18] SMEATON, A. 1986. "Incorporating syntactic information into a document retrieval strategy : An investigation". In Proceedings of the ACM Conference on Research and Development in Information Retrieval, Pisa, Italy.

[19] STONEBRAKER, M. 1988. "Current work on database system". In IEEE, Vol.2.

[20] TENNANT, H. 1980. Natural Language Processing, Petrocelli, Princeton.

[21] TURBO PROLOG TOOLBOX : User's Guide and Reference Manual.

[22] WALTZ, D. "Natural language access to a large database : An engineering approach". In Proceedings of the 4th International Joint Conference on Artificial Intelligence, Tbilisi, USSR.

[23] WOODS, W. A., et. al. 1972. "The Lunar sciences natural language interface system". Final report, BBN report 2378, Cambridge, Massachusetts.

[24] YOUNGER, D. H. "Recognition and Parsing of Context-free Languages in Time n3". In Information and Control, Vol. 10, No. 2, pp. 189-209.