

# **Improving the Portability of Natural Language Interfaces Through Learning**

by

**Gregory John McClement**

B.Sc., The University of Regina, 1989

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF  
THE REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES  
DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming  
to the required standard

---

THE UNIVERSITY OF BRITISH COLUMBIA

November, 1993

© Gregory John McClement, 1993

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of Computer Science

The University of British Columbia  
Vancouver, Canada

Date Oct 6 1993

## Abstract

One major concern with natural language interfaces is dealing with the problem of porting the system from one domain to another. Early approaches required that significant parts of the system be rewritten. Improvements have been made to the point that a system can be configured for a new database by a database expert who has no special knowledge of Computational Linguistics. Unfortunately, this has not yet lead to the widespread use of natural language interfaces. Proposed here is a natural language interface that can be configured by a user that has no knowledge of Computational Linguistics and no knowledge of databases ie - no knowledge of formal query languages, no knowledge of relational databases and no knowledge about how the data is organized in the database. Such a user merely needs to know some specific information about the domain that is modelled by the database. For an employee database, specific information is knowledge about a particular employee, eg - their name, address, and department.

## TABLE OF CONTENTS

Abstract .....	ii
Table of Contents .....	iii
List of Figures .....	viii
Acknowledgement .....	xiii
Chapter 1 Introduction .....	1
1.1 Contributions .....	3
1.2 What Follows .....	3
Chapter 2 Natural Language Interfaces .....	6
2.1 Question Answering Systems .....	6
2.1.1 ELIZA (Pattern Matching) .....	7
2.1.2 LUNAR (Linear Paradigm) .....	8
2.1.3 PLANES/LIFER (Semantic Grammars) .....	11
2.1.4 Case Frame Based .....	13

2.2 Desirable Characteristics . . . . .	17
2.2.1 Internal Characteristics . . . . .	17
2.2.1.1 Portability . . . . .	18
2.2.1.2 Semantic Productivity . . . . .	20
2.2.1.3 Minimum Coverage . . . . .	21
2.2.2 User Interface . . . . .	22
2.2.2.1 Habitable (User-friendly) . . . . .	22
2.2.2.2 Cooperative . . . . .	23
2.2.2.3 Robust with Respect to "Noise" . . . . .	27
2.2.2.4 Handling of Discourse Phenomena . . . . .	28
2.3 Learning Language . . . . .	31
2.3.1 Syntax . . . . .	31
2.3.2 Semantics . . . . .	32
 Chapter 3 System Overview . . . . .	 38
3.1 Objectives . . . . .	38
3.2 Knowledge Base Components . . . . .	38
3.2.1 Domain Independent Knowledge . . . . .	39
3.2.2 Domain Dependent Knowledge . . . . .	40
3.3 Processing Flow of Control . . . . .	41
3.4 Implementation Details . . . . .	47
3.5 Examples . . . . .	47

Chapter 4 Linguistic Knowledge .....	57
4.1 Lexicon .....	57
4.1.1 Word Classes .....	58
4.1.1.1 Domain Independent (Predefined) .....	58
4.1.1.2 Domain Dependent .....	59
4.1.2 Morphing .....	60
4.1.3 Implementation of the Lexicon .....	60
4.1.3.1 Inverted index .....	60
4.1.3.2 Implementing the Inverted index .....	62
4.2 Grammar .....	64
4.2.1 Coverage .....	64
4.2.1.1 Passives .....	64
4.2.1.2 Relative Clauses .....	65
4.2.1.3 Wh-Question .....	65
4.2.1.4 Possessives .....	66
4.2.1.5 Imperatives .....	66
4.2.1.6 Noise Phrases .....	67
4.2.1.7 Semantic Grammar Based-Phrases .....	67
4.2.2 Representation .....	69

<b>Chapter 5 Becoming Portable Through Learning</b> .....	<b>71</b>
<b>5.1 Tokenization</b> .....	<b>71</b>
<b>5.2 Parsing</b> .....	<b>71</b>
<b>5.3 Learning Frames</b> .....	<b>73</b>
5.3.1 Case Frames .....	73
5.3.2 Object Frames .....	74
5.3.3 Minimal spanning Trees .....	75
5.3.4 Ambiguity .....	110
<b>5.4 Learning Word Meanings</b> .....	<b>111</b>
 <b>Chapter 6 Question Answering</b> .....	 <b>114</b>
<b>6.1 Example</b> .....	<b>114</b>
6.1.1 Tokenizing .....	114
6.1.2 Parsing .....	119
6.1.3 Semantic Analysis .....	123
<b>6.2 Response Generation</b> .....	<b>134</b>
<b>6.3 Learning Word Meanings</b> .....	<b>134</b>
<b>6.4 Handling Ambiguity</b> .....	<b>135</b>
<b>6.5 Controlling the Dialogue</b> .....	<b>136</b>

Chapter 7 Conclusion . . . . .	138
7.1 Contributions of this Thesis . . . . .	138
7.2 Possible Enhancements . . . . .	140
7.2.1 Enhancements to Noun Phrase Interpretation . . . . .	140
7.2.2 Learning Grammatical Structures . . . . .	146
7.2.3 Enhancements to Verb Phrase Interpretation . . . . .	146
7.2.4 Enhancements of the Parser . . . . .	147
7.2.5 Enhancements Using Learning . . . . .	147
7.2.6 Enhancements Using Frames . . . . .	148
7.2.7 Enhancements to the Database Interface . . . . .	149
7.2.8 Inferences . . . . .	149
 Bibliography . . . . .	 151
 Appendices . . . . .	 155
A Relational Database Terminology . . . . .	156
B Grammar . . . . .	159
C Database Listing . . . . .	172
D Database Interface . . . . .	178
E Test Runs . . . . .	180



## **List of Figures**

Figure 2.1 - Match - Response Patterns . . . . .	7
Figure 2.2 - Sample Cases . . . . .	14
Figure 2.3 - Case Frame For Fix . . . . .	15
Figure 2.4 - Non-passive Sentence . . . . .	16
Figure 2.5 - Passive Sentence . . . . .	16
Figure 2.6 - Ambiguous Question . . . . .	26
Figure 2.7 - Paraphrases . . . . .	26
Figure 2.8 - Simple Choices . . . . .	27
Figure 2.9 - Ellipses . . . . .	28
Figure 2.10 - Anaphora . . . . .	29
Figure 2.11 - Vagueness . . . . .	30
Figure 2.12 - Part of a Concept Hierarchy . . . . .	34
Figure 2.13 . . . . .	35
Figure 2.14 - Arithmetic Field Acquisition (MAP83) . . . . .	36
Figure 3.1 - Knowledge Source Diagram . . . . .	39
Figure 3.2 - Data Flow Diagram . . . . .	43
Figure 3.3 - Case Attachment . . . . .	45
Figure 4.1 - Word Classes . . . . .	59
Figure 4.2 - Part Table . . . . .	61
Figure 4.3 - Inverted Index for the Producer Column . . . . .	61
Figure 4.4 - Part of a TRIE . . . . .	63

Figure 4.5 - Different Tokens . . . . .	63
Figure 4.6 - Sentence Form . . . . .	64
Figure 4.7 - Relative Clauses . . . . .	65
Figure 4.8 - Possessives . . . . .	66
Figure 4.9 - Imperative Sentences . . . . .	66
Figure 4.10 - Noise Phrases . . . . .	67
Figure 4.11 - Example Employee Frame . . . . .	67
Figure 4.12 - Semantic Grammar for Recognizing Object and Attributes . . . . .	68
Figure 5.1 - Cases and Fillers for Sample Sentence . . . . .	72
Figure 5.2 - Case Frame for Works . . . . .	74
Figure 5.3 - Employee Object Frame . . . . .	75
Figure 5.4 - Tables Used by Examples . . . . .	77
Figure 5.5 - Database Graph for Table of Figure 5.4 . . . . .	77
Figure 5.6 - Three Minimal Spanning Trees for "Smith works as a clerk for 800 dollars" . . . . .	78
Figure 5.7 - GMST . . . . .	80
Figure 5.8 Example minimal spanning trees connecting "Smith", "Clerk", "800", "Research", and "Ford" . . . . .	83
Figure 5.9 - Another Spanning Tree For "Smith", "Clerk", "800", "Research", and "Ford" . . . . .	84
Figure 5.10 - Generalized Minimal Spanning Tree . . . . .	85
Figure 5.11 - 'Join' Edge Cost . . . . .	86
Figure 5.12 - Tree Cost Calculation . . . . .	88

Figure 5.13 - Establish Value to Database Graph Node Correspondence . . . . .	90
Figure 5.14 - Values Queue . . . . .	91
Figure 5.15 - First MST . . . . .	92
Figure 5.16 - New Values Queue . . . . .	92
Figure 5.17 - Connecting Ford . . . . .	94
Figure 5.18 - New Values Queue . . . . .	95
Figure 5.19 - Connecting in 800 . . . . .	96
Figure 5.20 - New Values Queue . . . . .	97
Figure 5.21 - Connecting "Research" to "Smith" . . . . .	98
Figure 5.22 - Next Values Queue . . . . .	99
Figure 5.23 - Connect in "Clerk" . . . . .	100
Figure 5.24 Pseudo-Code Which Generate GMSTs such that $M \leq \text{cost}(\text{GMST}) \leq N$ .	101
Figure 5.25 - Choice 1 . . . . .	102
Figure 5.26 - Choice 2 . . . . .	102
Figure 5.27 - Value to Node Mapping . . . . .	105
Figure 5.28 - Presenting the User with Different Interpretations . . . . .	107
Figure 5.29 - Search from Each New Node to GMST . . . . .	109
Figure 5.30 - Search from GMST to all New Nodes . . . . .	109
Figure 6.1 - Example Sentence . . . . .	114
Figure 6.2 - Initial Position in the Sentence . . . . .	115
Figure 6.3 - Initial Position in the TRIE . . . . .	115
Figure 6.4 - Sentence . . . . .	115
Figure 6.5 - Inverse Index . . . . .	115

Figure 6.6 - Sentence After 'h' . . . . .	116
Figure 6.7 - TRIE after 'h' . . . . .	116
Figure 6.8 - Sentence after 'o' . . . . .	116
Figure 6.9 - TRIE after 'o' . . . . .	116
Figure 6.10 - Tokens of the Sentence . . . . .	117
Figure 6.11 - Possible Sentence Structure . . . . .	120
Figure 6.12 - Possible Syntactic Structure . . . . .	120
Figure 6.13 - Possible Syntactic Structure . . . . .	120
Figure 6.14 - Possible Syntactic Structure . . . . .	121
Figure 6.15 - Cases of the Only Parse Accepted . . . . .	121
Figure 6.16 - verbPhrase/3 . . . . .	121
Figure 6.17 - nounPhrase/5 . . . . .	122
Figure 6.18 - Case Frame for Earn . . . . .	124
Figure 6.19 - Case Frame for Work . . . . .	124
Figure 6.20 - Object Frame for Employee . . . . .	125
Figure 6.21 - Selecting Frames . . . . .	126
Figure 6.22 - Query Groups . . . . .	129
Figure 6.23 - Graph 1 . . . . .	130
Figure 6.24 - Graph 2 . . . . .	130
Figure 6.25 - Union . . . . .	130
Figure 6.26 - Main Query GMST . . . . .	131
Figure 6.27 - Sub-query GMST . . . . .	131
Figure 6.28 - SQL for the Sub-query . . . . .	132

Figure 6.29 - SQL for the Main Query . . . . .	133
Figure 6.30 - Possible Ambiguity . . . . .	135
Figure 6.31 - Asking for Input . . . . .	136
Figure 6.32 - Selection Option #1 . . . . .	136
Figure 7.1 - Job Class . . . . .	144
Figure 7.2 - Another Form of Classes . . . . .	144
Figure 7.3 - Preposition used as Particles . . . . .	147
Figure A1.1 - Oracle Sample Database . . . . .	156
Figure D.1 - whereLocation/2 . . . . .	178
Figure D.2 - where_prefix/3 . . . . .	178
Figure D.3 - SQL Access Predicates . . . . .	179

## **Acknowledgements**

I am indebted to my research advisor Dr. Richard Rosenberg. His initial advice helped me to select a concrete problem to solve and his insight, patience, and supervision fostered the development of a realistic solution. His careful editing and suggestions enhanced my work.

I would also like to thank my parents, John and Joan, who have always supported me in my endeavours and special thanks to my wife, Dawn, who encouraged me to further my education.

# Chapter 1

## Introduction

Everyday, more people begin to use databases for their work. Many of these people are not experts on database technology and do not want to become experts, but they still want access to the information in databases. In many cases, access to the information in the database is obtained through a programmer who writes programs to produce reports for the user. Many people are unhappy with both the cost of producing a report and the delay between asking for a report and receiving the report. This has lead to the demand for user-friendly database management systems which a naive user can employ directly.

A number of different approaches have been taken to try to achieve user friendliness. For each approach, there is a different tradeoff between ease of use and sophistication of the results. The two most common approaches are to provide a menu driven interface and a formal query language. A less common approach is to use interfaces which interpret natural language.

Most commercial database management systems (DBMS) have a menu driven interface which allows the user to produce reports. Typically, the user is asked to select a number of

database tables to use in the production of the report. The system then presents the user with a report template which can be manipulated manually to produce the final form of the report. After the user is satisfied with the template, the report can be generated. The disadvantages of this approach are that production of arbitrarily complicated reports is not possible and the user must know something about the internal structure of the database.

Another approach to providing access to databases has been to provide query languages such as SQL. Query languages are a powerful tool for generating reports quickly and accurately. Unfortunately, query languages are not easy to master for people inexperienced with computer programming languages. Many people do not have the background necessary for learning query languages. Although they provide a means to produce reports quickly and accurately, many users find query language too difficult to use. Another problem with query languages is that users still must know about the structure of the database.

The less common approach to providing access to information in databases is to use natural language interfaces. The goal of this approach is to allow the user to access the database in a familiar way. This approach does not require that the user learn unnatural query languages or that the user know how the database is structured internally. Ideally, the users would be able to tell the natural language interface what they wanted using concepts that are familiar. One of the problems with natural language interfaces is that a database expert with some knowledge of Computational Linguistics must be used to transfer them from one database to another. This thesis explores one approach that could be used to allow the users themselves to configure the natural language interface for a database.



## **1.1 Contributions**

The contribution of this thesis is to provide a simple means to translate the user's knowledge into the knowledge of the system. The approach presented in this thesis, allows the system to access the knowledge of the user in order to configure itself to a new database. In order to achieve this, the user does not need to know about the internal structure of the database or learn a formal language. The user merely describes the data in the database using words and concepts of his or her own choosing.

The interpretations of sentences are represented with a special type of semantic net called a minimal spanning tree. The relationship between utterances of the user and the data in the database is determined through use of these minimal spanning trees. The minimal spanning tree and its use in configuring a natural language interface comprise the major contributions of this thesis. This is what allows the system to be configured so easily.

## **1.2 What Follows**

This section briefly describes the contents of the remaining chapters of this thesis. Chapter 2 contains a review of other approaches to developing natural language interfaces. This includes general purpose natural language interfaces as well as systems that were developed to explore how learning could be used to enhance performance. A summary of desirable characteristics of interfaces is included.

Chapter 3 contains an overview of the system. A description of the high-level control structures of the system is presented along with information about the derivation of knowledge. A sample run of the system concludes the chapter.

Chapter 4 contains background information about the handling of the lexicon and the grammar. Implementation details are presented along with information about the extent of coverage of the lexicon and grammar.

Chapter 5 describes the major contribution of this thesis - the spanning tree approach to configuring a natural language interface. This chapter proceeds by examining a number of examples of the analysis process in action. The algorithm is presented along with a number of considerations related to implementing the algorithm efficiently.

Chapter 6 shows how the spanning tree data structure is used when interpreting questions. The process of analysis is examined in detail for a sample sentence, from the initial tokenization of the sentence to the final response to the question.

Chapter 7 contains a summary of the contributions of this thesis along with a detailed description of a number of enhancements that could be made to the system to produce a more robust and powerful natural language tool.

The appendixes contain information for readers unfamiliar with relational database terminology (Appendix A), the definite clause grammar used by the parser (Appendix B), a

listing of the tables in the database used for the examples (Appendix C), a description of the interface to ORACLE (Appendix D), and test runs of the system (Appendix E).

## **Chapter 2**

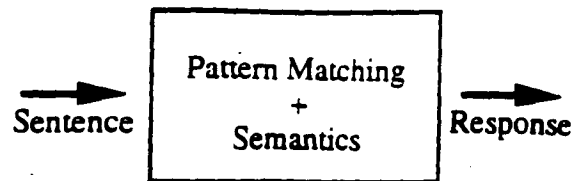
# **Natural Language Interfaces**

Many attempts to develop natural language interfaces have been made during the last 30 years. Some have been more successful than others but all serve to illustrate that natural language analysis is a difficult problem. What follows is a description of some of these attempts. Some well known systems are presented as examples of a particular approach to dealing with the problem of interpreting language. After discussing these examples, general observations about natural language interfaces will be summarized. This section concludes with a description of natural language systems that have used learning to enhance performance.

## **2.1 Question Answering Systems**

During the first 30 years of computational linguistics many approaches to natural language processing were proposed. This section reviews some approaches which are related to the work done in the thesis.

### 2.1.1 ELIZA (Pattern Matching)



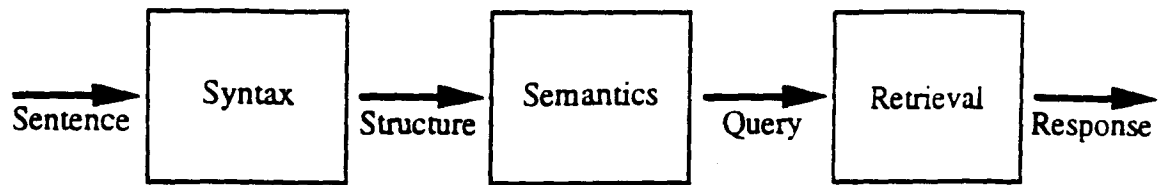
ELIZA is a pattern matching system developed in the mid-60's by Weizenbaum (WEIZEN66, WEIZEN67). This program attempts to maintain a conversation with a person. The program has the role of a therapist and the person is a patient. For this system, pattern matching provided a simple means of analyzing input sentences in order to determine a reasonable response. No sophisticated analysis of the input was attempted, instead the system associated a response pattern with each sentence recognition pattern. Some typical patterns are shown in Figure 2.1.

Match Pattern	Response Pattern
I am X	How long have you been X?
X you Y me	What makes you think I Y you?

**Figure 2.1 - Match - Response Patterns**

These examples illustrate that the system is not analyzing potentially large parts of the input sentences. In order to analyze the sentence entirely, the system would have to be provided with a large number of patterns. The need to provide large number of patterns makes this approach impractical as a base for modern question answering system, although pattern matching can still be used to enhance other methods of sentence analysis or to provide sentence analysis where the number of possible responses is limited. Further work in pattern matching approaches was explored in COLBY71.

### 2.1.2 LUNAR (Linear Paradigm)



LUNAR, WOODS85, is characteristic of systems which were developed according to a linear paradigm. Under this approach, syntactic and semantic analysis were performed separately. After initially analyzing the input using only syntactic information, the results were analyzed by a component that used semantic information. Unlike pattern matching approaches, the interpretation of the sentence was developed by combining the results of analysis of the components of the sentence. The theoretical base for syntactic analysis was Chomsky's Transformational Generative Grammars, developed in the late 1960's.

Transformational Generative Grammar is a mathematical formalism which is used to define the mapping of the surface structure of a sentence to its deep structure. The *surface structure* is the sentence itself and the *deep structure* represents an equivalence class of sentences with common meaning. One purpose of this mapping is provide a means to ignore insignificant differences between sentences. The mapping from surface to deep structure is defined using a grammar that generates parse trees and transformations of those parse trees. The following sentences have the same deep structure and are related to each other by the passive transformation.

John bought a car. A car was bought by John.
---

Augmented Transition Networks (ATNs) were proposed in WOODS70 as a means to efficiently analyze the syntax of sentences. ATNs are derived from recursive transition networks (push down automata) by adding registers to keep track of information collected during parsing and adding tests on the transitions of the machine. The ATN analysis produces the "deep structure" of the sentence which is then analyzed by the semantic component of the system.

The analysis of the deep structure of the sentence produces a domain dependent interpretation which is used to answer questions or perform some task. Procedural semantics (WOODS85) is a common method of providing this interpretation. For procedural semantics, the meaning of a sentence is defined using procedures that are evaluated to implement a command, or answer a question. The procedures are defined in terms of the information in a particular domain.

Because syntactic and semantic analysis are separate, the syntactic information is relatively domain independent. While this is a desirable property it has an undesirable consequence. Since no semantic analysis is performed by the ATN during the parse of a sentence, many parses which are not semantically sound can be generated. The ATN does not stop analyzing the sentence as soon as a semantic inconsistency is detected, so much work can be wasted producing parses which did not need to be produced. There are also disadvantages which are associated with using an ATN itself.

The ATN after backtracking will reanalyse components of the sentence because it does not

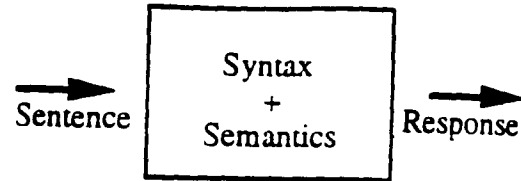
keep track of previous work. To avoid this problem, the ATN control structure must be modified to keep track of partial parses. Problems also arise when the input contains deviations from expected syntax. The ATN is dependent on the current state during the parse to control the alternatives expected and so it does not respond well to deviations. Procedural semantics is also a source of some problems.

Under procedural semantics, the semantic information is domain dependent. Providing the semantic information requires an expert and is time consuming. Procedural semantics is defined in terms of the underlying database. This means that general facts such as "All flights serve meals." and disjunctive facts such as "Flight F1 serves breakfast or lunch" cannot be represented in the system because they must be defined in terms of the query language of the database management system and query languages do not have such expressive power, ALLEN87. Attempts to address this problem involve the use of an intermediate representation language (IRL) which serves as a domain independent representation of semantic information as well as an interface to different DBMS query languages.

Most of the significant problems with ATNs arise from the underlying approach - the separation of syntax and semantics. Later approaches attempted to reduce this problem by combining these stages of analysis. Semantic Grammar-based systems represented the first attempt to address the problem of combining syntax and semantics.



### 2.1.3 PLANES/LIFER (Semantic Grammars)



Semantic Grammars were developed to try to deal with the problems of approaches that separated syntactic and semantic analysis. Semantic grammars instead of using syntactic categories of phrases such as noun phrase or verb phrase, used semantic categories such as "plane name" or "present command". Semantic grammars provided a simple yet effective means for applying semantic constraints during the parse. But as a result each different domain of discourse required different semantic categories.

SOPHIE (BUBR79) is a semantic grammar-based system developed to teach debugging of electronic circuits. For this system, measurements, voltages and parts are some of the semantic categories. The PLANES systems (PLANES76) which accessed a database about aircraft contained semantic categories such as plane type, flights and flight hours. The LIFER system (LIFER77) used with a naval database had ship, attribute, and port as semantic categories. There is a considerable variety of semantic categories among these systems.

Semantic grammars are not a new way of representing sentence structure. They are a new way of using existing formalisms to represent structure. Because of this, tools for producing parse trees of inputs can be used to produce semantic grammar parses. During the parsing process the interpretation of the input can be constructed.

To handle interpretation of the input, there are query templates associated with the semantic grammar rules. During analysis of the input, components can be identified and later inserted into a particular query template. These templates are combined to produce procedures that are implemented as a result of the query. ATNs or DCGs provide an environment where the construction can occur.

A number of benefits are derived from the use of semantic information during the parsing process. The parser will not generate parses that have no meaning. A related benefit is that there is a reduction in the total number of grammatical parses. This means that inputs that are syntactically ambiguous will not be ambiguous using the semantic grammar, so this system will be able to generate all parses quickly and there will be fewer parses. Spelling checking is also more efficient because the semantic categories limit the number of alternative words that must be checked. Not all of the benefits are related to efficiency.

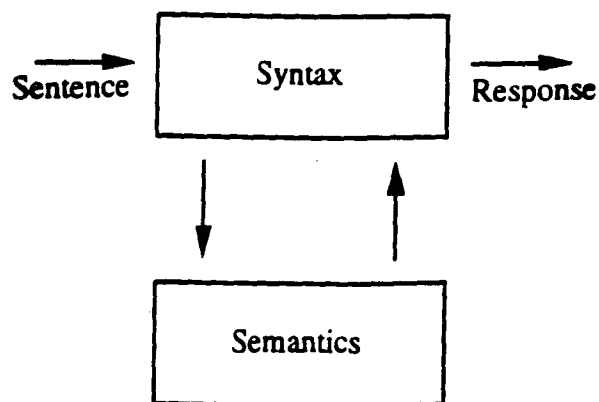
These systems can handle complex discourse phenomena effectively as well. Ellipsis can be handled by partially parsing inputs and determining their semantic categories. Once the categories are known, previous inputs can be scanned looking for components in the same category. Once found the old component can be replaced by the new component in order to produce an interpretation. Semantic grammars were found to limit the number of alternatives enough to make this approach effective.

The disadvantages of semantic grammars arise directly from the use of semantic categories. Different domains have different semantic categories and so have different semantic

grammars. To move the system from one domain to another requires specifying the grammar again. Defining the grammar is time consuming and complex. The use of semantic categories affects the generality of semantic grammars. Other problems arise from this lack of generality. Semantic grammars need to contain many rules in order to have broad coverage of the language. One reason for this is that general syntactic phenomena are stored redundantly in the grammar. For example, handling the passive form of sentences requires that productions be added for each semantic verb type. If some verbs are missed then the system will not always accept passive sentences. This can frustrate users.

A semantic grammar is one means to integrate syntactic and semantic analysis. However, the problems with this approach prevent the development of general purpose natural language interfaces. Of the other approaches developed to try to handle the problem of integrating syntax and semantics, case frame based approaches were the most successful.

#### 2.1.4 Case Frame Based



Case frame systems are based

upon case frames which contain syntactic and semantic information about a verb and related noun phrases. The case frame has two main components, patterns for recognizing an invocation of the frame in the input and a number of cases that are semantically related to

the concept the case frame represents. The case frame represents some relation between things and the cases represent what is related.

Semantic and syntactic constraints are associated with each case of the case frames.

Semantic constraints typically are based upon the role that the case has in the relation that the case frame represents. There have been numerous attempts to define a complete set of roles. Such a definition is called a case system. Figure 2.2 contains some sample cases from FILLMORE71.

<b>Agent</b>	The instigator of the event
<b>Object</b>	The entity that moves or whose position or existence is in consideration
<b>Instrument</b>	The stimulus or immediate physical cause of an event

**Figure 2.2 - Sample Cases**

In addition to the semantic information about roles, the case frame contains information used to recognize cases in the input. Case markers are associated with each case. During analysis of the input, case markers are used to identify case fillers. Case fillers are the sentence components that fill the role of a case. Markers are simple constraints on the position of the case in the input or they are patterns which precede the case filler. Patterns usually recognize prepositions.

Figure 2.3 contains a case frame for the verb "fix". As mentioned earlier, associated with each case frame there is a pattern that is used to recognize an occurrence of the case frame. For this case frame, an occurrence of "fix" (or any variation of it) in a sentence would indicate that this case frame should be used to interpret the sentence. This case frame has

three cases, the agent, the instrument and the object. Each of these cases is associated with a pattern (called case marker) for recognizing an occurrence of the case, and semantic

Fix		
Case	Marker	Description
AGENT	Logical Subject	• a person
INSTRUMENT	with	• a tool
OBJECT	Logical Object	• inanimate

**Figure 2.3 - Case Frame For Fix**

constraints. The first case is the agent case. The agent appears as the syntactic subject of non-passive sentence or the filler of a case marked with 'by' for passive sentences. In other words, this case is marked by position for non-passive sentences or with the preposition 'by' for passive sentences. There is a semantic constraint that the filler of the case be a person and so only noun phrases that refer to people could fill this case. The second case is the instrument. The instrument case represents the tools used to perform the act of fixing. This case is marked by the preposition "with". That means that the case filler must be part of a prepositional phrase beginning with the word "with". The semantic constraint on the filler is that it must be a tool. Semantic constraints depend on the application and have been arbitrarily chosen for this example. The last case is the object case. The object case is marked by position. The filler appears as a syntactic object for non-passive sentence or as the syntactic subject for passive sentences. The semantic constraint on the object is that it must be inanimate. As seen in this example, case frames provide both syntactic and semantic information which can be used to analyze sentences.

Consider the non-passive sentence in Figure 2.4. In the sentence, the verb "fixed" is a marker of the case frame in Figure 2.3. Once this is realized, the system can use the case

frame to analyze the sentence. According to the case frame, "Jack" is taken to be the agent because of position. The sentence is not in passive form so the agent appears as the syntactic subject. The semantic information in

Jack fixed the bench with a hammer.

**Figure 2.4 - Non-passive Sentence**

the case frame can be used to confirm this choice. "Jack" refers to a person. Because the sentence is not passive, the object case which appears as the syntactic object is filled by

The bench was fixed by Jack with a hammer.

**Figure 2.5 - Passive Sentence**

"the bench" which is inanimate

according to the semantic constraints

for the object case. The instrument case

is filled by "the hammer" which is marked by the preposition "with" according to the case frame. For the passive form of sentence shown in Figure 2.5, the syntactic subject is not the agent case but instead it is the object case. The agent case filler is to be found marked with the preposition 'by'. This transposition occurs because the sentence is in passive form. As shown, the parser can apply both syntactic and semantic constraints by using case frames.

In addition to constraints on each individual case, there are constraints on the cases as a whole. Constraints may require that certain cases always be present, that some cases cannot occur together or that some cases must occur together. There is no constraint that case frames can only represent verb relations. Case frames can be used to represent noun phrase level structures.

Case frames are accessed during the parsing process to obtain syntactic and semantic

constraints. ATNs have been used to parse sentences in many case frame-based natural language interfaces (TAYROS75, WHITE85, BOOTH83). The additional semantic constraints brought to bear early on in the analysis were able to alleviate some problems normally associated with ATN parsing. The problem of grammatical errors still remains. Other approaches to parsing using case frames have attempted to deal with these problems (Hayes & Carbonnell - CARHAY83).

Case frames are able to effectively combine syntactic and semantic information. By using semantic information during the parsing process, case frame systems are able to reduce ambiguity and the parsing time but unlike semantic grammar systems the grammar remains compact and easy to understand. Case frames are also able to handle different forms of expression without requiring the extensive coding that semantic grammars require.

## **2.2 Desirable Characteristics**

During the development of natural language interfaces, many design issues came to light. These issues involve how easy the interface is to use and how easily it can be transferred to other systems. This section examines some of these issues with respect to the systems described earlier.

### **2.2.1 Internal Characteristics**

If natural language interfaces are to have widespread use, they must not only be easy to use

but they must possess characteristics which facilitate application to different problem areas. Many systems although easy to use, did not become popular because they were not easily transferred. This section covers the different ways that a system can be easy to transfer as well as some characteristics that increase the ease with which a system can be transferred.

### **2.2.1.1 Portability**

Portability of a natural language interface is the ease with which it can be reused. There are two aspects of portability, domain portability and database portability - actually database management system (DBMS) portability. Domain portability is the ease with which the interface can be used with a different database, i.e. - a different domain of discourse. Database portability is the ease with which the interface can be used with a different database management system (for example, transferring from a relational DBMS to a network DBMS).

The systems described earlier had portability of varying degrees. LUNAR attained portability by separating syntactic from semantic analysis. The syntactic analysis component of LUNAR increased portability by using a single grammar for every domain. The reference to the domain as well as the DBMS was contained in the procedural semantics of the systems and in the lexicon. As a consequence, the semantic component and the lexicon were neither domain independent nor database independent and had to be rewritten after a change of either. LUNAR had portability of only the part of the syntactic component that contained the grammar. In contrast to LUNAR, semantic grammar systems



were in no significant aspect portable. By combining syntactic and semantic information into the grammar in order to increase parsing efficiency, semantic grammars sacrificed portability. Changing the domain required the grammar to be reconstructed from scratch using semantic concepts of the new domain.

One technique for increasing domain portability is to use an inverted index<sup>1</sup> of words in the database. This approach was used successfully by Harris in ROBOT<sup>2</sup> (HARRIS77, HARRIS84). Although still based upon the linear paradigm - syntactic parsing using an ATN followed by semantic analysis, Harris was able to make use of information in the database to such a degree that he reported that systems could be ported from one domain to another by a DBA in a week to two weeks.

A technique for increasing database portability is to use an intermediate representation language, IRL, PG87. For systems using IRLs, the result of parsing and interpreting an input is a sentence in the IRL which can later be converted to a database query. The IRL is an interface between the database management system and the natural language interface. Having a clearly defined interface means that transferring the system from one DBMS to another requires only rewriting of the component that translates IRL to the query language. Another advantage as shown in the Natural Language Corporation's NLI, is that the system may use the meaning in the IRL sentence to do things other than access a database.

---

<sup>1</sup> - see section 4.1.3.1

<sup>2</sup> - Later INTELLECT

Closely associated with the portability of a system is the ease with which it can be maintained. During the life of a database not only will changes to the database occur but new users with varying expectations will use the system. It is important that the cost of maintaining the system not be too great otherwise it will not be maintained and then, eventually, not used as it fails to keep up with changes to the database.

#### **2.2.1.2 Semantic Productivity**

In order to interpret utterances, natural language interfaces must contain syntactic and semantic knowledge in declarative or procedural form. This knowledge determines the utterances that the system can interpret - its coverage. Coverage can be too broad so that incorrect interpretations are made or too narrow so that intelligible inputs will not be interpreted. The knowledge base must be large enough to provide acceptable coverage. Semantic (or syntactic) productivity is a qualitative measure of how effective a formalism is in producing acceptable coverage.

Semantic productivity relates how implicit the knowledge of language is represented in the system. The least productive systems list every allowable utterance and its interpretation. Semantic grammars are at the low end of the productivity scale because they must contain rules for the different forms of expression for each group of semantic categories. Case frame based systems are more productive because the case frames provide a compact form for representing information and yet still allow many forms of expressions to be parsed. High productivity is closely related to high portability.

### **2.2.1.3 Minimum Coverage**

The systems described earlier attained broad coverage of language by increasing the amount of grammatical knowledge they contained. The goal was to be able to parse as many of the inputs as possible. The systems contained a single grammar for use with many different users who had many different styles of English. As observed in LEHMAN, if the systems instead had different grammars for each user then they would not have to handle such a large variety of forms of expression because each user's statements tend to have a certain fixed style. Minimizing coverage is the problem of trying to make sure that the parser will not accept more forms of expression than the current user will offer.

Benefits can be achieved by minimizing coverage. Sentences that were ambiguous relative to the grammars set up to handle all user's utterances, will not be ambiguous relative to a grammar set up for a particular user. By reducing the number of acceptable sentences, the parsing process becomes faster. It is demonstrated in LEHMAN that using a single grammar for all users is not as efficient as having a separate grammar for each user, because each user has to pay for other users' different forms of expression. Minimizing coverage is a difficult problem because it requires that the grammar be customized for each user.

## **2.2.2 User Interface**

The standards used to judge a natural language user interfaces are based upon the expectations of the users. Much work has been devoted not only to developing natural language interfaces that satisfy user expectations but also to determining what the user expectations are. Meeting user expectations is important to ensure that the interface becomes a valuable tool and not a source of frustration. This section attempts to provide a summary of the characteristics that users expect a natural language interface to have.

### **2.2.2.1 Habitable (User-friendly)**

A system is habitable if it can correctly interpret many different utterances that express the same idea. Habitable systems seem consistent to the user. Systems that are not require that the user search for an acceptable form of expression. With nothing to guide the user, this search can involve several tries and lead to frustration. Systems that are not habitable force users to learn what is acceptable and what is not acceptable. As a consequence the users must use restricted forms of English, if they want to use the system.

Pattern matching and semantic grammar based systems are more likely to not be habitable because general syntactic rules must be incorporated in each pattern or semantic grammar rule. Syntactic parsers have been able to achieve broad coverage. This is demonstrated by the commercial system of Harris, INTELLECT. He reports that one of the primary issues that had to be dealt with to satisfy commercial expectations, was coverage of the grammar

(HARRIS84). Prototype versions without broad coverage were thought to be inconsistent and annoying and were not commercially acceptable. INTELLECT did not find broad acceptance until habitability was dealt with.

Making a system more habitable requires that the coverage of the system be increased to handle more forms of expression. This can lead to problems because the coverage is not minimized. There is a compromise between habitability and minimizing the coverage.

#### **2.2.2.2 Cooperative**

Human communication is a complex form of interaction which is not only performed using language. In contrast to this, many natural language interfaces constrain interaction with the user to be questions followed by direct answers from the system. Much work has been done to try to increase the sophistication of natural language interfaces. Cooperative user interfaces provide more sophisticated dialogues in an attempt to work with the user and not just respond to the user's explicit requests. Many features have been developed to provide a cooperative system. This section describes a few.

A system that provides cooperative responses to questions does not necessarily answer a literal interpretation of the question. There is an attempt to anticipate follow-up questions, to respond before being explicitly requested, and to provide more information than was requested. Anticipating can involve simple pattern matching or complex modelling of the user's intentions and plans.

One problem to deal with is responding to questions with a negative answer (RENDEZVOUS, COOP - KAPLAN84). The question

Which of Dr. Lee's students is enroled in cs503?

could be answered "none", but there may be many different reasons that there are none. There might be no Dr. Lee and so there are no students of Dr. Lee. There might be a Dr. Lee but he may have no students. There might not be a class called cs503 and so there could be no students enroled in it. Finally, Dr. Lee may have students and there may be a class called cs503 but none of the students are enroled in it. The response "none" is ambiguous and would probably be misinterpreted by the user. The problem of analyzing the failure to find any students can be handled by removing part of the query and reissuing it (ALLEN87). Analyzing the cause for the failure involves determining which presuppositions of the question are wrong.

By not treating questions literally, the system can also become more cooperative. Questions such as "Can you list employees working in research" could be answered by listing the employees instead of by confirming that the system was capable of listing them. Sometimes yes/no questions require more than just a yes or no answer.

In this case the products must be sold at a price in a particular range. "No" answers the

Can I sell an ace tennis net for 69 dollars?  
No, the lowest price allowed is 72 dollars.

question literally but is not satisfactory because the system is not working with the user to solve the problem. These examples show it is important to provide a reason for a "no"

response.

Ambiguity is the reason for many of the problems that a natural language interface must deal with. In the previous section, the concern was to avoid ambiguity in the responses. Ambiguous responses can cause the user to develop misunderstandings. This section concerns how the system can deal with input that is ambiguous. This process involves detecting the ambiguity and interacting with the user to resolve it.

Ambiguity can arise when there is more than one choice for the interpretation of a word or phrase. In ROBOT such a problem occurs for:

Who lives in New York?

Does "New York" refer to New York state or New York city? Harris solves this problem by printing paraphrases of the interpretations and asking the user to select one. This is confirmation by paraphrase. The other approach to resolving the ambiguous reference is confirmation by effect.

The CHAMP system of LEHMAN89 is an adaptive user interface that is capable of learning the syntax as well as the meaning of statements. For CHAMP's confirmation by effect, when there is more than one interpretation of an input, the system will analyze the consequences of each alternative and ask the user to select an interpretation by presenting the potential consequences of each input. This means that the contents of the database can affect the selection of the intended meaning of an utterance. Confirmation by effect can

also be used to eliminate alternatives before even asking for a selection by the user. For the sentence

Which of the New York employees live in Buffalo?

ROBOT is able to determine whether "New York" refers to New York city or to New York state by issuing the queries corresponding to both interpretations. For the interpretation where "New York" is taken to be a city there is no result. ROBOT decides without asking the user that "New York" must refer to the state because that produces a result.

When using paraphrase to resolve ambiguity it is important to avoid providing information that is not needed to make a decision. For the ROBOT system, the request in Figure 2.6 is ambiguous.

What is the family status of the area managers that live in New York?

Does "New

**Figure 2.6 - Ambiguous Question**

York" refer to

the state or the city? ROBOT characterizes the two choices with the paraphrases shown in Figure 2.7.

**Choice 1:**

PRINT THE FAMILY STATUS, STATE AND CITY OF ANY EMPLOYEE WITH STATE=NY AND JOB=AREA MANAGER

**Choice 2:**

PRINT THE FAMILY STATUS, STATE, AND CITY OF ANY EMPLOYEE WITH CITY=NEW YORK AND JOB=AREA MANAGER

**Figure 2.7 - Paraphrases**



More information is provided than is needed to make the decision. A simple approach would be to provide enough information to distinguish the alternatives as shown in Figure 2.8.

<b>Choice 1:</b>  CITY = NEW YORK  <b>Choice 2:</b>  STATE = NY
---

**Figure 2.8** - Simple Choices

### **2.2.2.3 Robust with Respect to "Noise"**

Inevitably, a natural language interface will be forced to deal with inputs that contain errors or unforeseen forms of expressions. Spelling errors, unknown words, missing words, extra words, ellipsis and strange phrase orderings are all part of this problem. Simply rejecting the input without explanation or alternatives is not acceptable. Systems that cannot deal with simple forms of these problems will be seen as inflexible.

Spelling errors can be dealt with by searching for close matches in a list of words.

Semantic grammar and case frame grammar systems are able to offer good spelling checking because the semantic information can be used to reduce the size of the word list that must be checked. This reduction is possible because the semantic information is available during the parsing process to constrain the search.

#### 2.2.2.4 Handling of Discourse Phenomena

Another important issue for natural language interfaces is managing the dialogue with the user. The scope of the interface must exceed the individual sentence if the system is to be able to respond to the shortcuts people take when communicating. Anaphora is common during conversations. Anaphora is the use of reduced linguistic forms such as pronouns (he, they, it), demonstratives (that, this) or definite noun phrases for purposes of reference. People also employ ellipsis, or sentence fragments, to ease communication. This section briefly describes some dialogue phenomena.

##### 2.2.2.4.1 Ellipsis

Through the use of semantic information gained during the processing of previous sentences, case frame and semantic grammars are able to effectively deal with ellipsis. The

example

in

Figure 2.9

is taken

from

```

What is the length of the Constellation
> (LENGTH 1072 feet)
of the Nautilus
> (LENGTH 319 feet)
displacement
> (STANDARD-DISPLACEMENT 4040 tons)
length of the fastest American Nuclear sub
> (LENGTH 360 feet NAME LOS ANGELES SPEED 30.0 knots)

```

LIFER.

**Figure 2.9 - Ellipses**

LIFER is able to determine the correct interpretations by partially parsing the inputs to determine what type of phrases they are and then looking for phrases of the same type in the originating sentence. A substitution of the new phrase for the old is used to produce an

interpretation. The use of semantic information during parsing makes this approach feasible. Syntactic constraints are often not enough to correctly interpret ellipsis. Case frame grammars can also effectively deal with this type of ellipsis because of the availability of semantic information during the parsing process.

#### 2.2.2.4.2 Anaphora

Handling anaphora is as important as dealing with ellipsis, and as difficult a problem.

There are many different forms of anaphora. The dialogue of Figure 2.10 illustrates a few of these problems. Pronouns can be used to reference things

```
Who manages Smith?
> Ford
What is the manager's salary?
> 3100
Who else makes at least that much?
> Jones and King.
What do they make?
> Jones makes 3100 and King makes 5000.
```

**Figure 2.10 - Anaphora**

mentioned not only in the previous sentence but in the current sentence or in sentences before the previous one. As described in ALLEN87 (pp 334-365), the history list is an effective technique for solving this problem.

The history list is a list of objects mentioned in previous sentences in reverse chronological order. Semantic information can be associated with the objects. Upon encountering an anaphoric reference the history list can be searched for objects with compatible semantic information. Additional constraints can be derived from syntactic information as described in HOBBS78.

### 2.2.2.4.3 Vagueness

In KAPLAN84 vague statements are portrayed as a means employed by the user to reduce the size of statements as well as to provide the responder with fewer restrictions on what the response should be. The utterer of the statement expects that the user can make reasonable inferences and so a cooperative system must be able to respond to vagueness. Vagueness and ambiguity are similar in that the correct interpretation cannot be obtained from a literal reading. They are different because ambiguity arises when there is more than one interpretation immediately available and vagueness arises when the system must fill in unstated connections. Vagueness does not imply ambiguity.

Dealing with vagueness involves making inferences about additional unstated constraints. These constraints can be related to how the query is to be interpreted as well as to what response the user

expects. The

following example

<p><b>Q:</b> What is the quantity-on-hand of each size of radial tire? <b>R:</b> 125, 0, 500, 32, 40</p>
--

**Figure 2.11 - Vagueness**

(Figure 2.11) from KAPLAN84 illustrates how literal interpretations of vague questions is not satisfactory. A better response would include the sizes of the tires as well as the information that was explicitly asked for.

## 2.3 Learning Language

This section briefly describes how learning has been used by natural language systems to obtain information necessary to improve performance. Natural language systems can enhance their performance by learning a combination of syntax and semantics as well as lexical information. Problems that any learning system must deal with are inputs that contain errors and so should not be learned, learning with the fewest number of examples, being able to correct previous misunderstandings, and producing similar results if examples are ordered differently. The systems described here solve these problems with varying degrees of success.

### 2.3.1 Syntax

BERWICK85 is an example of a system which attempts to learn syntactic information only. The goal of this system was to explore how children can learn language. It is based on the Marcus parser [MARCUS80] and X-bar theory of grammars. The system learns X-bar grammar rules and transformation rules by being presented with a number of grammatical sentences. Learning arises from each input and the results of analysis immediately modify the existing grammar.

This system has a number of problems, some of which are due to the lack of semantic information. Words must have feature markers (+noun for example) in order for the system to learn properly. This information is not likely to be available to a child. The ordering of

input sentence also affects learning. If a question such as "Is Fred a postman" is presented first, the system would incorrectly learn that verbs come first in sentences. The system also has no way of correcting structures that were learned but should not have been, such as the previous example. The main advantage of the system is that it will learn quickly. This system illustrates that there is a trade off between learning quickly and learning only correct forms. If a system is learns too quickly then there must be some mechanism to correct mistakes. Without such a mechanism, the system will not perform effectively.

## **2.3.2 Semantics**

### **2.3.2.1 CHAMP**

LEHMAN89 presents a technique for developing a natural language interface that can learn syntax and simple semantics of utterances related to a table of data - an adaptive interface. Lehman identifies three types of user interfaces, adaptable, customizable and instructible. The difference lies in the method used to provide domain dependent information about the syntax and semantics of the grammar and lexicon. Customizable user interfaces as represented by INTELLECT (HARRIS84) and TEAM (MAP83) are configured by an expert using a special facility. Instructible user interfaces such as ASK, (THOMP85), require that the user provide customization using a simple command language. An adaptable user interface reconfigures itself during interaction with the user. No special command language is required. The system presented in this thesis is an example of an adaptive user interface which doesn't learn syntax but focuses on learning semantics.

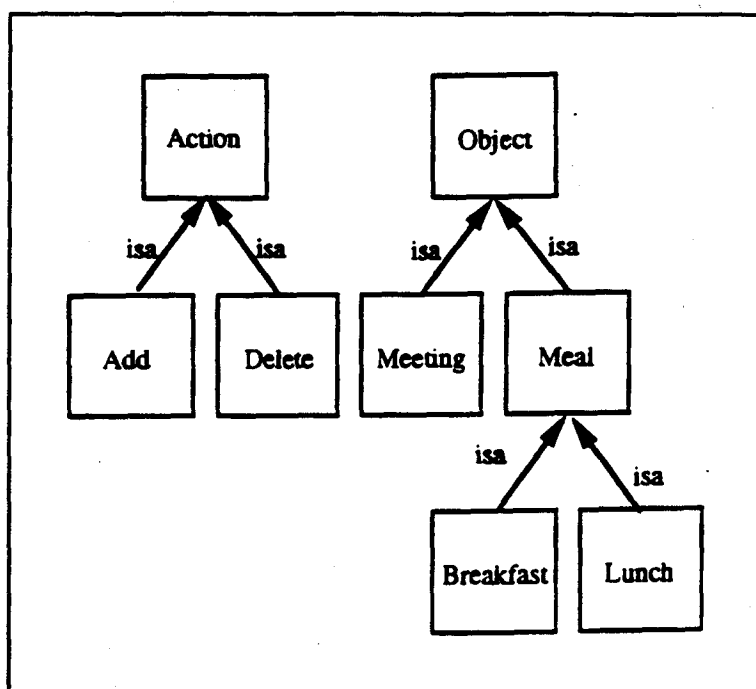
There are many advantages of adaptable user interfaces. The reasons for these advantages can be seen after exploring characteristics of users of the systems. These observations are derived from empirical research presented in (LEHMAN89). Users can be knowledgeable in command languages and database structure, or they can be relatively naive in terms of database management system technology. The former is a *sophisticated user* and the latter a *naive* one. These observations concern the naive users who would most benefit from natural language interfaces. The main observation is that users tended to limit not only the words but also the structure of their utterances. Forms of expression that are not ambiguous to a grammar customized to the user are ambiguous to a grammar common to all users. These observations lead to several reasons for preferring adaptive interfaces.

Most natural language interfaces have a single grammar that is to be used by everyone. Relative to a single user, this parser would be able to analyze sentences that the user is never going to utter. Thus parse time is spent trying to interpret utterances in a way that the user would never employ. Users have idiosyncratic forms of expression. Extending the parser to be able to handle each user's forms would result in a parser that ran too slowly. The parser would also see ambiguity where relative to an individual user's form of expression there is no ambiguity. The final problem with extending the parser is that every user would have to pay for the extra parse time needed to handle some users' strange forms of expression. Having an adaptive interface which is customized to each user eliminates these problems and provides a system that is more effective for each user.

CHAMP is the system that is used by Lehman to implement an adaptive parser. CHAMP is

a bottom up parser that selects the least deviant parse to learn from. Deviation is measured from the current grammar of the system in terms of the number of insertions, deletions, substitutions, or transpositions that must be performed to the utterance to allow it to be recognized by the current grammar. Recognizing an utterance results in a database action record. The database action record has four allowable functions, to add, delete, modify or print a row in a table of the database. The database action record approach is a form of procedural semantics.

CHAMP uses a combination semantic and case frame grammar. The grammar contains constituents to recognize expressions referring to time intervals, meetings and places, etc. These are domain dependent structures which the system updates as it learns. To support recognition CHAMP maintains a concept hierarchy.



**Figure 2.12 - Part of a Concept Hierarchy**

The concept hierarchy provides information that can be used to simplify the grammar and facilitate interpretation. The concept hierarchy is specified by an expert before system use. Figure 2.12 depicts part of a concept hierarchy. The boxes represent concepts and the arrows represent isa links.



In addition to the isa links each node of the concept hierarchy contains information that can be used to provide default values and constraints. In the sentence of Figure 2.13, the time or the place that the people will eat at is not specified. The concept hierarchy can be used to obtain additional information. In this case

Schedule a lunch with John on June 4.

**Figure 2.13**

the time is derived from the lunch concept and the place is derived from the meal concept.

The lexicon contains many domain dependent components as well. The words that are in the database appear in the lexicon and are put there manually. There are words that refer to concepts in the concept hierarchy such as arrival, meeting, and dinner in the lexicon. The definitions for these words identify them as marking slots in the frames or as referring to a concept in the concept hierarchy.

The end result of the parse is a database action record. The only actions allowed are to add, delete, modify or show a row in the database table. The system is not capable of performing any other action to the database nor can an utterance refer to more than one table at a time. The system never learns new meanings to sentences only new ways of referring to one of these actions or to one of the objects in the database. The new ways of referring can be synonyms of known words or new orderings of component phrases.

Several other researchers have developed systems to investigate how children learn language or how effective a particular approach is in learning language. CHAMP is the only such system which actually interfaces with a database. A comprehensive review of

these other systems can be found in LEHMAN89.

### 2.3.2.2 TEAM

TEAM is an example of a customizable user interface. For this approach, a database administrator (DBA) interacts with a menu driver interface in order to provide domain dependent information. This section briefly describes how the system gathers information.

TEAM learns by asking a DBA, structured questions. The actual learning process of TEAM is not as interesting as what the system is learning, however. TEAM contains domain independent knowledge of certain concepts such as numbers, features, and measurements. This information is stored in a hierarchy of objects called the sort hierarchy. The user extends this hierarchy through the process of adding new concepts to the knowledge base. Associated with each type of concept are a number of items. For an arithmetic field the information shown in Figure 2.14 is obtained.

```

Value Type - DATES MEASURES COUNTS
Are the units implicit? YES NO
Enter implicit unit - DOLLAR
Abbreviation for this unit?
Measure type of this unit -
TIME WEIGHT SPEED VOLUME LINEAR AREA WORTH OTHER
Minimum and Maximum numeric values - 0. 100000.
Positive adjectives - (HIGH PAID)
Negative adjectives - (LOW PAID)

```

**Figure 2.14 - Arithmetic Field Acquisition (MAP83)**

The main disadvantage of TEAM is the process that is used to derive domain dependent

knowledge. The DBA must first determine which words the users will want to use. Then the definition must be standardized. Standardization involves compromise and compromise implies settling for less than was wanted. After that, the words must be entered into the system along with additional information. Of course the users want new words added later. This means the DBA must start the process all over again. A better approach would allow users to independently configure the NLI on their own time.

## **2.4 Summary**

Many different approaches to providing natural language interfaces have been proposed. The more recent case frame-based approach which provides a means to combine syntactic and semantic information seems to be the most promising. During the course of development of natural language interfaces, a number of design principles were developed. One of the most important was portability, i.e. - the ease with which the natural language interface can be transferred to a new domain of discourse. Portability issues play a major role in this thesis. A number of researchers have investigated the advantages of using learning to enhance system performance. Recently a learning approach to configuring a natural language interface has been developed and experimentally evaluated. This thesis presents another aspect of the use of learning to enhance natural language interfaces. Here learning is used as a means of increasing portability of a natural language interface.

## **Chapter 3**

# **System Overview**

### **3.1 Objectives**

The goal of this system is to make it easier to transfer a natural language interface from one domain to another. This has been a goal (not the only one) of a number of other systems such as COOP (KAPLAN84), and TEAM (MAP83). For early natural language systems, an expert in setting up grammars and in databases was needed to transfer a system from one domain to another. Later work resulted in systems that could be transferred by a database expert. This thesis advances portability by providing a means that allow naive users to set up a natural language interface. In this case, a naive user is a user who does not necessarily have knowledge of the database structure or formal query languages.

### **3.2 Knowledge Base Components**

This section briefly describes the components used to represent knowledge and where they are derived from.

### 3.2.1 Domain Independent Knowledge

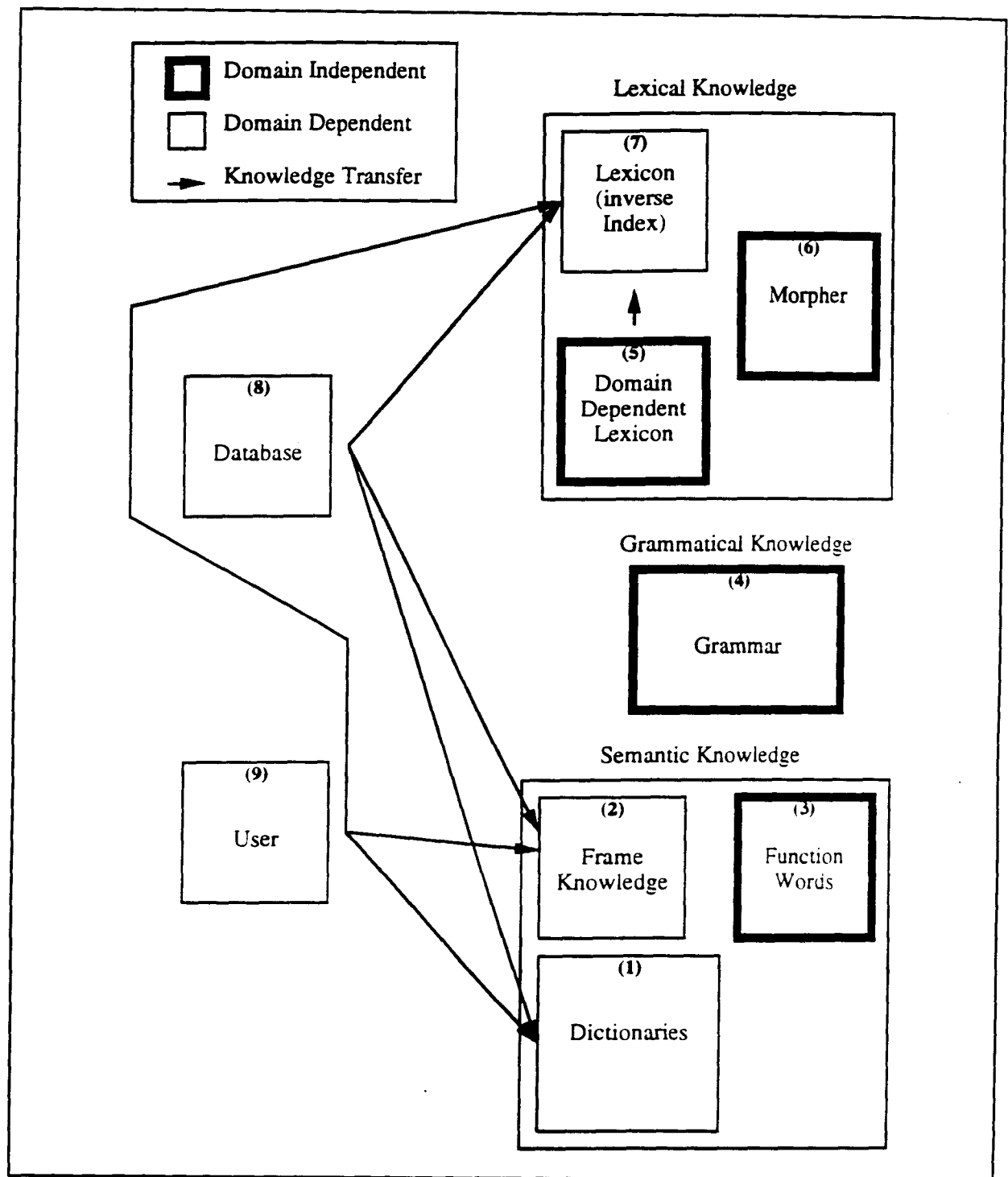


Figure 3.1 - Knowledge Source Diagram

Domain independent knowledge is knowledge that remains fixed during program execution. For this system, domain independent knowledge includes part of the lexicon, the grammar and part of the semantic knowledge. These are represented in Figure 3.1 by the boxes with thicker outlines (knowledge stores (3), (4), (5), and (6)). The domain independent part of the lexicon (knowledge store (5)) includes prepositions, auxiliary verbs, articles, and function words (average, total). The entire grammar (knowledge store (4)) and the morpher (knowledge store (6)) are domain independent. The grammar developed for the system presented in this thesis is listed in DCG form in appendix B. Some of the knowledge used to interpret sentences is also domain independent. This includes information on how to convert questions, queries involving function such as "average", and queries involving comparisons such as "more than", into SQL (knowledge store (3)).

### **3.2.2 Domain Dependent Knowledge**

Domain dependent knowledge is knowledge that depends on the particular database being used. In order to function properly with a new database, domain dependent knowledge for that database must be given to the system. For this system, parts of the lexicon and the semantic knowledge are domain dependent. These are marked in Figure 3.1 by boxes with outlines of lesser thickness (knowledge stores (1), (2) and (7)). There are two different type of words added to the lexicon (knowledge store (7)), words or phrases that are database elements and words referring to knowledge structures that the system constructs such as case frames or slots in case frames. The words that are database elements are derived

directly from the database prior to program execution. This is depicted as a transfer of knowledge (represented by the arrow) from the database (knowledge store (8)) to the lexicon (knowledge store (7)). Words that refer to case frames or case frame slots are added to the lexicon as the system learns them during interaction with the user. This is depicted in Figure 3.1 as a transfer from the user (knowledge store (9)) and the database (knowledge store (8)) to the lexicon (knowledge store (7)) and the dictionaries (knowledge store (1)).

The semantic information that the system learns mainly concerns the frame knowledge (knowledge store (2)). There are two type of frames in this system, case frames for verbs and object frames for objects. An object frame is used to keep information about case fillers. Object frames keep track of object names, and attributes. Case frames keep track of event dates, and cases for the verbs. Associated with the frame information is a number of dictionaries (knowledge store (1)) that map various words and types to frames. These are also maintained by the system and are part of the domain dependent information. As depicted in Figure 3.1, the frame knowledge (knowledge store (2)) and the dictionaries (knowledge store (1)) are derived from both the user (knowledge store (9)) and the database (knowledge store (8)).

### **3.3 Processing Flow of Control**

This section describes the flow of control through the system which is depicted in Figure 3.2. The processing starts with reading a sentence which is then tokenized, parsed

and then interpreted. After interpretation, the system either updates its knowledge base for a declarative statement or generates SQL and prints the results for a question. The following section describe each step in more detail.



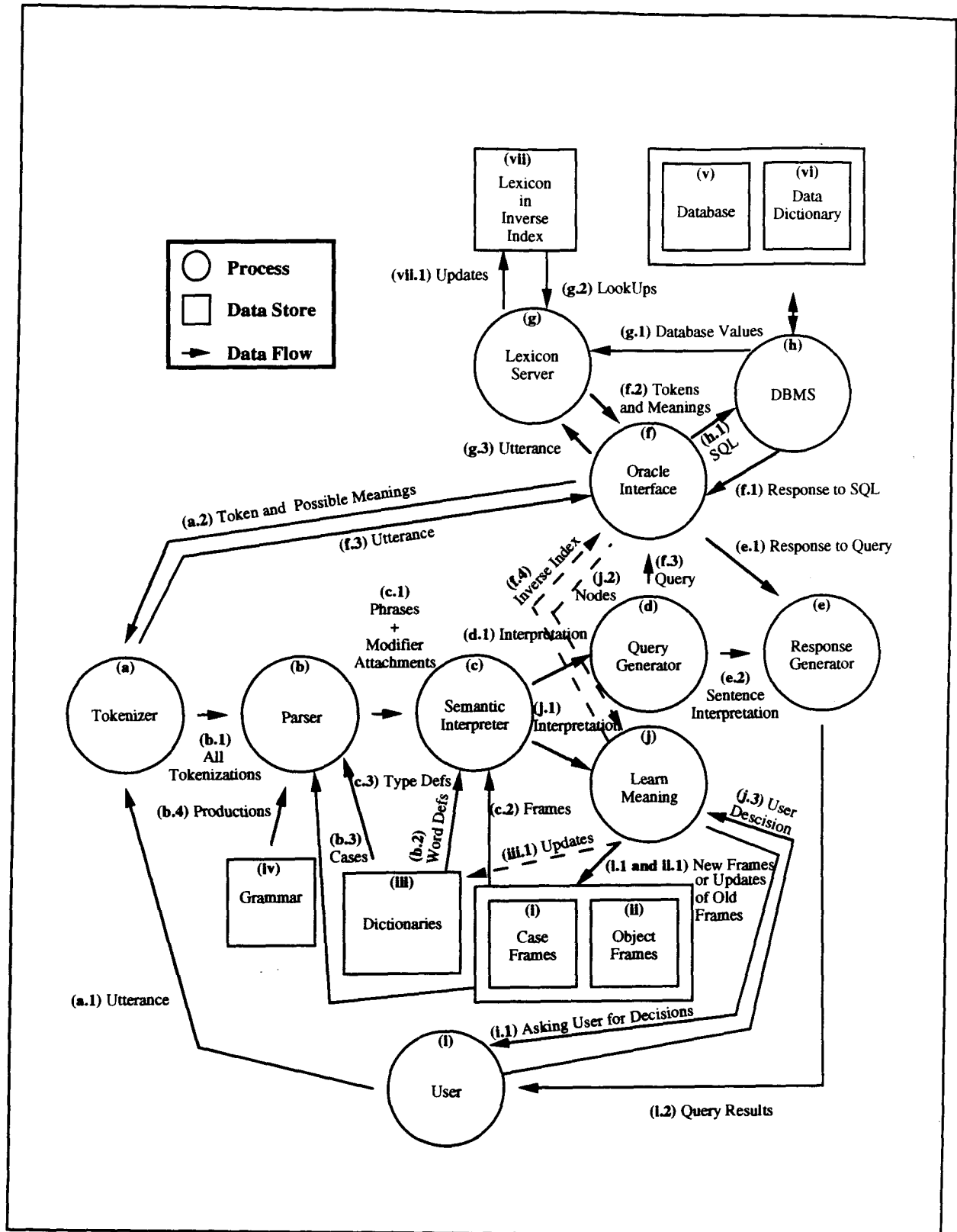


Figure 3.2 - Data Flow Diagram

### 3.3.1 Tokenizing

Tokenization occurs at process (a) of Figure 3.2 where the utterance (flow (a.1)) arrives from the user (process (i)). The tokenizer (process (a)) passes the utterance (flow (f.3)) to the ORACLE interface (process (f)) which in turn passes the utterance to the lexicon server (process (g)) along flow (g.3). The lexicon server uses the inverted index (data store (vii)) to break the utterance up into tokens (words or phrases). Associated with each token is possible meanings which may indicate that the token is in a fixed word class (preposition, article, etc), that the token refers to a case frame or a slot in a case frame, or that the token appears in the database. There may be more than one way to break the sentence into tokens. After the lexicon server (process (g)) determines all tokenizations, the results are passed back to the tokenizer (process (a)) along flows (f.2) and (a.2) through the ORACLE interface (process (f)).

### 3.3.2 Parsing

The parser (process (b)) receives the tokenizations from the tokenizer (process (a)) along flow (b.1). The parser then uses the grammatical information obtained from the grammar (data store (iv)) along data flow (b.4), to analyze the sentence. The grammar is in the form of a definite clause grammar (DCG) as described in PERWARR80. In order to avoid producing non-sense parses, the parser accesses semantic information. The information from the dictionaries (data store (iii)) is used to determine which case frames are associated with which words. The case frames (data store (i) and (ii)) are accessed along data flow

(b.3) to help resolve  
 attachments of cases

Who earns over the average salary of a person working in research.
--

**Figure 3.3 - Case Attachment**

to verbs. For the

sentence of Figure 3.3, this information is used to help determine whether "in research" is a case of "working" or of "earns".

### **3.3.3 Interpretation**

The semantic interpreter (process c) receives the partial parses of the utterance from the parser (process b) along data flow c.1. Each parse contains information about noun phrases and verb phrases mentioned and constraints on modifier attachments. The semantic interpreter (process c) accesses the dictionaries (data store iii) along dataflow c.3, and the frames (data stores i and ii) along data flow c.2. This information is used to produce an interpretation of the utterance. During this process, the system will detect that some of the parses are unintelligible and discard them. At this point there are two major alternatives for the processing flow of control depending on whether the sentence is declarative or interrogative.

#### **3.3.3.1 Declarative Sentence**

This part of the processing is described in detail in chapter 5. Declarative sentences are used to define new concepts. The 'learn meaning' process (process (j)) is invoked after the semantic interpreter (process (c)), if the utterance is declarative. The 'learn meaning'

process (process (j)) receives the interpretation from the semantic interpreter (process (c)) along data flow (j.1). At this point, interaction occurs with the user to determine a correct interpretation for the meaning of the concept that the user is describing in the sentence. Interaction occurs with the user (process (i)) along data flows (i.1) and (j.3). During the process of interacting with the user, the database and the inverted index are accessed along data flows (f.4) and (j.2). This interaction occurs as long as the user has not helped to determine a meaning for the utterance. At some point, if the user is satisfied with the interpretation, the system updates the dictionaries (data store (iii)) along data flow (iii.1) and the frames (data store (i) and (ii)) along data flow (i.1) and (ii.1). Processing of the utterance is now complete.

### **3.3.3.2 Questions**

This part of the processing is described in detail in chapter 6. This processing stream is invoked after the semantic interpreter (process (c)) if the utterance is a questions. The query generator (process (d)) receives the sentence interpretation from the semantic interpreter (process (c)) along data flow (d.1). The interpretation is converted into an SQL query which is passed to the ORACLE interface (process (f)) along data flow (f.3). The ORACLE interface (process (f)) then passes the SQL query to the DBMS (process (h)) for processing along data flow (h.1). The results of the SQL query are returned from the DBMS (process (h)) along data flow (f.1) to the ORACLE interface (process (f)) which passes them on to the response generator (process (e)) along data flow (e.1). The response generator (process (e)) also receives the sentence interpretation from the query generator

(process (d)) along data flow (e.2). The response generator presents the results to the user (process (i)) along data flow (i.2). The response generator for this system merely prints out the data returned by the DBMS rather than producing a more elaborate display. This ends the processing of the utterance.

### **3.4 Implementation Details**

This system currently accesses the ORACLE RDBMS only. Because the system is written mainly in Prolog and ORACLE does not support access with Prolog, a C server was set up to access ORACLE (process (f) of Figure 3.2). The C server makes use of ORACLE's PRO\*C compiler that contains extensions for accessing the database. The lexicon server (process (g)) (an inverted index - see section 4.1.3.1) was set up in C++ as a separate process accessed only through the ORACLE interface (process (f)). The reason for a separate process was that the C++ compiler used BSD Unix and the ORACLE PRO\*C libraries required UNIX System V libraries and so could not be compiled together. Due to licensing agreements the Prolog interpreter ran on one platform and the Oracle server ran on another so communication between the Prolog process and the C server occurred through a TCP package in Prolog and with TCP in C.

### **3.5 Examples**

What follows is a brief example of how the system works. A more complete session can be

found in appendix E. For this system, the user is a person who has little or no knowledge of DBMS. This means little or no knowledge of relational database theory, formal query languages and the structure of the database that he or she will be using. It is assumed that the user knows about the data in the database. For example, if the database was an employee database the user must know about some particular employees (perhaps themselves). For a movie database, the user must know about some particular movies. What the user lacks in knowledge can be obtained from more traditional sources such as reports, data collection forms, or other people.

**smith works for ford.**

EMP0.ENAME	EMP1.ENAME
-----	-----
ALLEN	BLAKE
FORD	JONES
SCOTT	JONES
WARD	BLAKE

Do these examples seem reasonable?

|: yes.

To configure the system for a new database, the user (in **bold**) enters sentences that describe data in the database. By using these sentences and accessing the database the system can determine the meaning of nouns and verbs. (Users tend to know a lot about some of the data in the database).

| : **smith works in research.**

EMP0.ENAME	DEPT0.DNAME
-----	-----
CLARK	ACCOUNTING
KING	ACCOUNTING
MILLER	ACCOUNTING
SMITH	RESEARCH

Examples are presented to provide feedback

Do these examples seem reasonable?

| : **yes.**

| : **smith is managed by ford.**

EMP0.ENAME	EMP1.ENAME
-----	-----
ALLEN	BLAKE
FORD	JONES
SCOTT	JONES
WARD	BLAKE

Do these examples seem reasonable?

| : **yes.**

| : **smith's salary is 800.**

EMP0.ENAME	EMP0.SAL
-----	-----
ALLEN	1600
JONES	2975
SMITH	800
WARD	1250

Do these examples seem reasonable?

| : **yes.**

| : **the manager of smith is ford.**

EMP0.ENAME	EMP1.ENAME
-----	-----
ALLEN	BLAKE
FORD	JONES
SCOTT	JONES
WARD	BLAKE

Do these examples seem reasonable?

| : **yes.**

| : the salary of smith is 800.

EMP0.ENAME	EMP0.SAL
-----	-----
ALLEN	1600
JONES	2975
SMITH	800
WARD	1250

Do these examples seem reasonable?

| : yes.

| : smith earns 800.

EMP0.ENAME	EMP0.SAL
-----	-----
ALLEN	1600
JONES	2975
SMITH	800
WARD	1250

Do these examples seem reasonable?

| : yes.

| : what is smith's salary.

EMP0.SAL
-----
800

That's it. System configuration is complete (for now). Questions can now be asked about the database. Note that questions can be asked as soon as the user has defined the words that will appear in the question. After asking some questions, more words can be defined if desired.

First an easy question to get warmed up.

| : which person's manager's salary is 2975.

EMP1.ENAME
-----
SCOTT
FORD

| : which person's salary is over ford's salary.

EMP1.ENAME
-----
KING

Possessives and comparison.



| : who earns over an employee's average salary.

EMP0.ENAME

-----

JONES  
BLAKE  
CLARK  
SCOTT  
KING  
FORD

Functions such as average, total,  
highest, and lowest are provided.

| : who earns over an employee working in research's average salary.

EMP0.ENAME

-----

JONES  
BLAKE  
CLARK  
SCOTT  
KING  
FORD

| : what is the lowest salary of the people who earn over the average salary of an employee.

MIN(EMP0.SAL)

-----

2450

| : how many people earn over the average salary of an employee.

COUNT(EMP0.ENAME)

-----

6

| : can you list the name manager and salary of the employees working in research.

EMP0.ENAME	EMP0.SAL	EMP1.ENAME
-----	-----	-----
SCOTT	3000	JONES
FORD	3000	JONES
SMITH	800	FORD
JONES	2975	KING
ADAMS	1100	SCOTT

|: **tkb sport shop paid 58 for ace tennis net.**

2 Interpretations. They Are:

Option #1

"58" refers to ACTUALPRICE in the ITEM table

Option #2

"58" refers to ITEMTOT in the ITEM table

Let's teach the system a word with a complicated meaning. This one has ambiguity and requires 4 table joins. The user is presented with a choice of two meanings.

You may select an option as a correct interpretation, ask for more options, or decide to stop this definition

|: **can you show me some more options.**

[moreOptions]

3 Interpretations (1 are new). They Are:

Option #1

"58" refers to ACTUALPRICE in the ITEM table

Option #2

"58" refers to ITEMTOT in the ITEM table

Option #3 (New)

"58" refers to STDPRICE in the PRICE table

Asking for more options gets the user an additional interpretation which is less likely to be the correct one.

You may select an option as a correct interpretation, ask for more options, or decide to stop this definition

|: **more.**

[moreOptions]

There are no more alternative interpretations. All alternatives have been listed.<sup>3</sup> Interpretations. They Are:

Option #1

"58" refers to ACTUALPRICE in the ITEM table

Option #2

"58" refers to ITEMTOT in the ITEM table

Option #3

"58" refers to STDPRICE in the PRICE table

There are no more options so the user must select an interpretation or give up.

You may select an option as a correct interpretation, ask for more options, or decide to stop this definition

|: **i like option number 1.**

[pickAnOption(1)]

| : **who has paid under 58 for ace tennis net.**

CUSTOMER0.NAME

-----  
JOCKSPORTS

Let's try a couple questions.

| : **what did jocksports pay for ace tennis net.**

ITEM0.ACTUALPRICE

-----  
50

I added another database about **movies**. Let's try using it for a while.

| : **steven spielberg directed jurassic park.**

MVDIR0.DIRNAME

MVMOVIE0.MVNAME

-----  
PENNY MARSHALL

-----  
LEAGUE OF THEIR OWN

SPIKE LEE

MALCOLM X

STEVEN SPIELBERG

JURASSIC PARK

Do these examples seem reasonable?

| : **yes.**

| : **who directed malcolm x?**

MVDIR0.DIRNAME

-----  
SPIKE LEE

| : **league of their own starred tom hanks.**

MVMOVIE0.MVNAME

MVSTAR0.STNAME

-----  
JURASSIC PARK

-----  
JEFF GOLDBLOOM

LEAGUE OF THEIR OWN

GEENA DAVIS

LEAGUE OF THEIR OWN

MADONNA

LEAGUE OF THEIR OWN

TOM HANKS

Do these examples seem reasonable?

| : **yes.**

| : who did jurassic park star?

MVSTAR0.STNAME

-----  
JEFF GOLDBLOOM

| : steven spielberg directed jurassic park.

MVDIR0.DIRNAME	MVMOVIE0.MVNAME
-----	-----
PENNY MARSHALL	LEAGUE OF THEIR OWN
SPIKE LEE	MALCOLM X
STEVEN SPIELBERG	JURASSIC PARK

Do these examples seem reasonable?

| : yes.

| : steven spielberg directed jeff goldbloom.

MVDIR0.DIRNAME	MVSTAR0.STNAME
-----	-----
PENNY MARSHALL	GEENA DAVIS
PENNY MARSHALL	MADONNA
PENNY MARSHALL	TOM HANKS
STEVEN SPIELBERG	JEFF GOLDBLOOM

Do these examples seem reasonable?

| : yes.

| : who directed malcolm x.

MVDIR0.DIRNAME

-----

SPIKE LEE

| : who directed tom hanks.

MVDIR0.DIRNAME

-----

PENNY MARSHALL

Now we'll try defining two different senses for the verb direct. Let's start with director directing movie. Notice that we defined this earlier. The system can handle such mistakes.

Now we'll define the other sense of the verb direct - director directing star.

Now a couple of question to check things out.

| : **smith works for ford.**

EMP0.ENAME	EMP1.ENAME
-----	-----
ALLEN	BLAKE
FORD	JONES
SCOTT	JONES
WARD	BLAKE

Let's try two different senses of the verb works. One will relate to the employee database and one will relate to the movie database. Will start with employee works for manager.

Do these examples seem reasonable?

| : **yes.**

| : **tom hanks works for penny marshall.**

MVSTAR0.STNAME	MVDIR0.DIRNAME
-----	-----
GEENA DAVIS	PENNY MARSHALL
JEFF GOLDBLOOM	STEVEN SPIELBERG
MADONNA	PENNY MARSHALL
TOM HANKS	PENNY MARSHALL

Now we'll define the 'movie star works for director' sense.

Do these examples seem reasonable?

| : **yes.**

| : **who works for king.**

A couple of test questions.

EMP0.ENAME
-----
JONES
CLARK
BLAKE

| : **who works for steven spielberg.**

MVSTAR0.STNAME
-----
JEFF GOLDBLOOM

## 3.6 Summary

The main objective of this system is to provide a means to increase the portability of natural language interfaces. This has been facilitated by separating the components of the system based upon whether or not they are domain dependent (change with the domain of discourse) or domain independent (constant over all domains of discourse). An overview of the control structures of the system has been presented to provide the reader with an understanding of how and what information is used during the analysis process. The chapter concluded with a brief session with a program implemented to demonstrate the configuration process.

## **Chapter 4**

# **Linguistic Knowledge**

This chapter discusses the components of the system that contain linguistic knowledge - the lexicon and the grammar. The purpose is to provide the reader with a feel for the extent of the coverage of the grammar.

### **4.1 Lexicon**

In order to analyze utterances, knowledge about words is needed. This knowledge does not only comprise knowledge about the possible meanings of words. Words also have structure which encodes meaning. For example, 'cow' refers to a single cow and 'cows' refers to more than one cow. Morphology is the study of the internal structure of words. In order to avoid storing many different forms of a word, morphological analysis is used to extract additional meaning from the structure of words. Words obviously have a function in an utterance. The function of the word affects how it can be used. For example, words used as a conjunction cannot appear at the end of a sentence. "I bought some eggs, cheese and", is not a well formed sentence. In order to effectively analyze utterances, knowledge about the

function of words is also necessary. A more detailed discussion of the function of words and morphology can be found in WINOGRAD83.

The lexicon is used to store information about the function and meaning of words. The lexicon is a dictionary which associates meanings with each word. In most cases, the root form of a word is the only form stored in the lexicon. Exceptions are made for words with irregular forms which the morpher cannot analyze. The irregular forms are stored in the lexicon in addition to the root forms.

The lexicon contains entries that are derived from many sources. Some sources are domain dependent and some are not. This section considers all of the information sources needed to break the utterance into tokens and assign interpretations to those tokens. Tokens, in this case, may be individual words or complex phrases.

### **4.1.1 Word Classes**

The word classes used by the system are based upon common functional categories. For the purposes of this discussion, the domain independent classes will be presented first, followed by the domain dependent classes.

#### **4.1.1.1 Domain Independent (Predefined)**

Domain independent words are defined before the system is used. Figure 4.1, contains the



word classes used by this system. The lexicon marks words as belonging to the classes listed in this table where appropriate.

Class	Examples	Notes
Prepositions	of, out of	<ul style="list-style-type: none"> <li>used as markers for cases of the case frames</li> </ul>
Auxiliary Verbs	had, been	<ul style="list-style-type: none"> <li>includes tense, person, number and root verb</li> </ul>
Irregular Verbs	seek/sought	<ul style="list-style-type: none"> <li>includes irregular form and root form</li> </ul>
Determiners	the, a, which	<ul style="list-style-type: none"> <li>marked as definite or indefinite</li> <li>marked as question word</li> </ul>
Relative Clause Markers	that, which, who	<ul style="list-style-type: none"> <li>can mark start of relative clauses</li> </ul>
Function Word Modifiers	average, total	<ul style="list-style-type: none"> <li>includes SQL function name</li> </ul>

**Figure 4.1 - Word Classes**

#### **4.1.1.2 Domain Dependent**

Verbs and nouns comprise a major part of the domain dependent part of the lexicon. Verbs are learned during interaction with the user. They are defined in terms of case frames which describe the meaning of the verb. Nouns that refer to frames or case frame slots such as "employee" or "salary" are also learned during interaction with the user. After they are discovered by the system, the morphological analyzer produces a root form which is then added to the lexicon.

### 4.1.2 Morphing

Morphology is the study of the structure of words and how that structure is used to affect the meaning of words. Structure changes that affect word features, such as changing 'cow' (singular) to 'cows' (plural), are called *inflections*. Structure changes that affect the meaning of words, such as changing 'logical' to 'illogical', are called *derivations*. The morpher, which is based upon an algorithm presented in WINOGRAD72 on page 72, can analyze a large number of morphological structures but the system itself will only interpret a limited number of features marked by these structures (inflections). The system will not interpret derivations. The morpher is employed during tokenization of the sentence. For verbs, it is used to determine the verb tense (the difference between kill and killed). The code for the morpher is listed in the appendix.

### 4.1.3 Implementation of the Lexicon

This section describes the data structures used to implement the lexicon. The first section describes the data structure used to organize the lexicon, an inverse index, and the second section describes the data structure used to implement the inverted index.

#### 4.1.3.1 Inverted index

Figure 4.2 depicts a file containing information about the parts produced by a fictitious company. The top row of the table contains the names of each column. The address column contains a physical address of each row. This column is not an explicit part of the table. It

Address	P#	PNAME	COLOR	PRODUCER
100	P1	Widget	Red	Acme
110	P2	Gizmo	Blue	Acme
120	P3	Gadgit	Red	Parts Inc.
210	P4	Thing	Blue	Acme

**Figure 4.2 - Part Table**

is shown to be referenced by the example. The P# column contains the number of each part produced. The names of the parts are in the PNAME column. The remaining two columns provide information about each part. The COLOR column contains the color of the parts and the PRODUCER column contains the name of the company that produces the part.

Producer	Address	Address	Address	...
Acme	100	110	210	
Parts Inc.	120			

**Figure 4.3 - Inverted Index for the Producer Column**

An inverted index is constructed on a particular column of a table of data. Typically not all of the columns are indexed due to the cost of setting up an inverted index. For this example, an inverted index on the PRODUCER column will be described. The inverted index is depicted in Figure 4.3. For each distinct value in the PRODUCER column of the table of Figure 4.2, there is a row in the inverted index. In this case, there are only two values, "Acme" and "Parts Inc." in the inverted index. The inverted index associates row addresses with each value. The addresses are of rows that have the same value for the column that the inverted index is set up on. So for the 'Acme' row in the inverted index, the address of all the rows in the parts table that contain 'Acme' in the PRODUCER column are listed. These are rows 100, 110, and 210. This example illustrate that the

number of addresses associated with each value in the inverted index varies.

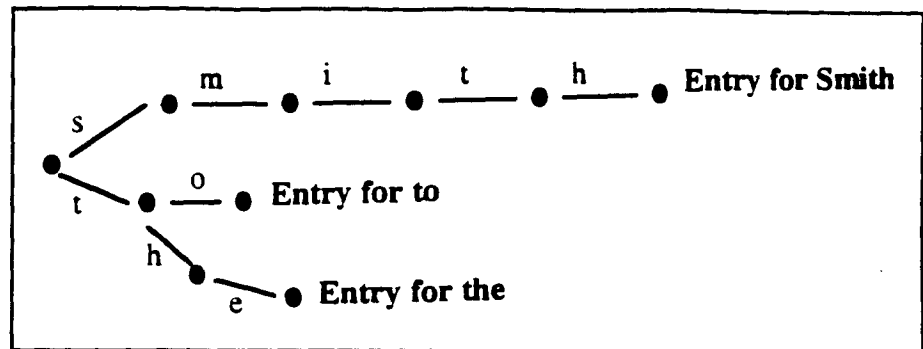
The main reason for using an inverted index is to decrease processing time. For example, if a list of red parts produced by Acme was desired and there was no inverted index, then a linear search must be performed on the parts table. However by using the inverted index, all the rows containing parts produced by Acme can be quickly accessed and then just these rows can be searched linearly for red parts. Results could be even faster if the COLOR column also was inverted. Then to find red parts produced by ACME, the list of red parts and the list of parts produced by ACME could be intersected. The inverted index is a means to produce results faster. A more detailed description of inverted indexing can be found in BRADLEY82.

#### **4.1.3.2 Implementing the Inverted index**

The lexicon is an inverted index implemented using a TRIE data structure as described in SEDGE90. It is constructed by first loading the predefined words of the domain independent part of the lexicon (knowledge store 5 of Figure 3.1) and then accessing the database (knowledge store 8) of the user to load the domain dependent values. All columns of the database are added to the inverted index. During the interaction with the user new nouns and verbs are added to the index.

A TRIE is short for reTRIEval tree. The TRIE is an m-degree tree. Each record stored in the tree has a key with a number of levels. In this case each level is a character. At a given

node there is one  
branch out for each  
character value at  
the current level in  
the key (see



**Figure 4.4 - Part of a TRIE**

Figure 4.4). In other

words, each edge corresponds to a character. Techniques were employed to compress the TRIE. Due to compression, the TRIE for the test database had a depth of 7 although the longest key in the TRIE was over 30 characters in length.

The TRIE has an advantage over hashing for this application. When the sentences are tokenized prefixes of characters are pulled off and treated as a single token. There might be more than one way to do this as

shown in Figure 4.5. For the first case "the", "man" and "shop" are treated as individual token in this domain. "The

Input:	The man shop is ....
Output:	(The) (man) (shop) (is) ... (The man shop) (is) ...

**Figure 4.5 - Different Tokens**

man shop" is also a token, perhaps a store in a customer database. A TRIE allows the system to make use of the current position in the TRIE after processing the first alternative, in order to process output two. Under a hash table approach, the prefixes would have to be picked off individually and have separate hash values computed. There is no reuse of computation. For other flavours of trees the prefixes would also have to be picked off and searched for individually. The TRIE is able to reuse computation where the others do not.

## 4.2 Grammar

Effective natural language systems must analyze the structure of sentences. A grammar is a definition of the structure of sentences that can be analyzed. There are a number of syntactic phenomena which a grammar must handle in order to be acceptable. The following sections outline which structures are in the coverage of the grammar of this system and the means used to represent and apply the grammar.

### 4.2.1 Coverage

#### 4.2.1.1 Passives

The subject and objects of a case frame are marked by position in a sentence. This position can be affected by the form of the sentence. If it is passive, then the syntactic subject, if it appears, must follow the verb and an object must precede it. This is the distinction between the active and passive form of a sentence. Figure 4.6 lists some examples.

Active Sentence	Passive Sentence
Sally took the pen	The pen was taken by Sally
The seals will raise the pups	The pups will be raised by the seals
I closed the door	The door was closed

**Figure 4.6 - Sentence Form**

#### 4.2.1.2 Relative Clauses

Relative clauses are a simple means for qualifying referents in a statement. They are marked by function words such as *that*, *who* or *which*. A reduced relative clause is a relative clause that is not marked by one of those words. In Figure 4.7, examples from ALLEN87<sup>3</sup>, the relative clauses are underlined.

<p>The man <u>who hit Mary with a book</u> has disappeared</p> <p>The man <u>who(m) Mary hit with a book</u> has disappeared.</p> <p>The man <u>hitting Mary with a book</u> is angry.</p> <p>The man <u>Mary is hitting with a book</u> is angry.</p> <p>The man <u>Mary hit John with</u> has disappeared.</p> <p>The man <u>whose book was used to hit John</u> has disappeared.</p> <p>The man <u>with whom Mary disappeared</u> wore a red hat.</p>
--

**Figure 4.7 - Relative Clauses**

#### 4.2.1.3 Wh-Question

Wh-questions are questions in which the wh-phrase has been moved to the start of the sentence such as

<p>Who does Smith manage.</p> <p>Who(m) does smith work for.</p> <p>For who(m) does smith work.</p>
---

The parser will handle only the cases where the phrase moved to the start of the sentence is not marked (by a preposition). It will not accept the third example. This form of expression

---

<sup>3</sup> see page 136

is described in detail in ALLEN87.

#### 4.2.1.4 Possessives

Possessive noun phrases are marked with ' or 's after the final word of the phrase. This system will accept multi-word possessive noun phrases with embedded relative clauses. It will also accept recursive embeddings of possessive markers. It will not recognise 'whose' as a possessive marker. Figure 4.8 lists some examples. The interpretation of the possessive marker (described in section 4.2.1.7), is limited.

Who is Smith's manager? What is the person working in research's average salary? What is Smith's manager's salary?
--

**Figure 4.8 - Possessives**

#### 4.2.1.5 Imperatives

Imperatives are sentences in which the subject is the hearer and so is not stated explicitly in the sentence. The parser will accept all imperatives although the system will only interpret imperatives whose main verb is 'print', 'list', or 'show'. Figure 4.9 contains some examples of imperative sentences that the system will accept.

List employees working in research for Ford. Print the name, manager and salary of employees working in research.
--

**Figure 4.9 - Imperative Sentences**



#### 4.2.1.6 Noise Phrases

Noise phrases are phrases that can be ignored when interpreting the sentence because they cannot be represented in the system's interpretation. The grammar will accept only some polite sentence introductions as are shown in Figure 4.10. Although they are accepted, they will not be interpreted.

Would you please ... Can you ... I would like you to ... Please ...
--

**Figure 4.10 - Noise Phrases**

#### 4.2.1.7 Semantic Grammar Based-Phrases

Another aspect of the grammar is the use of productions that are defined in terms of the semantic categories of phrases. This approach was only briefly explored and would benefit greatly from more work.

The system keeps track of frames that represent objects. The object frames have cases for object names, and attributes. In

this way, these objects are similar

to the objects represented in

object-oriented languages although

the objects of this system have no

Case Class	Case	Filler
Names	Last Name	EMPLOYEE.NAME
Attributes	Salary	EMPLOYEE.SAL
	Commission	EMPLOYEE.COMM

**Figure 4.11 - Example Employee Frame**

methods. Figure 4.11 shows an example of a frame for employee information. This frame

contains one name for the employee, the employee's last name, and it can be found in the NAME column of the EMPLOYEE table. There are two attributes of the employee represented in the frame, the employee's salary which is found in the SAL column of the EMPLOYEE table and the employee's commission which is found in the COMM column of the EMPLOYEE table. The object frames also contain information that ties the individual cases together in terms of the database (described in detail in chapter 5).

In order to use these object frames, special productions were set up to recognize references to these categories. The productions are defined in terms of phrases that represent references to attributes and objects as opposed to noun phrases and so in this sense this part of the grammar can be regarded as a semantic grammar. The following DCG productions (Figure 4.12) which are simplified for illustration purposes recognize phrases referring to objects. The actual productions which are integrated with the syntactic analysis for efficiency, are listed in appendix B.

```

AttributeReference --> Determiner, AttributeWord, "OF", ObjectWord
                       | PossessiveObjectWord AttributeWord.
AttributeList --> Attribute, ("", "!", | []), AttributeList1, "OF", ObjectWord, !
                  | [].
AttributeList1 --> "AND", Attribute, !
                  | Attribute, ("", "!", | []), AttributeList1
                  | [].
Sentence -->
AttributeReference BEVerbPhrase, AttributeValue
|
PrintVerbPhrase, AttributeList.

```

**Figure 4.12** - Semantic Grammar for Recognizing Object and Attributes

The `AttributeReference` production recognizes references to the attributes of objects such as, "the salary of Smith", or "Smith's salary". In both these examples, "Smith" is the object and salary is the attributes. The `AttributeList` production recognizes a list of attributes of an object such as "the name, age and salary of Smith". For this example, "Smith" is the object and "name", "age" and "salary" are attributes of Smith. The last production `Sentence` recognizes the use of attribute phrases in sentences such as "Smith's salary is 800", or "Print the name, salary, and department of Smith". The former sentence is used to define salary and the latter is used to get a list of employee information.

### **4.2.2 Representation**

The previous section described the grammar of the system. In order to be employed, the grammar must be encoded in a computational mechanism. For this system a definite clause grammar (DCG) formalism is used. A detailed discussion of DCG can be found in PERWARR80. Appendix B contains a listing of the definite clause grammar of this system.

## **4.3 Summary**

Linguistic knowledge consists of knowledge about words and knowledge about phrases. In addition to being associated with a concept, the structure of a word can contain meaning. A lexicon is used to associate words with concepts and morphological knowledge is used to

analyze the structure of words to derive additional meaning. The lexicon is implemented using an inverse index which provides a number of advantages over other techniques. A grammar is used to contain knowledge about phrases. There are a number of different type of phrases that the grammar must be able to recognize in order to function effectively. The range of phrases that can be recognized by a grammar is called its coverage. The grammar of this system is implemented using the definite clause grammar notation.

## **Chapter 5**

# **Becoming Portable Through Learning**

This section describes the main knowledge representation structure of this system and how it is constructed. A brief introduction is given in order to help the reader understand where this structure fits into the overall system.

### **5.1 Tokenization<sup>4</sup>**

After a sentence is read in, the system tokenizes it. The result is a sequence of words and phrases<sup>5</sup> referring to case frames, unknown words and words and phrases that appear in the database. Some lexical information is produced as well. After tokenization, parsing occurs.

### **5.2 Parsing<sup>4</sup>**

The parser produces a list of phrases. For each verb mentioned, there is a list of the cases for that verb, and for each object there is a list of the attributes of that object. The table of

---

<sup>4</sup> - see Chapter 6 for a detailed example

<sup>5</sup> - such as 'The Body Shop' which appears in the database as a store name

Figure 5.1 illustrates this information for the following sentence

Which<sub>1</sub> man<sub>2</sub> working<sub>3</sub> in<sub>4</sub> research<sub>5</sub> works<sub>6</sub> for<sub>7</sub> under<sub>8</sub> Ford's<sub>9</sub> salary<sub>10</sub>?

Frame Marker	Case	
	Marker	Filler
WORK <sub>3</sub>	Syntactic Subject	MAN <sub>2</sub> (question)
	IN <sub>4</sub>	RESEARCH <sub>5</sub>
WORK <sub>6</sub>	Syntactic Subject	MAN <sub>2</sub>
	FOR <sub>7</sub>	RANGE OVER SALARY <sub>10</sub>
FORD <sub>9</sub> (marked by type)	Attribute	Salary <sub>10</sub>

**Figure 5.1 - Cases and Fillers for Sample Sentence**

There are two case frame markers shown in the table, WORK<sub>3</sub> and WORK<sub>6</sub>. These correspond to the different occurrences of "work" in the example sentence. For each of these case frame markers there are two cases present which are shown in the table. The table also contains one object frame marker, FORD<sub>9</sub>. Type information that is known about "Ford" is used to determine which object frame should be used for interpretation. Only one case is associated with "Ford", the salary attribute. The parse also contains other information. Associated with each noun phrase is determiner information if there is any. Each verb phrase (case frame marker) has tense, number, auxiliary verbs and main verb information.

## 5.3 Learning Frames

The system learns the meaning of words by analyzing the relationship between statements that the user makes about the database and the data in the database. The results of this analysis are stored in frames - case frames for verbs and object frames for objects.

### 5.3.1 Case Frames

A case frame is used to store information about object relationships. The case frame is marked by a verb. The system will not perform inferencing using case frames and so the only way to access a case frame is to explicitly mention the marker.

Case frames contain a number of cases which are determined through interaction with the user. Each case has a marker. The marker is either a position number or a preposition. Position numbers are ordinals which indicate the surface position of the phrase in the sentence. A case can have more than one marker, and a case marker may mark more than one case.

The system supports multiple senses for a verb. This corresponds to a verb that is a marker for more than one case frame. There is currently no way to tell the system that a new verb is a marker for a known case frame. Figure 5.2 is a sample case frame for works. It was produced from the sentences "Smith works for Ford", "Allan works for 1100" and "Ford works in Research". In order to produce it, the values "Smith" and "Ford", "Allan" and

1100, and "Ford" and "Research" were independently connected and later combined into one frame.

Marker	Meaning
Syntactic Subject (1)	EMPLOYEE.ENAME
For	MANAGER.ENAME
	EMPLOYEE.SAL
In	DEPT.DNAME

As described earlier, a case frame contains a number of cases - roles that

**Figure 5.2 - Case Frame for Works**

are played in the event corresponding to the case frame. In Figure 5.2, there are four cases which are shown along with their markers. The first case is the subject case which is filled by an employee. The employee is referenced by the name which appears in the ENAME column of the EMPLOYEE table. This case is marked by position. There are two cases which are marked by the preposition "for". One case is the role of manager of the employee who fills the subject case. The manager case is filled by a name in the ENAME column of the MANAGER table. The other case marked by "for" is the employee's salary which is filled by a value in the SAL column of the EMPLOYEE table. The final case is the department that the employee works in. This is marked by the preposition "in". In addition to this information, there is a data structure which describes how these cases are related. This data structure (which is not shown) is the spanning tree which is described in detail in section 5.3.3.

### 5.3.2 Object Frames

Object frames are used to keep track of information about objects. The only information that the system currently has about objects are names, and attributes. Attributes must be



defined in terms of the database information. There is no object hierarchy. Object frames are marked by the object frame names, such as employee, or by the type of an object mentioned such as "Fred"'s type which is EMP table - ENAME column. As described earlier, some grammar productions exist especially for recognizing references to object frames.

Attributes of an object are marked by words such as salary or manager (the system does not calculate manager from manage). The following object frame (Figure 5.3) was produced from the sentences

"Smith's manager is Ford", "The department of Smith is research", and "Allen's salary is 800" which were independently analyzed and combined. The filler information

Class	Marker	Filler
Name	"Name"	EMPLOYEE.ENAME
Attribute	"Salary"	EMPLOYEE.SAL
	"Manager"	EMPLOYEE.MGR
	"Department"	DEPT.DNAME

**Figure 5.3 - Employee Object Frame**

actually references the spanning trees described later in this section and not the database directly. The fillers are shown for illustration only.

### 5.3.3 Minimal spanning Trees

The knowledge structure used in the frames of the system is a type of semantic net. The semantic net is viewed as a spanning tree over a database graph. The semantic net in this context is called a *generalized minimal spanning tree* of the *database graph* for the set of *database graph nodes*, N. The next section defines these terms and presents an algorithm

for calculating these spanning trees.

### 5.3.3.1 Definitions

#### 5.3.3.1.1 Database Graph

The *database graph* (DBG) is implicitly represented by the database. Each node in the database graph is the entry at a particular row and column of a table in the database<sup>6</sup>.

There is an edge between two database graph nodes if they are in the same row or if they have the same value<sup>7</sup> (used for joins). Associated with each edge between nodes that have the same value is the join condition required to connect the tables corresponding to each of the nodes. Figure 5.5 depicts the database graph for the portion of the database shown in Figure 5.4. In the graph in Figure 5.5, the thicker edges connect nodes of the same value and the thinner edges connect nodes in the same row. The observant reader will note that not all of the thinner edges have been shown. For each pair of nodes in the same row, there should be a thin edge incident. All of these thin edges are not shown because the drawing would need to contain 212 thin edges to accurately reflect the connections among nodes in the same row. The algorithm will act as though all of these edges are present.

---

<sup>6</sup> - See appendix A for a brief description of relational database terminology

<sup>7</sup> - Exceptions:

- nodes cannot be in the same column of the same table
- no reflexive edges (edge from a node to itself)

---

DEPT	
DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH

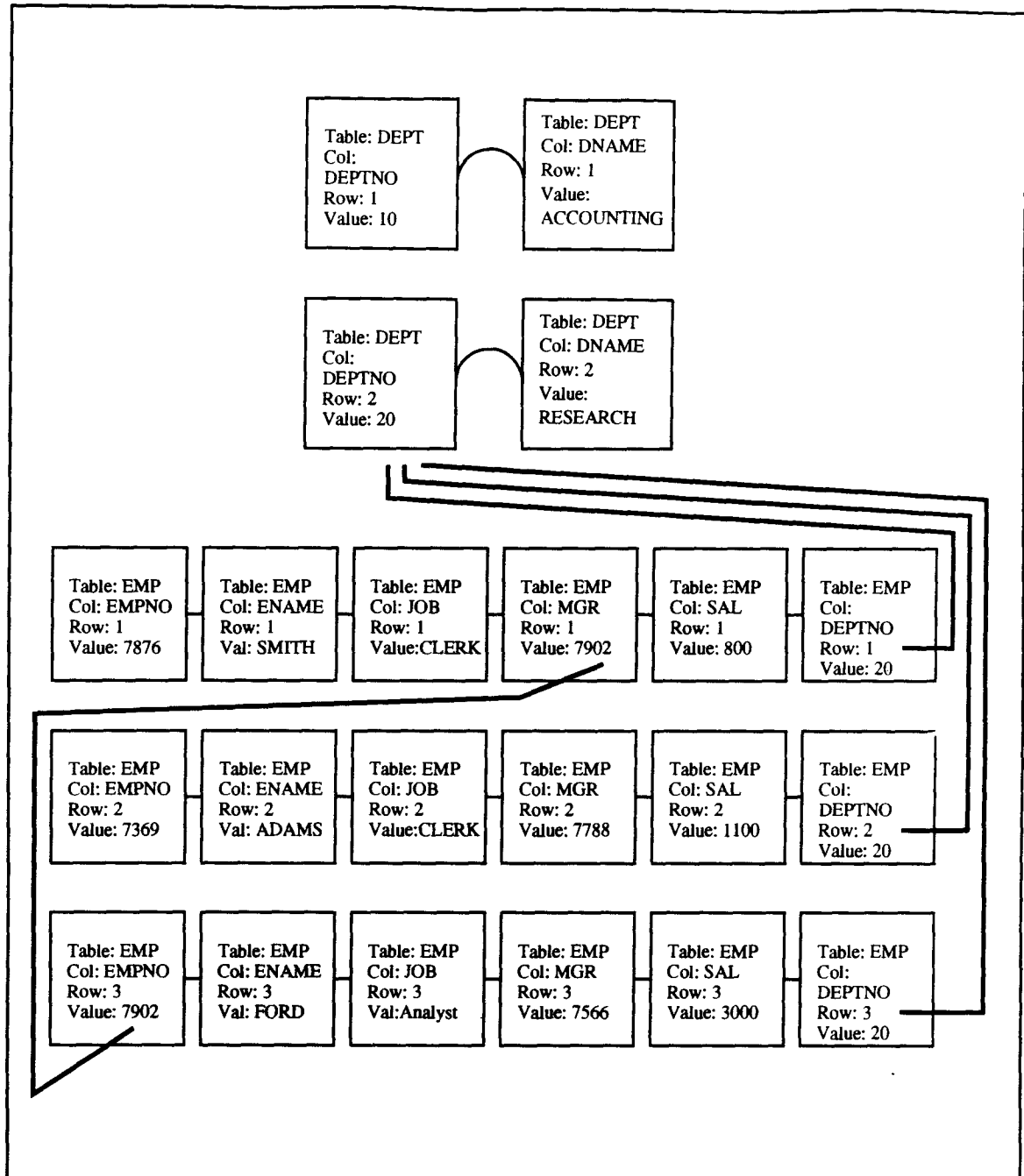
EMP							
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7902	FORD	ANALYST	7566	03-DEC-81	3000		20

---

**Figure 5.4 - Tables Used by Examples**

#### 5.3.3.1.2 Minimal spanning Tree

A *minimal spanning tree* (MST) is a sub-tree of the database graph. The minimal spanning tree is defined relative to a set of database graph nodes,  $N$ . *Spanning* means that the tree contains the set of database graph nodes,  $N$ . The tree is *minimal* in the sense that there is no proper sub-tree of the tree which spans the given database graph nodes,  $N$ . The database graph nodes correspond to phrases mentioned in the defining sentences. Figure 5.6 contain minimal spanning trees for the connecting the values "Smith", "Clerk", and "800".



**Figure 5.5 - Database Graph for Table of Figure 5.4**

#### 5.3.3.1.2.1 Formal Definition of MSTs

For the following definitions,  $G$  is a database graph,  $Values$  are a set of values that appear in the database, and  $Nodes$  is a set of database graph nodes that correspond to the values in

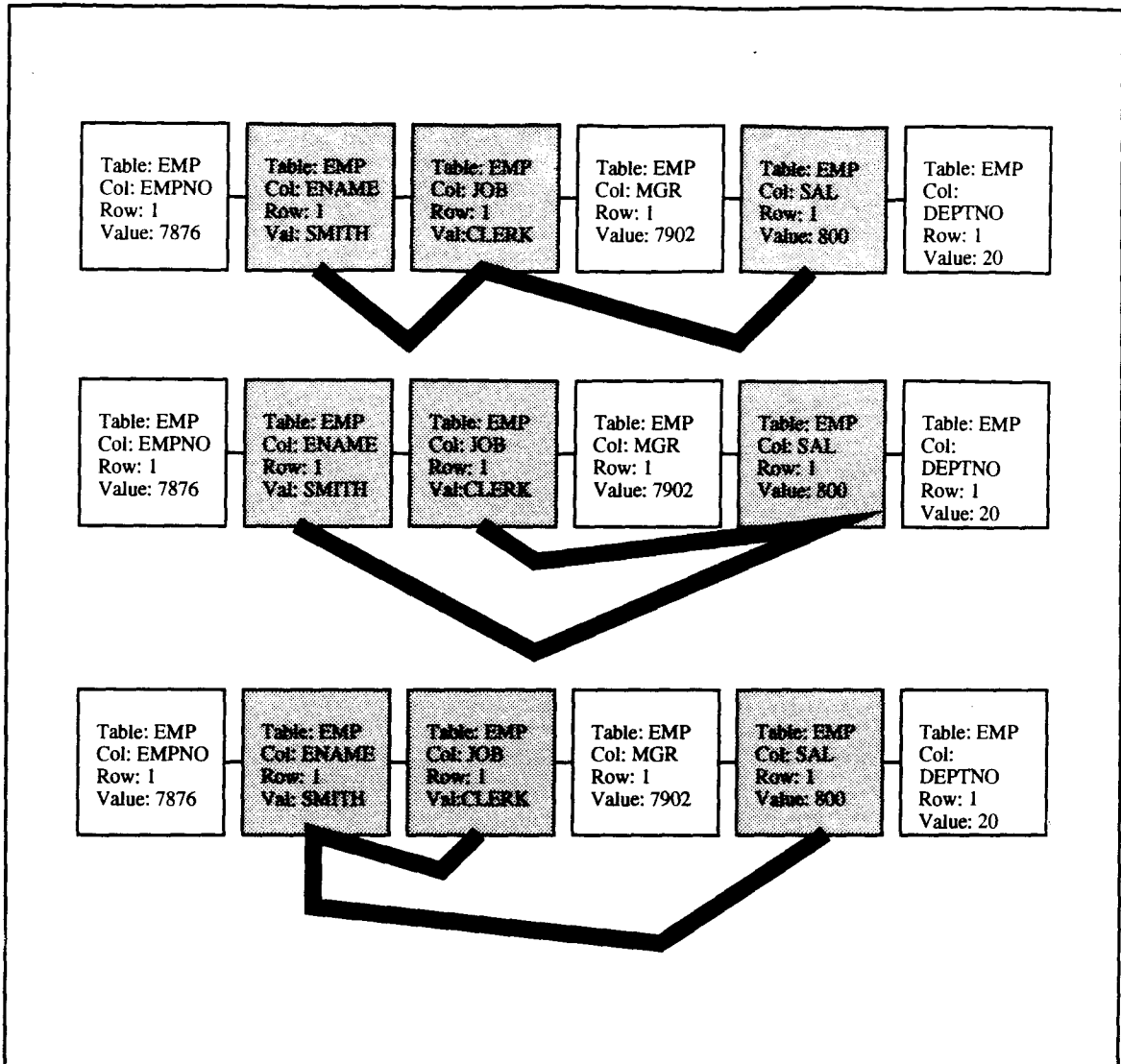


Figure 5.6 - Three Minimal Spanning Trees for "Smith works as a clerk for 800 dollars"

the set of values, Values.

$$SpanningTrees_G(Nodes) = ST_G(Nodes) = \{t | t \subseteq G \wedge isTree(t) \wedge Nodes \subseteq Nodes(t)\} \quad (1)$$

$$MinSpanTrees_G(Nodes) = MST_G(Nodes) = \{t | t \in ST_G(Nodes) \wedge \neg \exists (t' \in ST_G(Nodes)) t' \subset t\} \quad (2)$$

$$\text{Meaning of Values is select one of } \bigcup_{\substack{\text{Each Set of Nodes} \\ \text{for Values}}} \text{MST}_G(\text{Nodes}) \quad (3)$$

As seen in equation (3), the meaning is an arbitrarily selected tree of a set of MSTs. The goal was to try to avoid using structural properties of the tree as the sole criteria for selecting the meaning. In some cases, interaction with the user is necessary to determine the correct interpretation (WHITE85). By providing this leeway, the system is able to impose other selection constraints automatically or to interact with the user in order to select a meaning.

### 5.3.3.1.3 Generalized Minimal spanning Tree

#### 5.3.3.1.3.1 A Problem with MST

Although minimal spanning trees effectively capture meaning, they are not efficient. Large numbers of equivalent trees can be produced for a given set of values. As shown for the sentence, "Smith works as a clerk for 800 dollars", there are three equivalent spanning trees (see Figure 5.6). By using a generalized minimal spanning tree (GMST), the number of alternatives is reduced to one (see Figure 5.7).

Table: EMP
Row: 1
ENAME
SMITH
JOB: CLERK
SALARY: 800

**Figure 5.7 - GMST**

These problems with the MSTs arise because the nodes in the same row form a complete

sub-graph of the database graph. Each of the different minimal spanning trees select a different path through this sub-graph. The generalized minimal spanning tree (GMST) was defined to avoid this.

#### **5.3.3.1.3.2 GMST Definition**

The GMST is an equivalence class of MSTs. The nodes of the GMST are sets of columns in the same table and row. There is an edge between GMST nodes iff there is an edge in the MST between columns in each GMST node. For example, the GMST in Figure 5.10 corresponds to the MST in Figure 5.8 (and Figure 5.9). Notice that there is a node in the GMST for each row in the MST and the edges in the GMST corresponds to the 'join' edges of the MST. The algorithm is given in terms of the MST because that is easier to understand and changing the algorithm to work with GMSTs only requires converting the data structure that represents the MST to a data structure that represent the GMST equivalence class.

#### **5.3.3.2 Examples**

The following example illustrates the various definitions for the sentence "Smith works as clerk in research under Ford for 800 dollars". This sentence is hard to process because there is a lot of ambiguity about how the database is encoding this information. The purpose of using this sentence is to generate a complicated example.

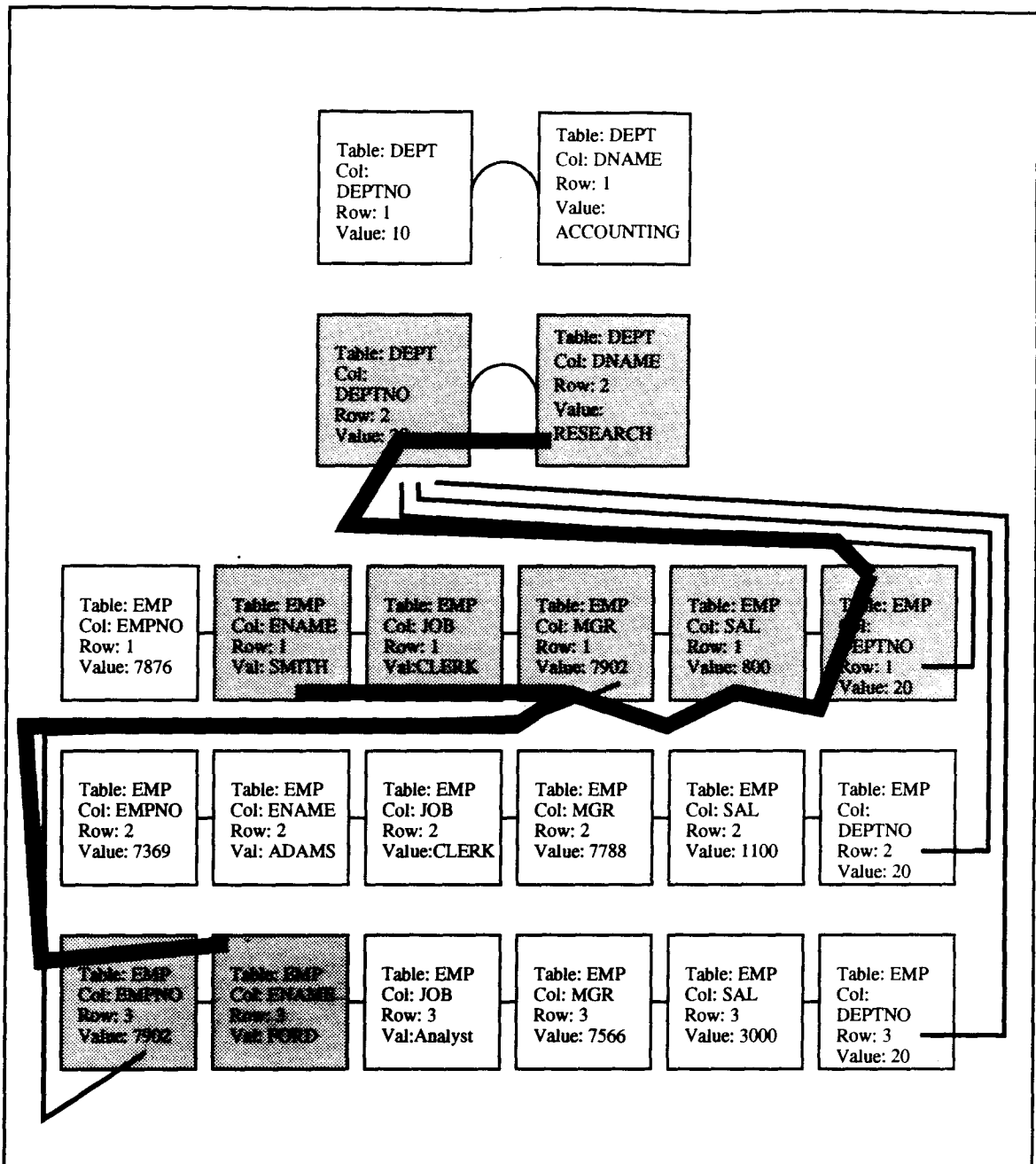
### **5.3.3.2.1 Values to DBG Node Mappings**

The first step in the process of generating trees is to associate values in the sentence with nodes in the trees. This will be shown in detail in a later example (section 5.3.3.3.2).

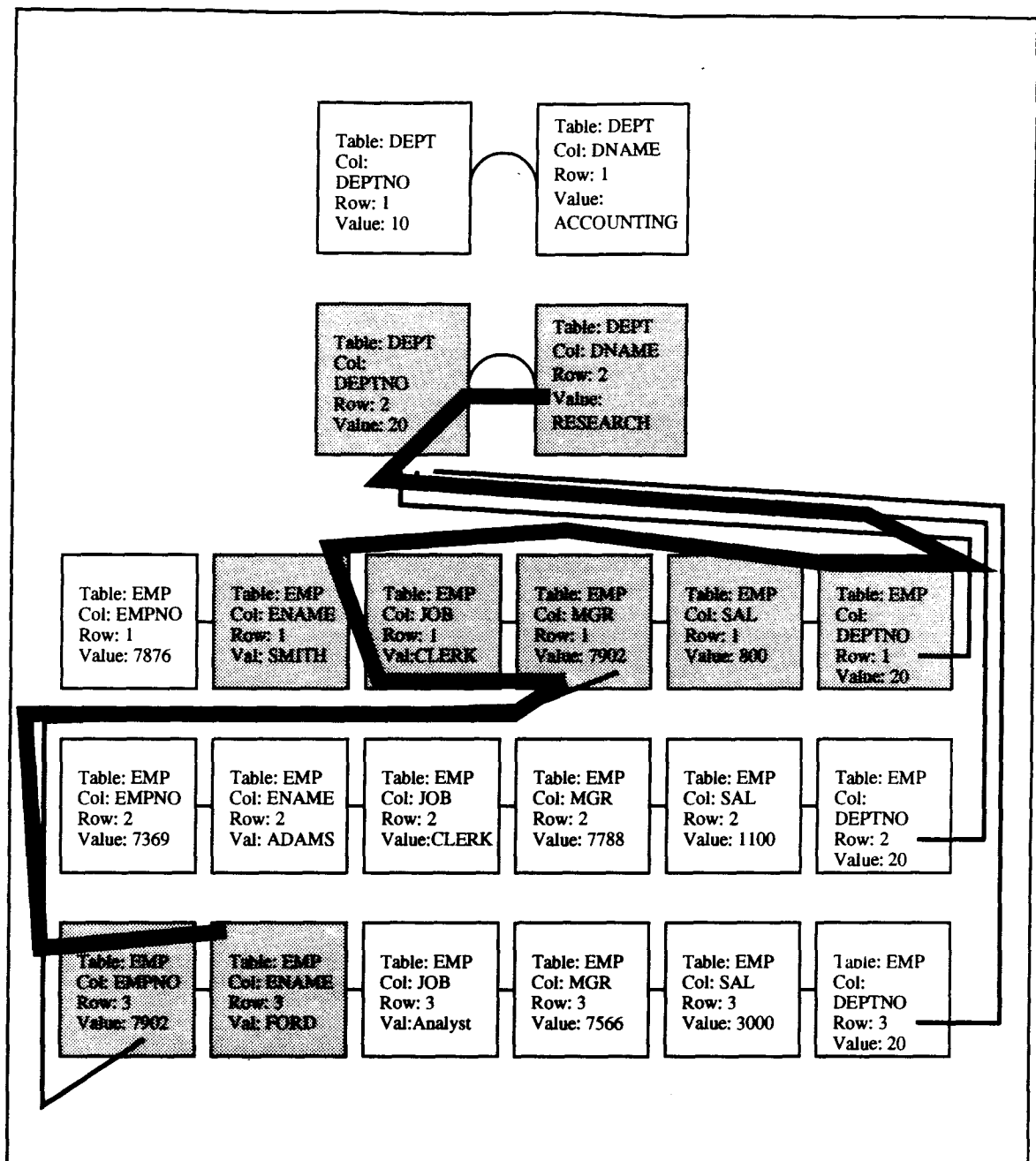
### **5.3.3.2.2 MSTs**

Figure 5.8 and Figure 5.9 depict two MSTs for the example. The thick line marks the MST. Nodes that are in the MST are darkened. There are many others which are not shown for the sake of brevity.





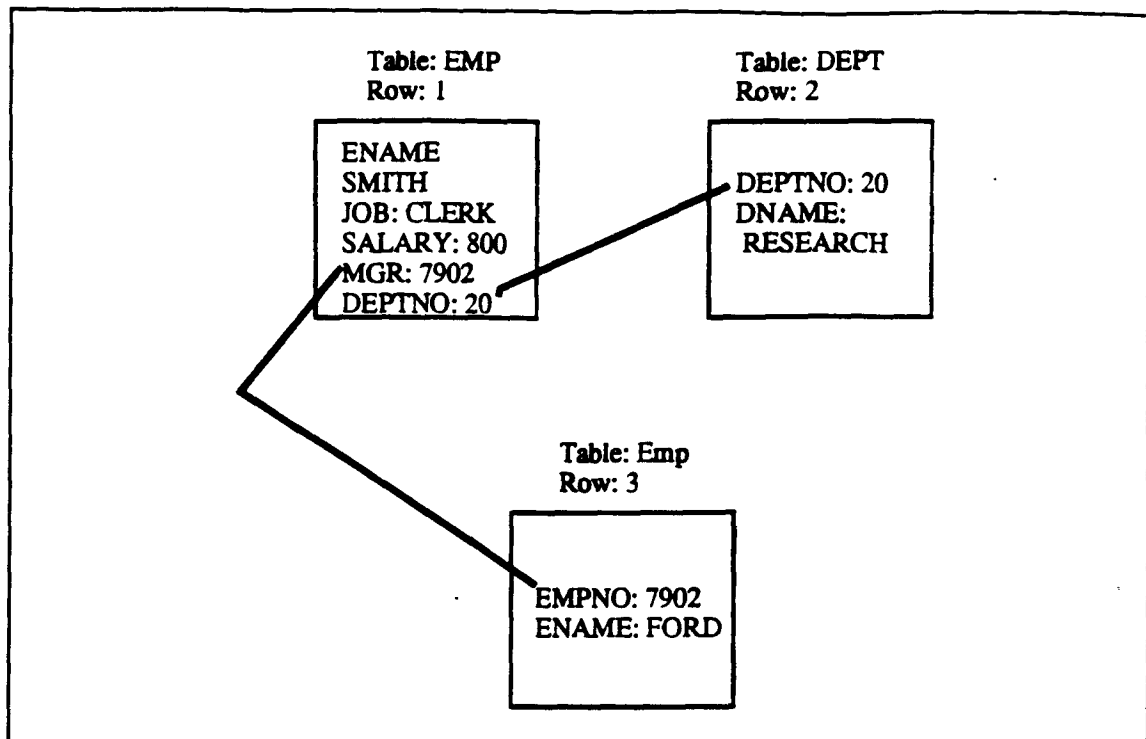
**Figure 5.8** Example minimal spanning trees connecting "Smith", "Clerk", "800", "Research", and "Ford"



**Figure 5.9 - Another Spanning Tree For "Smith", "Clerk", "800", "Research", and "Ford"**

#### 5.3.3.2.3 GMSTs

Figure 5.10 depicts one of the GMSTs for the sentence. Another possibility, not shown, is to associate "Research" with "Ford". Each box in the diagram marks a GMST node which is identified by the table name and the row number. The edges connecting the nodes mark



**Figure 5.10 - Generalized Minimal Spanning Tree**

the columns which are used to join the tables corresponding to the nodes which are being connected.

### 5.3.3.3 Algorithm

This section presents the algorithm for calculating the GMSTs. A number of heuristics are described, first followed by an example and then by the definition of the algorithm. The approach is based upon well known graph algorithms for calculating spanning trees.

#### 5.3.3.3.1 Heuristics

According to the definition of the MSTs, the meaning is selected from one of a number of

trees. No additional criteria for selection is given. This section describes some additional constraints that are used in order to select a MST.

#### 5.3.3.3.1.1 Produce the Best Guesses First (Solution Space Ordering)

One common approach used to enhance search is to use a cost function. A cost function is used here but it should be stressed that the cost function is used to affect the order in which solutions are generated and is not used to eliminate GMSTs from consideration but only to possibly postpone their consideration. The goal is to produce the best guesses first and avoid presenting the user with too many choices at once.

##### 5.3.3.3.1.1.1 Tree Cost

According to Occam's Razor the simplest approach may be the best. In this case, simple was taken to mean the fewest number of table joins (ie MST edges) and the fewest number of nodes. The cost of a tree is defined in such a way that ideally the

Type	Cost
Key	1
Foreign Key	2
Other	20

**Figure 5.11 - 'Join' Edge Cost**

simplest trees will have lowest costs. The cost of a MST is the 'join' edge cost plus the 'same row' edge cost. Currently the 'same row' edge cost is 4. The 'join' edge cost depends on the type of nodes joined. These nodes can be keys<sup>8</sup>, foreign keys or other. The cost of a 'join' edge is the sum of the node type cost of the incident nodes. Currently the

---

<sup>8</sup> - See appendix A

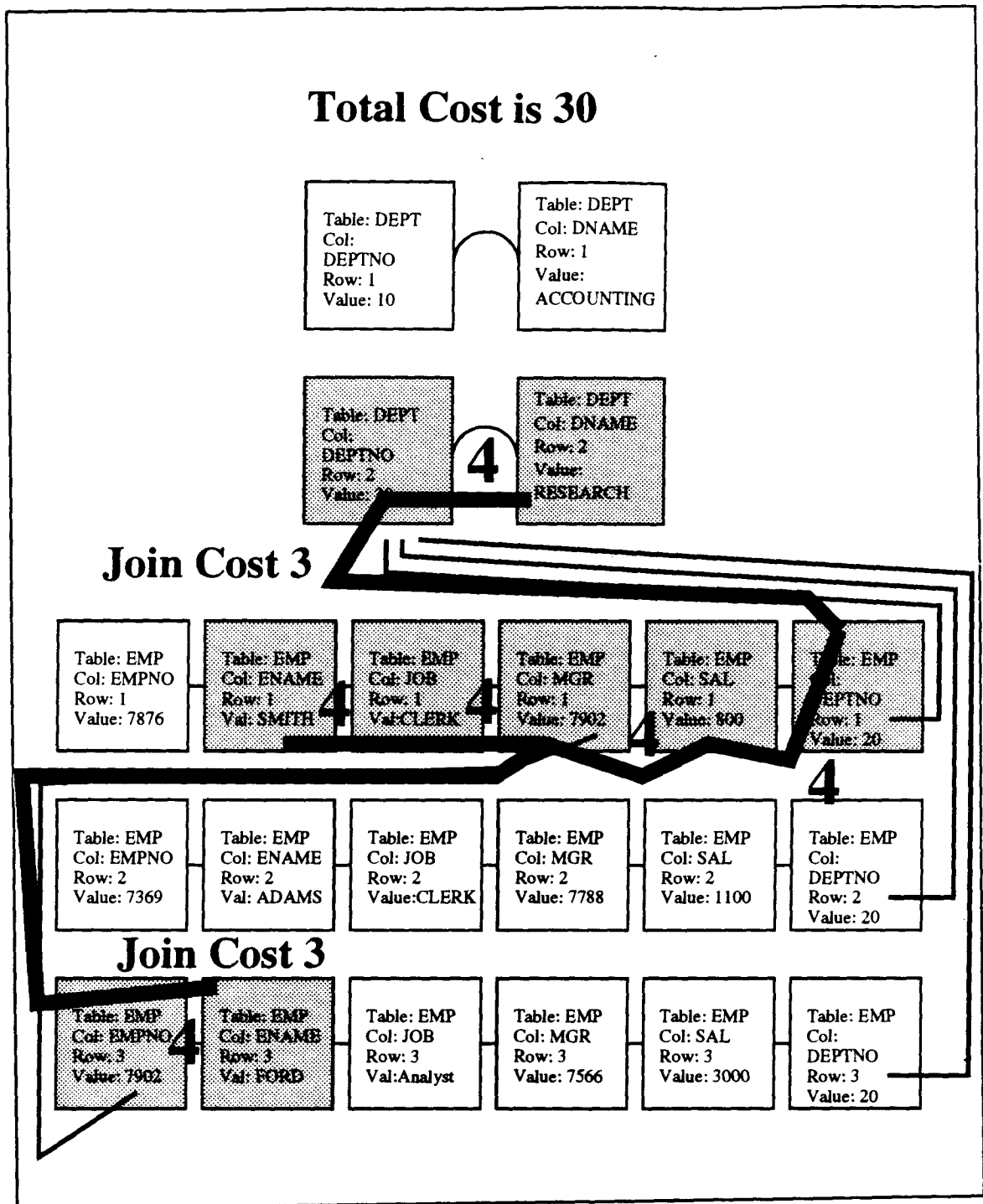
cost of a key node is 1, foreign key is 2, and other 20. These values were determined by trial and error. Figure 5.12 illustrates a tree cost calculation. In this example, the cost of connecting the EMPNO node of the EMP table at row 3 to the MGR node of the EMP table at row 1 is 3 because the former is a key which has cost 2 and the latter is a foreign key which has cost 1.

#### **5.3.3.3.2 Example**

The algorithm used for calculating the spanning tree of a given set of values combines a slight generalization of Prim's algorithm for calculating spanning trees of a graph with a best first search (TARJAN83). The basic approach is to start with a tree that spans one of the values. This is a tree with only the node corresponding to the value in it. Then for each other value to be connected, a path is sought from the node for that value to a node in the current spanning tree. The following example illustrates the generation of a minimal spanning tree for the values "Smith", "Clerk", "800", "Research", and "Ford". Note that the example is generating a MST not a GMST in order to better illustrate important points. The example is a depth first search that makes correct decisions at all points. The algorithm defined later is a best first search.

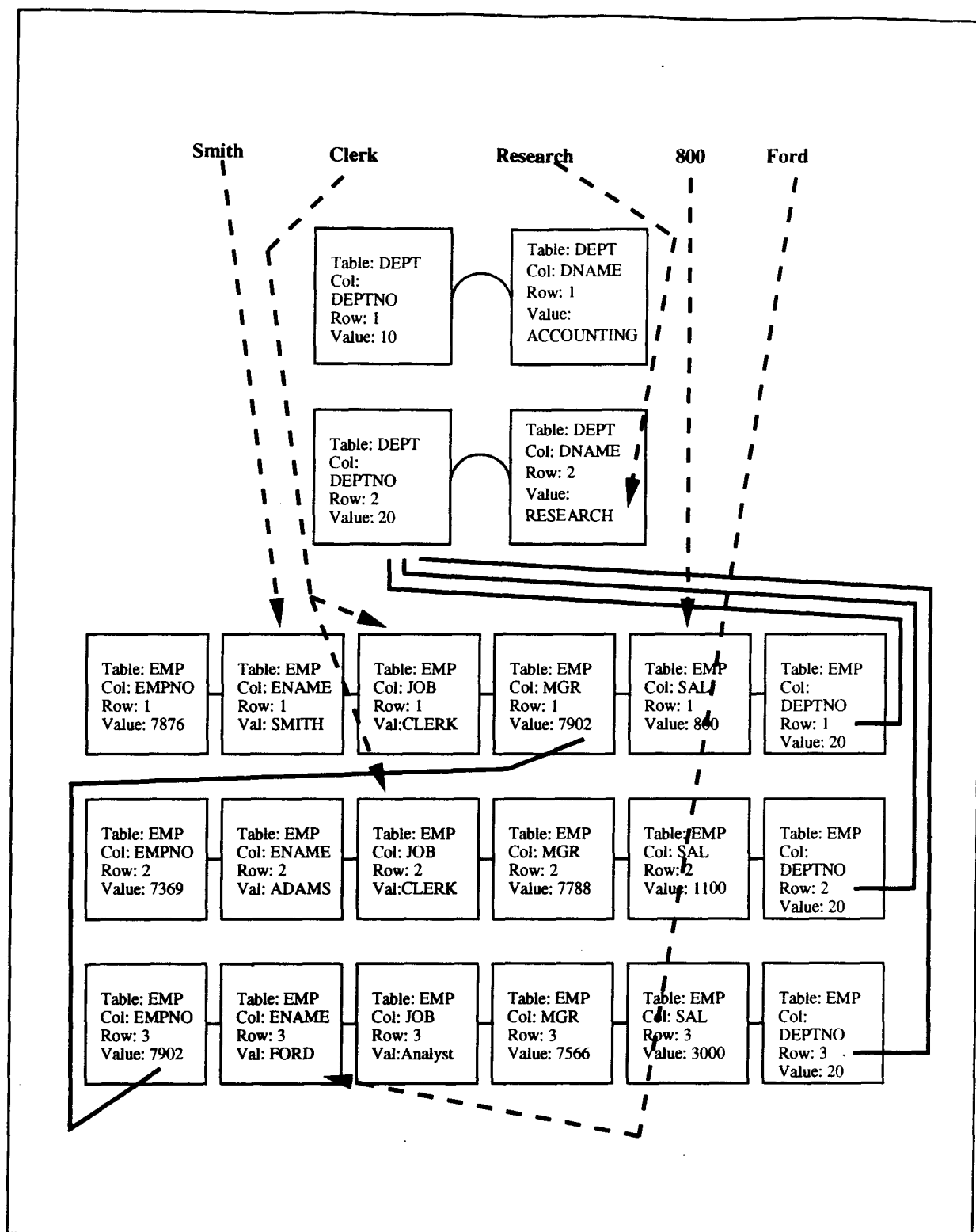
##### **5.3.3.3.2.1 Establish Value to Node Mapping**

The first step is to establish a correspondence between the values mentioned in the sentence and the database graph nodes. At this point morphing and spell checking could cause the



**Figure 5.12 - Tree Cost Calculation**

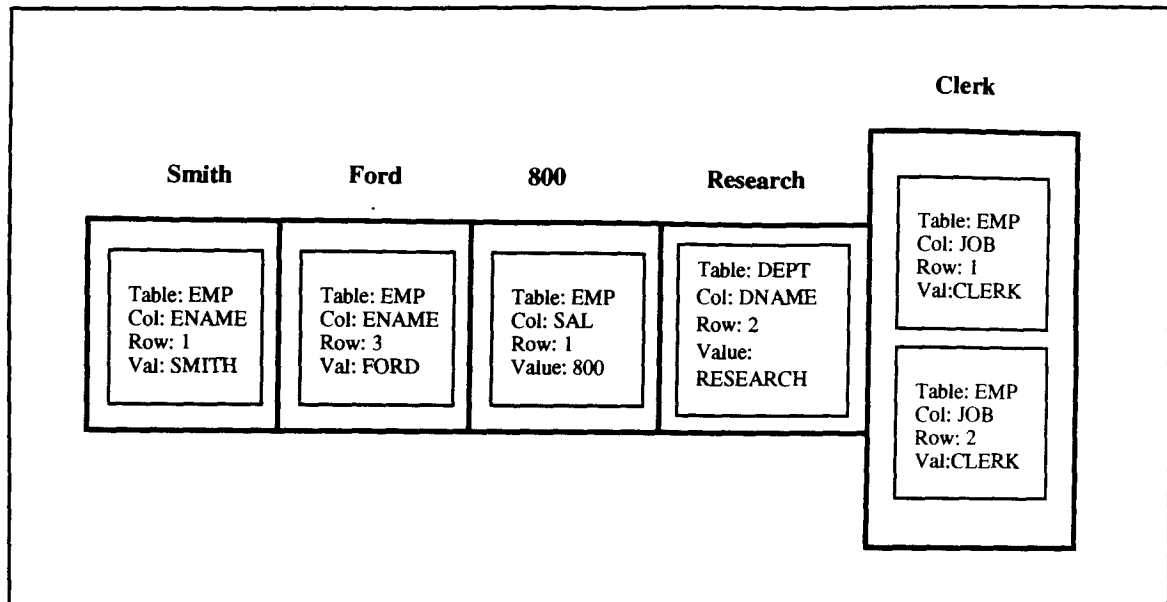
value to be mapped to a node that does not have exactly the same value. However this has not been implemented. Figure 5.13 depicts the mapping of values to nodes. The values are listed at the top of the figure and dotted arrows are used to indicate their corresponding nodes.



**Figure 5.13** - Establish Value to Database Graph Node Correspondence



After determining the correspondence, a priority queue of node groups is set up. The queue contains one set of nodes for each value to be connected. The smaller sets are listed first in arbitrary order. Figure 5.14 contains the initial values queue. The front of the queue is to the left.



**Figure 5.14 - Values Queue**

After the queue is set up, the frontier is created. The first MST contains one of the database graph nodes in the first set of database graph nodes from the priority queue. There is one MST for each node in the set. In this case there is only one MST. Figure 5.15 depicts the first MST with one node in it (nodes in the current MST are marked by grey shading) and Figure 5.16 shows the values queue after the first entry is removed.

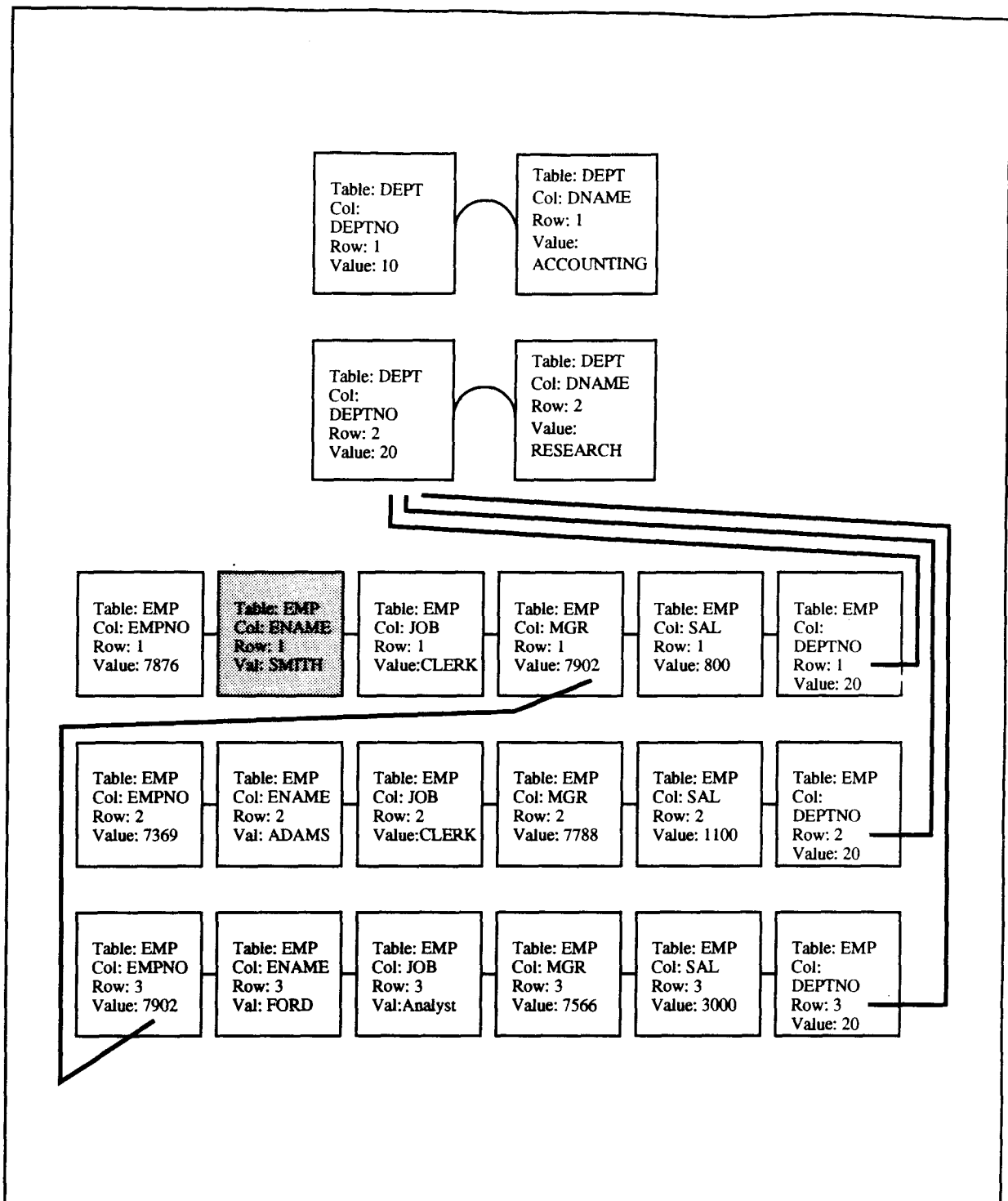
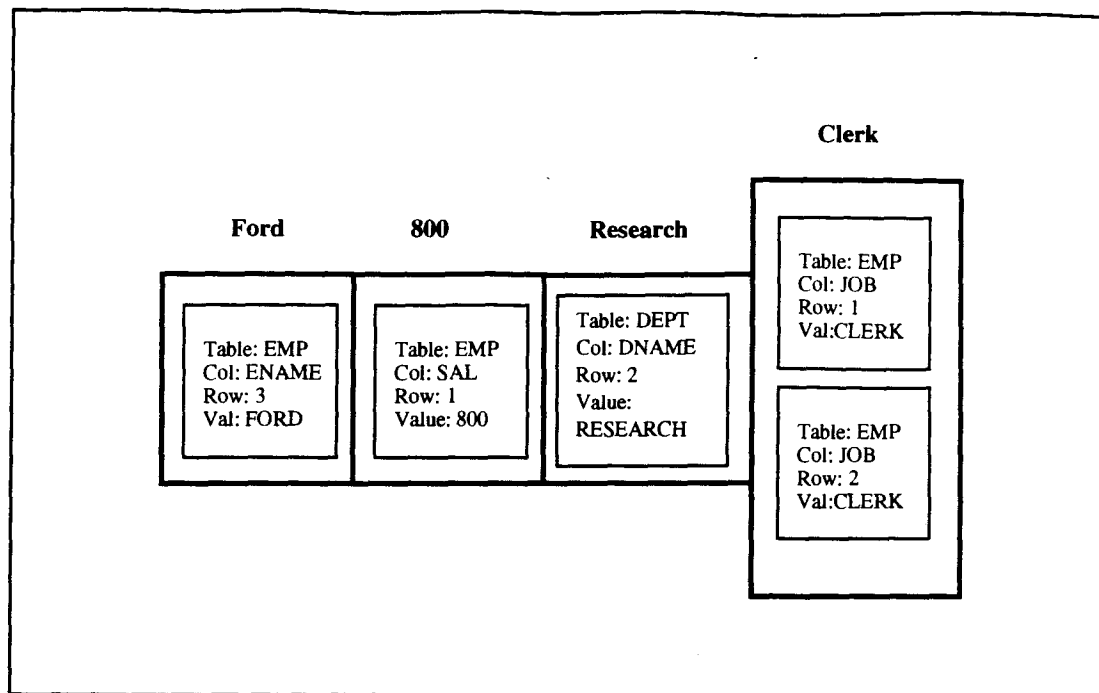


Figure 5.15 - First MST



**Figure 5.16 - New Values Queue**

The next step is to connect the node for "Ford" to the current MST. This is done by searching for paths from each node for "Ford" (marked by box with diagonal line shading) to the MST. In Figure 5.17 the dashed line marks the path found from "Ford", the new node, to the current MST. All nodes along this path including the first are shaded with diagonal lines. Figure 5.18 shows the values queue after adding "Ford".

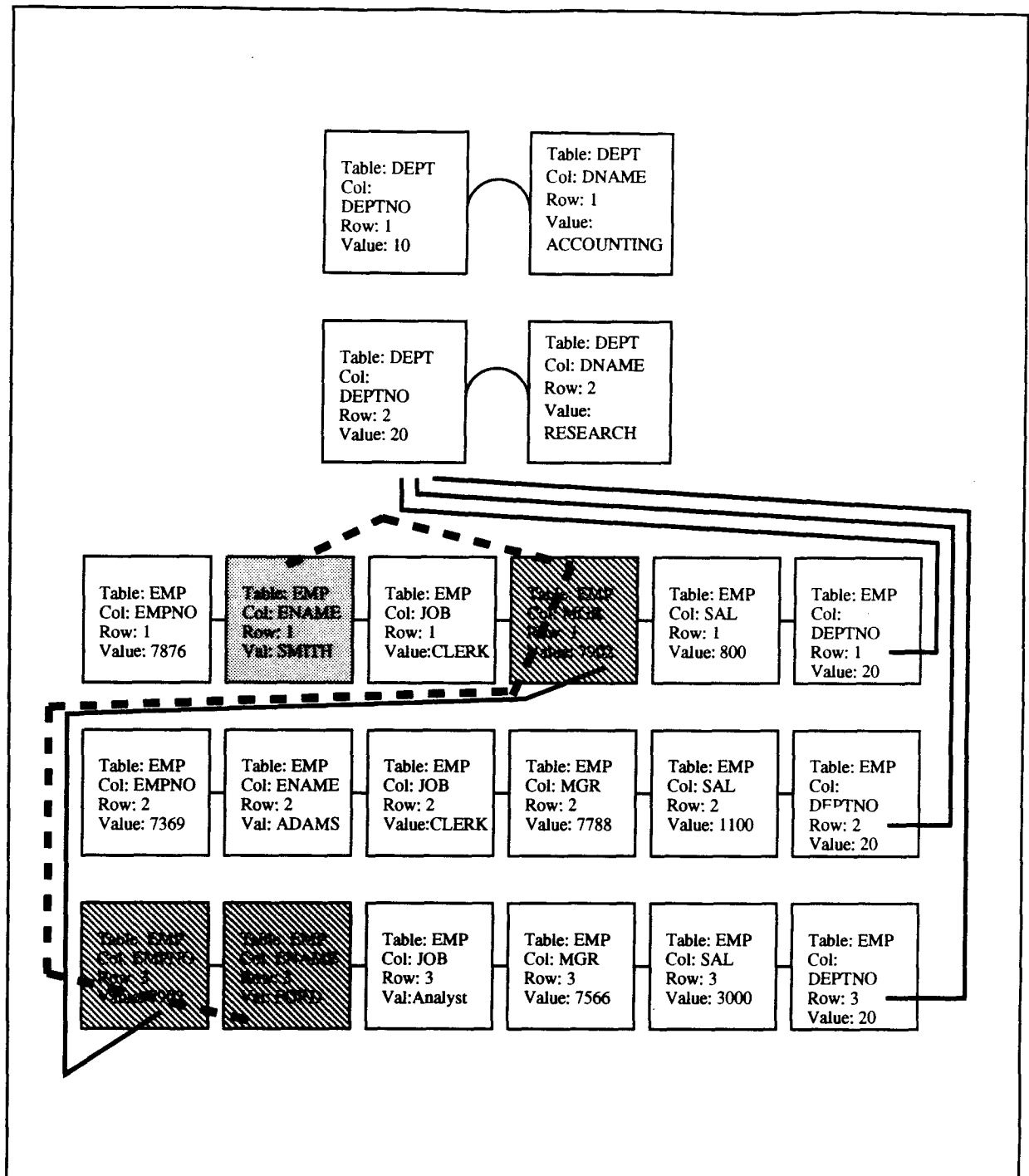
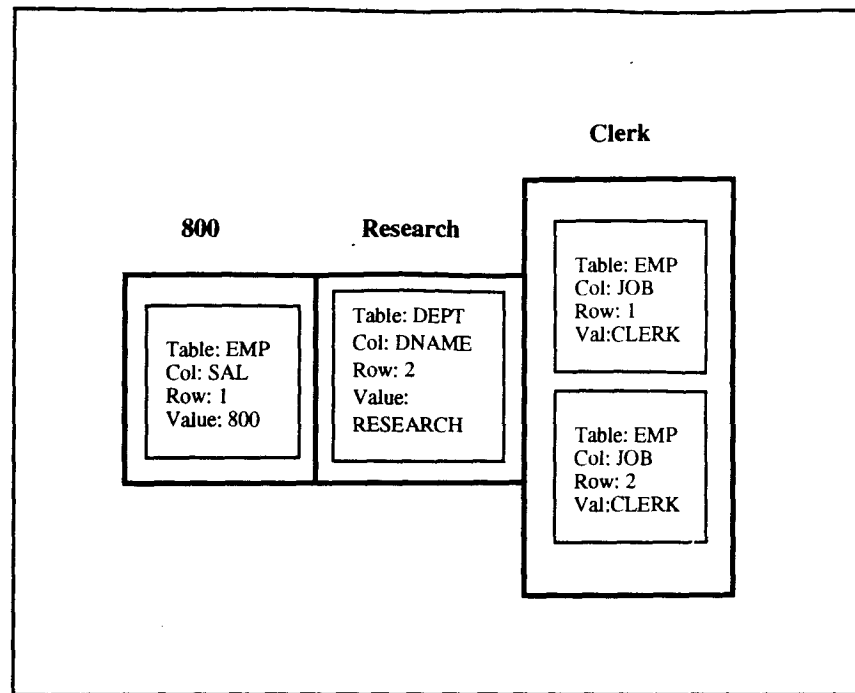


Figure 5.17 - Connecting Ford



**Figure 5.18 - New Values Queue**

Now we add 800 (Figure 5.19). Figure 5.20 contains the queue after adding 800.

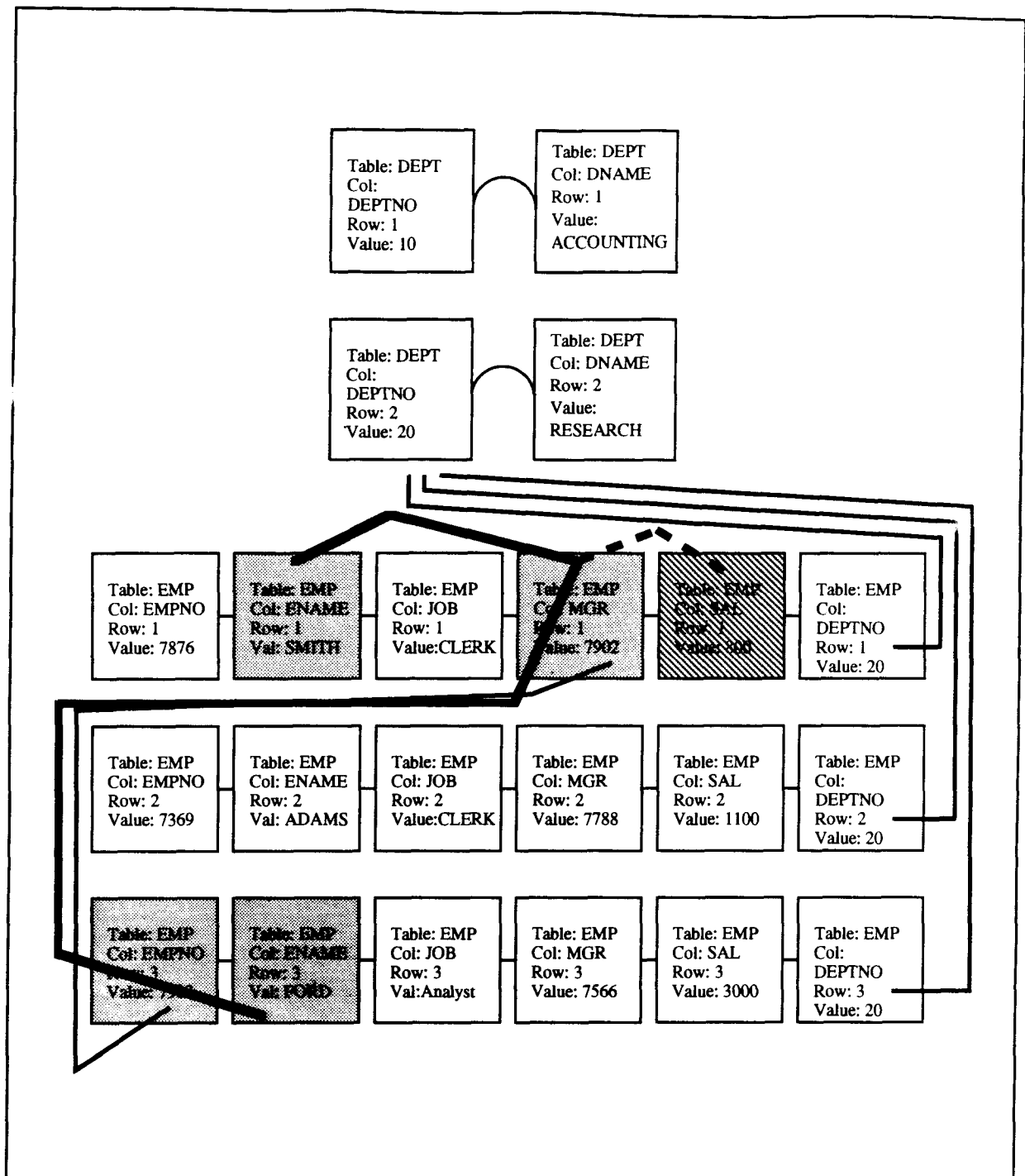
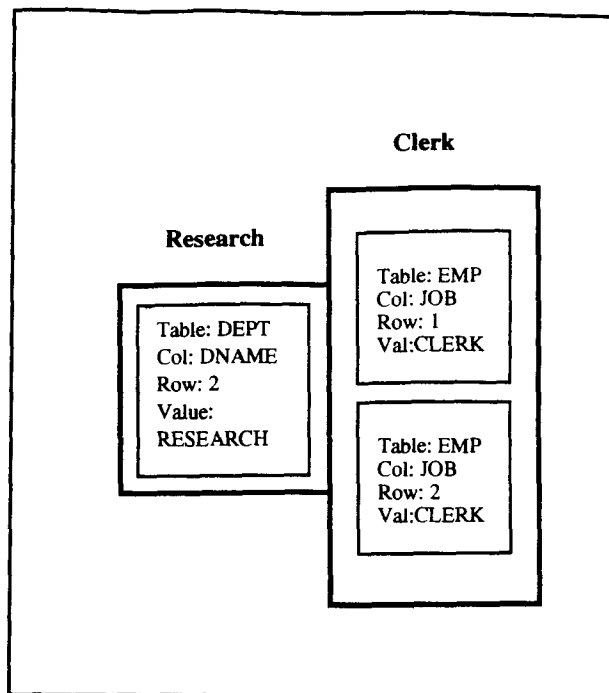


Figure 5.19 - Connecting in 800



**Figure 5.20 - New Values Queue**

"Research" is connected now (Figure 5.21). At this point there is ambiguity about whether or not to connect it to "Ford" or to "Smith". The real algorithm would allow both. Here, we will arbitrarily choose "Smith".

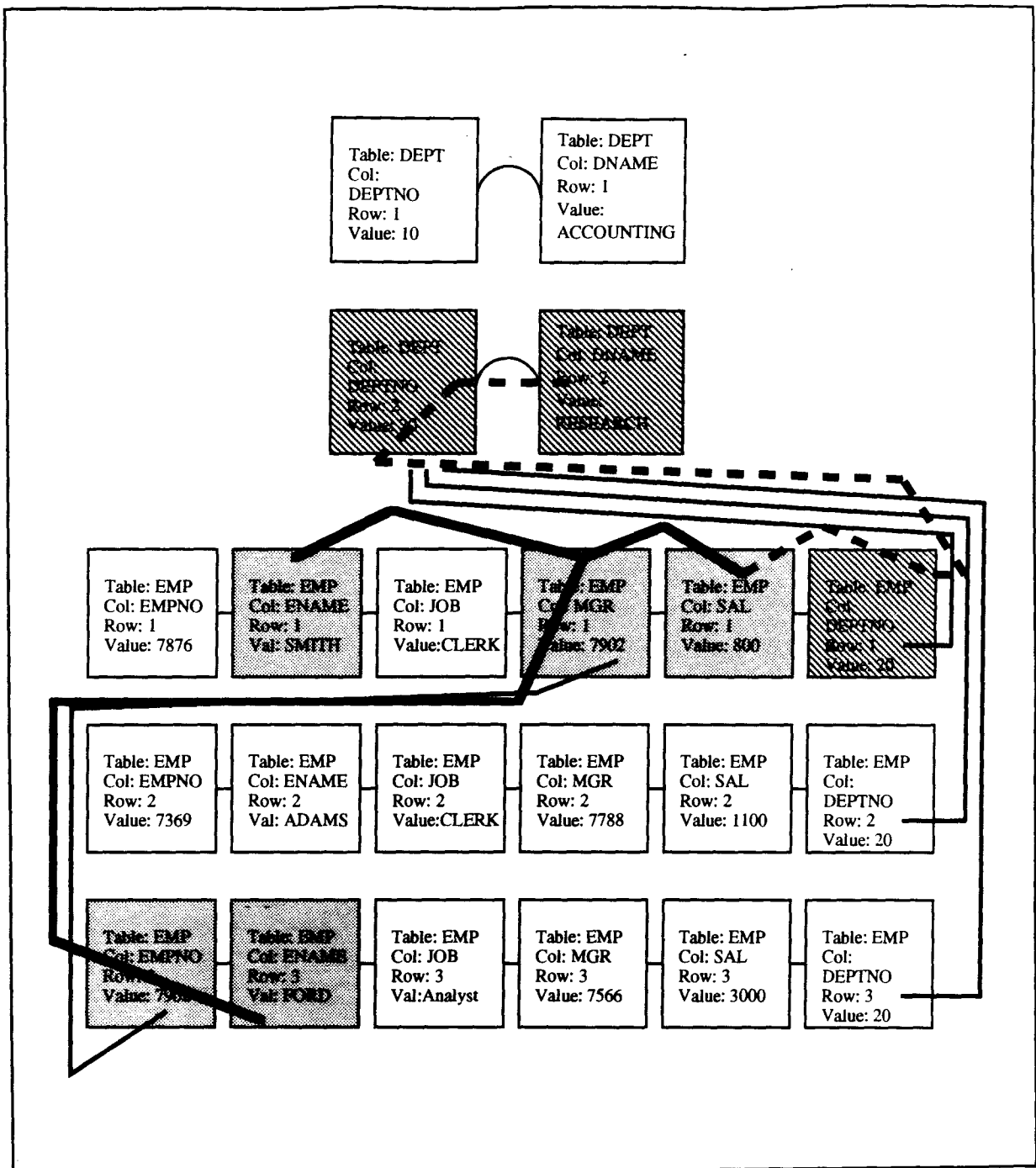
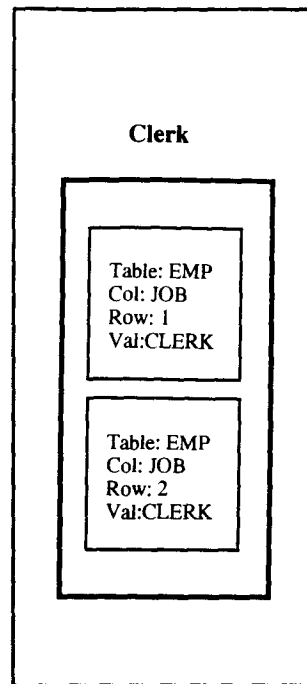


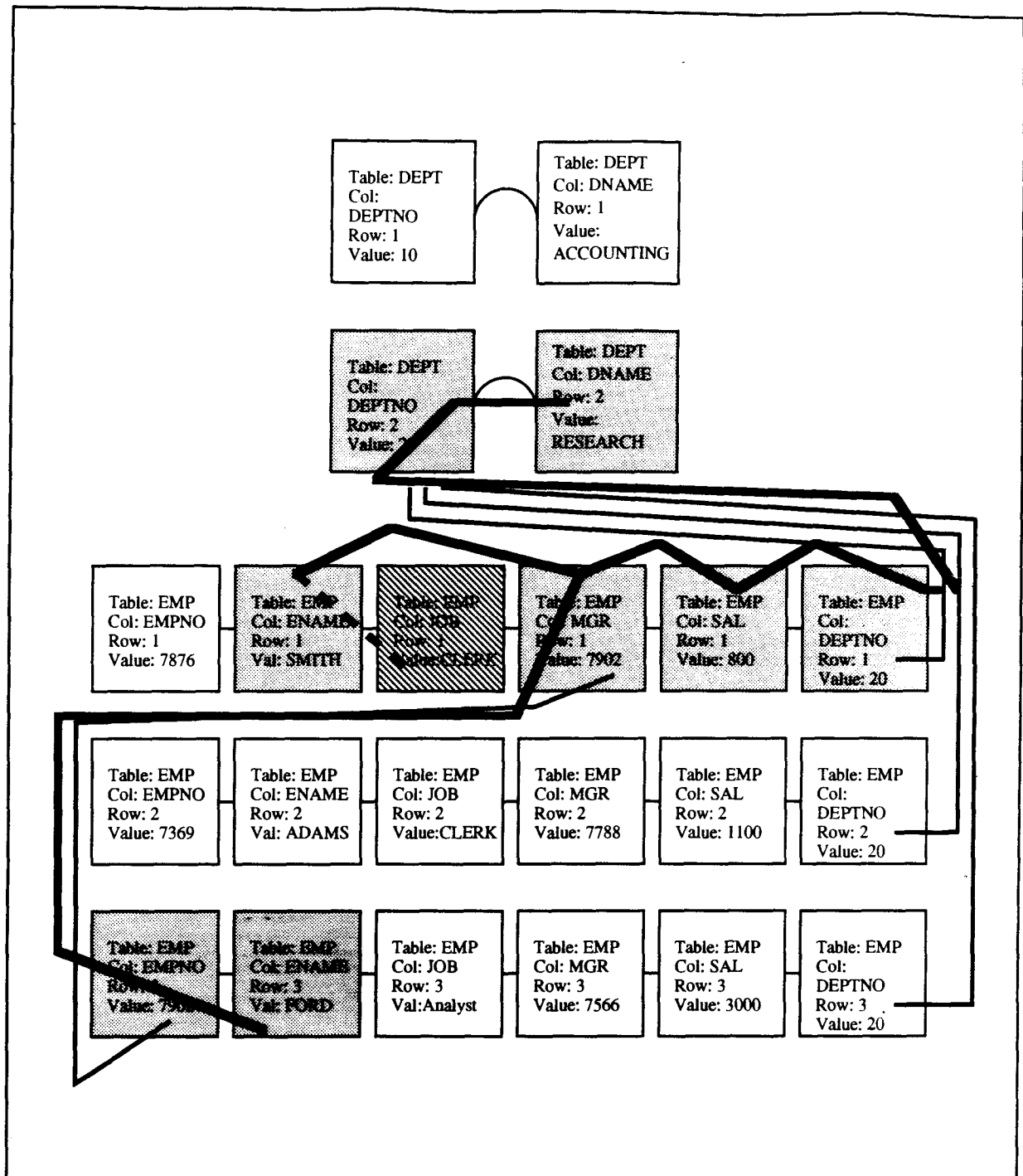
Figure 5.21 - Connecting "Research" to "Smith"





**Figure 5.22 - Next  
Values Queue**

The last step is to connect "Clerk" (Figure 5.23). Here we must choose a database graph node to represent "Clerk" from the two choices. We will arbitrarily select the correct one. The real algorithm which does a best first search would try the other node for "clerk" and then immediately give up due to cost. It would try that node again later if the user insisted on seeing more alternative interpretations.



**Figure 5.23 - Connect in "Clerk"**

At this point the queue is empty and so the tree is complete. This example left out a number of implementation details which are discussed in later sections. The next section presents a pseudo-code definition of the best first search implementation of this algorithm.

### 5.3.3.3 Definition

```

gmsts( [M, N], Frontier, Solutions )

  IF Frontier is Empty THEN
    RETURN

  get best GMST-ValuesQueue from Frontier

  IF cost(GMST) > N THEN
    RETURN

  IF ValuesQueue is empty THEN
    IF cost(GMST) >= M THEN
      Add GMST to Solutions
    ELSE
      get next NodesList from ValuesQueue
      FOR EACH node IN NodesList
        find path, P from node to GMST
        IF P union GMST is a Tree
          add (P union GMST)-ValuesQueue to the Frontier
      NEXT node

  gmsts( [M, N], Frontier, Solutions )

```

**Figure 5.24** Pseudo-Code Which Generate GMSTs such that  $M \leq \text{cost}(\text{GMST}) \leq N$

Frontier is a priority queue with the lowest cost GMSTs listed first. Values queue is a priority queue with the smallest list of nodes first. As mentioned earlier this is a best first search implementation of the approach demonstrated by the example.

### 5.3.3.4 Implementation Notes

#### 5.3.3.4.1 Cycle Check

The implementation of the algorithm takes advantage of the fact that the frontier contains

only trees. So when a new node is connected to a tree in the frontier, the resulting graph is connected. Since the graph is connected then it is well known that the graph is a tree if and only if the number of edges is equal to the number of nodes minus one. This provides an efficient test for performing a cycle check after a node is connected to a tree in the frontier.

#### 5.3.3.4.2 Extension of Tree Isomorphism

The GMSTs are used to eliminate redundant MSTs. However it was realized late in the development stage that there is another form of redundancy the GMSTs do not eliminate.

Consider the following GMSTs for the sentence "TKB Sport shop bought an ace tennis net for 58 dollars".

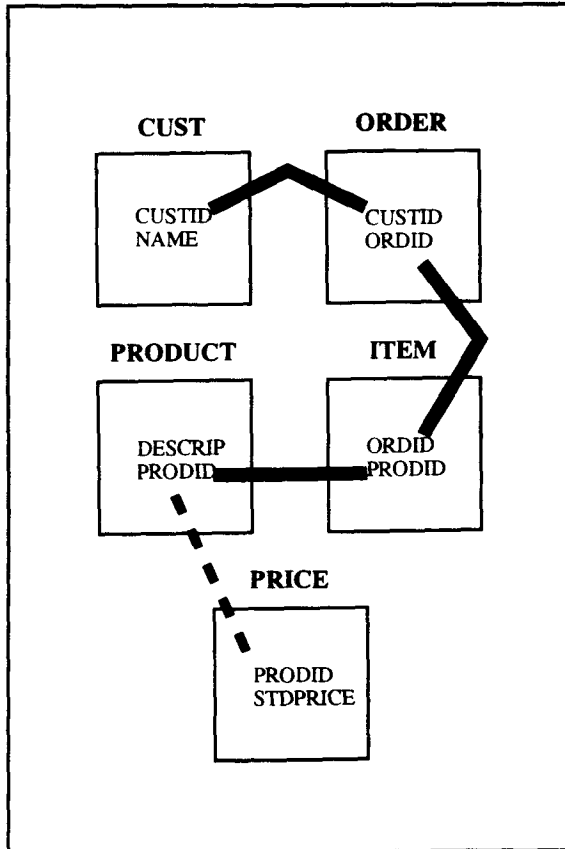


Figure 5.25 - Choice 1

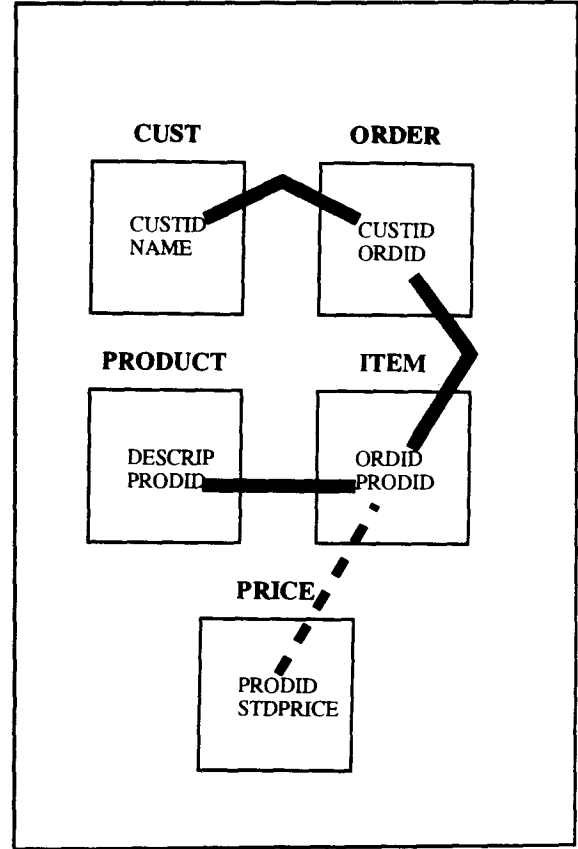


Figure 5.26 - Choice 2

The transitivity property of equality (arising from the join conditions) implies that these two GMSTs are equivalent. To see this, consider the join conditions that are represented by the graphs in Figure 5.25 and Figure 5.26. There is only one difference between these two trees. In Figure 5.25, the PRICE table is connected to the PRODUCT table by the dashed edge. In Figure 5.26, the PRICE table is not connected to the PRODUCT table but is instead connected to the ITEM table. Each dashed edge corresponds to a join condition in the SQL. For the first graph (Figure 5.25), the join condition is  $PRICE.PRODID = PRODUCT.PRODID$  and for the second graph (Figure 5.26), the join condition is  $PRICE.PRODID = ITEM.PRODID$ . For both graphs there is an edge connecting the PRODUCT and the ITEM nodes. This edge corresponds to the join condition  $ITEM.PRODID = PRODUCT.PRODID$ . This implies that the join condition of the two graphs are logically equivalent. To see this, consider the graph of Figure 5.25, the join condition includes  $PRICE.PRODID = PRODUCT.PRODID$  and  $PRODUCT.PRODID = ITEM.PRODID$ . From this, one can deduce that  $PRICE.PRODID = ITEM.PRODID$  (ie -  $(A - B \wedge B - C) \supset A - C$ ). This is the join condition for the other graph as well. So the two graphs describe logically equivalent interpretations.

This consideration was incorporated in the definition of GMSTs isomorphism used in the program but not directly in the GMST data structure. It should also be noted that the costs of these two graphs are not the same. In one case the join is between two keys ( $PRICE.PRODID$  and  $PRODUCT.PRODID$ ) and in the other the join is between a key and a foreign key ( $PRICE.PRODID$  and  $ITEM.PRODID$ ).

#### 5.3.3.4.3 Can't Join Table to itself by the Same Column (Directly or Indirectly)

Another constraint put on the implementation is to not allow a table to be joined to itself by the same column. This can happen directly or indirectly as a result of joining a number of tables. For example

Join: Customer1 to Order1 by Customer1.CustID = Order1.CustID  
Join: Order1 to Customer2 by Order1.CustID = Customer2.CustID

Due to the transitivity of equality, the effect of these joins is to connect the customer table to itself by the CustID column. The leeway given in selecting a meaning has been used here to eliminate a special case which has no value.

#### 5.3.3.4.4 Avoiding Combinatorial Explosion

##### 5.3.3.4.4.1 Value to node mappings

Another problem that can arise in searches is combinatorial explosion. This is where the number of alternatives becomes very large through the effect of making a number of non-deterministic choices. In this context, the problem can arise when mapping values to database graph nodes (as in Figure 5.13). For the test database, "1" has over 30 mappings. If the user enters "TKB Sport Shop bought 1 ace tennis net", there would be three values to connect but the "1" value has more than one possible mapping to a database graph node. The mapping of these values to database graph nodes is shown in Figure 5.27. Only three

of the mappings of "1" are shown.

Value	Database Graph Nodes			
	Table	Column	Row	Value
TKB Sport Shop	CUSTOMER	NAME	2	TKB Sport Shop
Ace Tennis Net	PRODUCT	DESCRIPT	5	Ace Tennis Net
1	ITEM	QUANTITY	1	1
		ITEMID	2	1
		QUANTITY	2	1

**Figure 5.27 - Value to Node Mapping**

By ordering the selection of which of these node to consider, the system can try to avoid considering the nodes for "1" until it has considered the nodes for the other two values. This will increase the number of constraints on the solution which may help reduce the number of alternative mappings of "1" that are kept. Producing the solutions in order of cost also helps to minimize this problem.

#### **5.3.3.4.5 Graph Isomorphism**

For general graphs, checking isomorphism is time consuming. In this case, the nodes of the graph are typed by the table and column. This means that the bijection that maps nodes in one graph to nodes in the other has an additional constraint. The matched nodes must be in the same table at the same column. There is only one such bijection between two graphs whenever a table is not joined in more than once. The number of bijections is no more than the product of the number of times each table is joined in. So if the employee table is

joined in twice and the department table once, then there are at most two times one bijections. In order to cause graph isomorphism to be time consuming, several tables would have to be joined together many times. However this would cause problems for the database as well.



### 5.3.3.4.6 Producing the Most Reasonable Tree First

**|: tkb sport shop paid 58 for ace tennis net.**

2 Interpretations. They Are:

- Option #1 "58" refers to ACTUALPRICE in the ITEM table
- Option #2 "58" refers to ITEMTOT in the ITEM table

You may select an option as a correct interpretation,  
ask for more options, or decide to stop this definition

**|: can you show me some more options.**

[moreOptions]

3 Interpretations (1 are new). They Are:

- Option #1 "58" refers to ACTUALPRICE in the ITEM table
- Option #2 "58" refers to ITEMTOT in the ITEM table
- Option #3 (New) "58" refers to STDPRICE in the PRICE table

You may select an option as a correct interpretation,  
ask for more options, or decide to stop this definition

**|: more.**

[moreOptions]

There are no more alternative interpretations.

All alternatives have been listed.3 Interpretations. They Are:

- Option #1 "58" refers to ACTUALPRICE in the ITEM table
- Option #2 "58" refers to ITEMTOT in the ITEM table
- Option #3 "58" refers to STDPRICE in the PRICE table

You may select an option as a correct interpretation,  
ask for more options, or decide to stop this definition

**|: i like option number 1.**

[pickAnOption(1)]

**Figure 5.28 - Presenting the User with Different Interpretations**

In order to seem intelligent to the user, the system will select only some of the alternative trees which can connect a set of values and present them to the user (Figure 5.28). The system presents a summary of the differences between the choices if there is more than one

choice, or a number of examples if there is only one choice. The user can confirm a choice or ask for more alternatives. The system uses the tree cost to determine the order that the solutions will be presented. The tree cost is most effective when the keys and foreign keys have been identified; however, the system does not require that they be identified. In fact they can even be incorrectly identified. A number of heuristics can be used by the system in order to determine keys and foreign keys.

#### **5.3.3.4.6.1 Recognizing Keys and Foreign Keys**

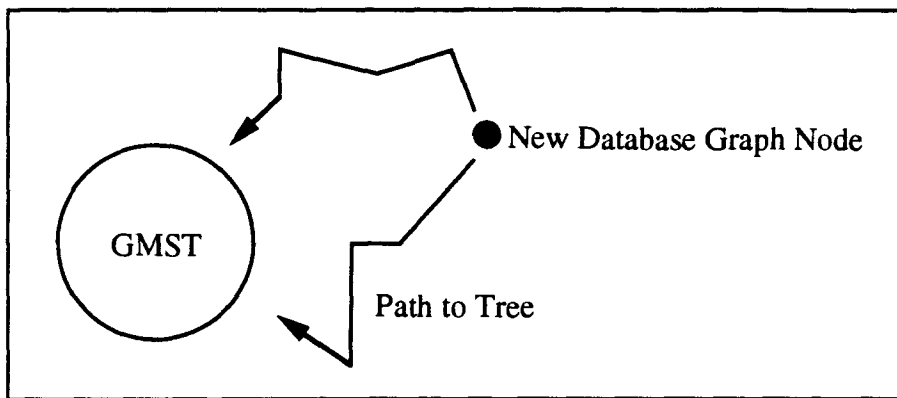
In some DBMS, such as DBase, joins can be specified directly in the system and so no investigation is necessary. Other DBMS provide means to directly define columns as being keys or foreign keys. Some systems, however, have no such facilities. For these systems some simple rules of thumb can be used.

Columns with a large number of unique values are probably keys. Columns with a large number of rows but a small number of values are probably foreign keys. The keys can be searched for the parent key of the foreign key. All of these tests can be done with simple SQL commands and remember that the system does not require that the analysis be correct. It should be noted that the system does not currently determine which columns are keys and which are foreign keys.

### 5.3.3.5 Improvements

There are a number of improvements that could be made to the algorithms. Some make them faster and some make them more general.

#### 5.3.3.5.1 Search Direction Choice

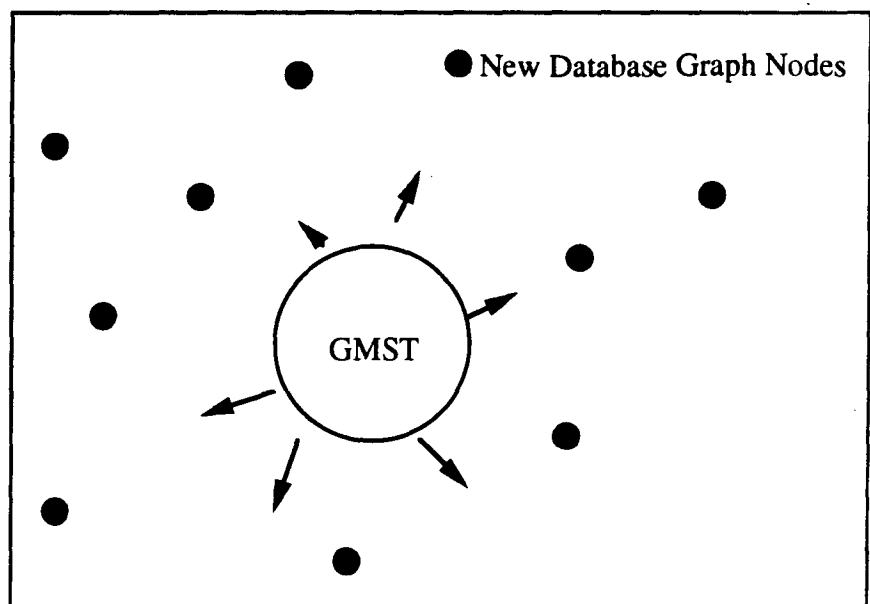


**Figure 5.29** - Search from Each New Node to GMST

A search occurs when a new node is to be connected to the best GMST. Starting at the new node, paths to the GMST are sought. This is

efficient when the number of new nodes is small. However if there are many new nodes it would be better to

search from the GMST to the new nodes. To do this the algorithm must choose the direction of search dynamically. The direction of search choice would be made using branching factors



**Figure 5.30** - Search from GMST to all New Nodes

in each direction. This could be estimated with the number of nodes in the GMST and the number of new nodes to be connected.

#### **5.3.3.5.2 More Sophisticated Joins**

This system allows only simple joins to occur. Single columns are joined using equality. It is possible to extend the system to multi-column joins that use only equality tests. This would be done on the join conditions of the 'same value' edges in the database graph. When such a join was desired, the additional nodes in the join condition would need to be added to the GMST. Other type of joins might be also required.

Tables might be joined by using calculated values or tests that are not equality. Another possibility is multi-column joins using only a subset of the columns of another multi-column join. Of course it is possible to imagine many strange ways of joining tables. Most of which are not realistic. No attempt was made to handle these types of joins because it was unclear how realistic they are.

#### **5.3.4 Ambiguity**

In some cases the user's example sentence can have a number of interpretations. For these, the system interacts with the user in order to determine which selection is intended. Unlike other systems which produce paraphrases of the interpretations, this system performs some analysis of the alternatives to determine what is the difference between them. The user is

only presented with descriptions which distinguish between the alternatives<sup>9</sup>. The analysis is performed using an identification tree which is a type of decision tree (WINSTON91).

A number of other approaches that would provide a better environment in which to resolve this ambiguity, are considered here. The first approach is to provide examples of the interpretation using data from the database. When there is only one interpretation the system does show examples but the user has no control over the selection of examples. Control over the example generation could be provided by allowing the user to ask questions using the different possible interpretations. Another nice feature would be to allow the user to ask questions about the objects mentioned in the examples. Providing information about how things are being connected would also be an interesting extension but some work would have to be done to determine a clear form for presentation. It would also be reasonable to allow the sophisticated user to ask for an SQL paraphrase of the interpretations. All of these features but the 'show how they are connected' one would be straightforward to implement.

## 5.4 Learning Word Meanings

Aside from learning frames the system also learns which words mark which frames or frame slots. The first way this learning occurs is obvious but the second is more subtle.

When the user presents example sentences such as "Smith works for Ford", "works" is

---

<sup>9</sup> for an example see Figure 5.28.

recognized as a verb marking the "work" case frame. For the sentence "Smith's salary is 800", the system learns that the word "salary" marks the "salary" attribute slot of the employee object frame. The system also learns type information about "Ford" which used to mark the object frame. It is obvious that the learning of words is occurring here.

The second place where word meanings is learned is from user questions. When the user asks "Which employees work for Ford", the system has learned that "employee"<sup>10</sup> marks the subject slot of the "work" case frame. When the user asks "Which employee's salary is 800", the system has learned that the word "employee" marks the "employee" object frame. There are probably many other places where unintrusive learning can occur. This is but one which was investigated.

## 5.5 Summary

This chapter described the spanning tree approach to database configuration. By viewing the database as a graph, called the database graph, relationships among elements in the database can be found by using straightforward spanning tree algorithms derived from graph theory. The minimal spanning tree is used to represent these relationships through its structure. It was found that the minimal spanning tree represented information redundantly and so the generalized minimal spanning tree was defined to eliminate this problem. In addition to presenting a pseudo-code algorithm for calculating the generalized minimal spanning trees a detailed example was presented. The chapter concluded with notes about

---

<sup>10</sup> - The morpher is used

implementing an efficient version of the algorithm.

# Chapter 6

## Question Answering

The tree structures described in the previous chapter are used during question answering to interpret sentences. During the process of question answering, the system can also learn the meanings of some words. This section describes this process through an example.

### 6.1 Example

The example proceeds by examining the process of analyzing a question using the sentence in Figure 6.1. The stages of processing are described according to the data flow diagram of Figure 3.2.

Who earns over the average salary of an employee working in research?

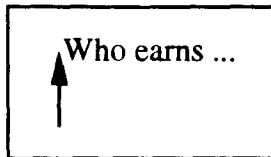
**Figure 6.1 - Example Sentence**

#### 6.1.1 Tokenizing

The first stage of analysis of a sentence is the stage where the sentence is broken down into tokens (Tokenizer - process a of Figure 3.2). The lexicon (inverted index) is accessed in order to determine which sequences of characters should be treated as a single token. The basic idea is to select prefixes of the input string which appear in the lexicon. If a

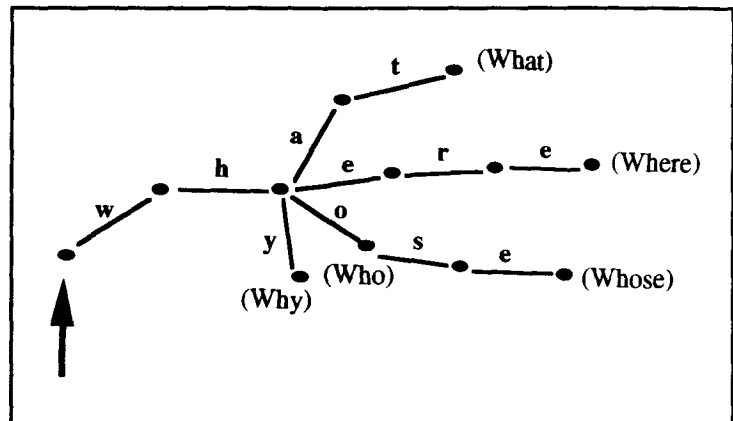


prefix does not appear in the lexicon then spaces and punctuation are used as delimiters in order to create a token. A search is conducted



**Figure 6.2** - Initial Position in the Sentence

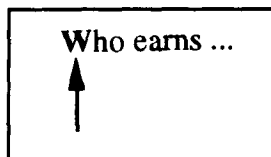
through the index for prefixes of the sentence. Figure 6.3



**Figure 6.3** - Initial Position in the TRIE

contains the inverted index in the form of a TRIE. The initial

position (at the root) is marked by the arrow. Figure 6.2 contains the sentence that is being tokenized. The initial position is before the first letter of the sentence.

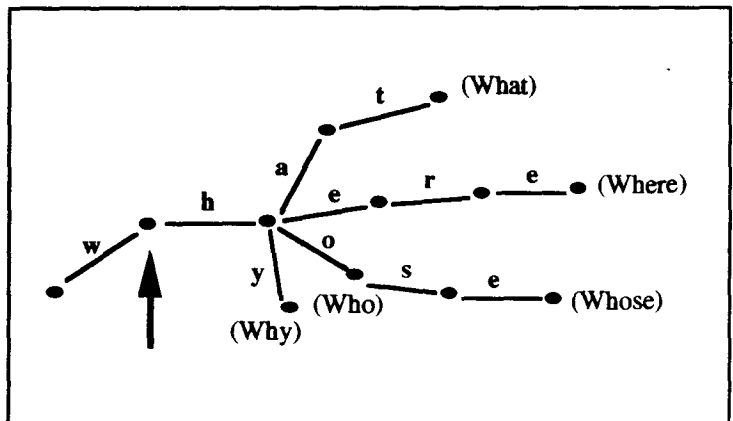


**Figure 6.4** - Sentence

The next step is to move one position to

the left in the sentence

(Figure 6.4). The letter<sup>11</sup> that is traversed (in this case 'w') is used



**Figure 6.5** - Inverse Index

to select the arc in the TRIE to traverse (Figure 6.5).

<sup>11</sup> - not case sensitive



Token	Interpretation Alternatives
"WHO"	<ul style="list-style-type: none"> <li>questionWord</li> </ul>
"EARNs"	
"OVER"	<ul style="list-style-type: none"> <li>rangeMarker(compare(&gt;))</li> <li>preposition(over)</li> </ul>
"THE"	<ul style="list-style-type: none"> <li>article(definite)</li> </ul>
"AVERAGE"	<ul style="list-style-type: none"> <li>groupFunctionModifier("AVG")</li> </ul>
"SALARY"	<ul style="list-style-type: none"> <li>attribute</li> </ul>
"OF"	<ul style="list-style-type: none"> <li>preposition(of)</li> </ul>
"AN"	<ul style="list-style-type: none"> <li>article(indefinite)</li> </ul>
"EMPLOYEE"	<ul style="list-style-type: none"> <li>word not in index is delimited by spaces</li> </ul>
"WORKING"	
"IN"	<ul style="list-style-type: none"> <li>preposition(in)</li> </ul>
"RESEARCH"	<ul style="list-style-type: none"> <li>node(location(DEPT,DNAME,00000250.0001.0004),value(1,row(RES EARCH)),cost(other))</li> </ul>
"."	

**Figure 6.10 - Tokens of the Sentence**

Figure 6.10 presents the only tokenization that is produced. Some information about the interpretation of words is associated with the tokens at this point. This information is derived from the lexicon entry. Each alternative interpretation in the table is marked by a bullet. The meanings listed in the table will be briefly described. The word "Who" is marked as a questionWord which tells the system that the referents of the word are to be listed in the response. "Earns" has no interpretation at this point. The morpher was to be added to the indexing process but this was not done and so morphing occurs later.

"Over" has two different interpretations. For one interpretation, "over" is merely a

preposition that marks a case of some verb. This is indicated by preposition(over) in the meaning. For the other interpretation, "over" marks a range of values. In this case, the range is salaries greater than the average salary of an employee working in research.

"Over", "more than", "under" and "less than" are treated as markers for ranges of values. In the lexicon, they are marked as rangeMarkers/1. The compare/1 argument contains the type of comparison that is to be used. For "over" the comparison is greater than (>).

"The" and later "an" are marked as articles. This is part of the domain independent knowledge of the system. "Average" is one of a number of group function words which includes "total", "highest", "lowest" and "how many". Group function words correspond to functions performed on sets of values. The group functions that are interpreted by this system correspond directly to the SQL group functions (AVERAGE, COUNT, MIN, MAX, and TOTAL). The lexicon marks these words with the groupFunctionModifier/1 predicate. The argument of the predicate is the name of the SQL function that is to be used to do the calculations. In Figure 6.10, the word "average" is a group function word which is calculated by the SQL AVG function.

"Of" is marked in the lexicon as a preposition. Prepositions, as discussed earlier, serve as markers for cases of case frames. "Employee" is a word that is not in the lexicon. The system decides that "employee" is a word because it is delimited by spaces. For this sentence, the system later learns that "employee" is a noun which marks the subject case for the case frame of "work" and marks the object frame for employees. The lexicon will be updated to reflect this. "Working" is not in the lexicon directly but through morphing

the system determines that "working" marks a case frame. "In" is in the lexicon marked as a preposition.

The last word, "research", is a word that is in a field of the database. For each field that the value appears in, there is a node/3 predicate in the lexicon entry for the word. In this case, "research" appears in only one position in the database and so there is only one node/3 predicate in the lexicon entry of "research". The first term of the node/3 is the location/3 term of the node. location/3 contains the table, column and row address of the position that contains the word "research", ie - location(TABLE, COLUMN, ROW\_ADDRESS<sup>12</sup>). The second term is the value/2 term. The value/2 term contains information about the value - the data type and the actual data value, ie - value( Type, Value ). There are two options for the data value - raw(RawValue) where RawValue is the data in the database, or null which marks a null value. It is possible that through morphing or spell checking<sup>13</sup> that the word in the sentence would not be the same as the RawValue in the node/3 predicate.

### 6.1.2 Parsing

Parsing breaks the sentence up into phrases and produces restrictions on the structure of the sentence. The parser (process b of 3.3) is implemented using definite clause grammars<sup>14</sup> (DCG). A description of definite clause grammars can be found in PERWARR80, and

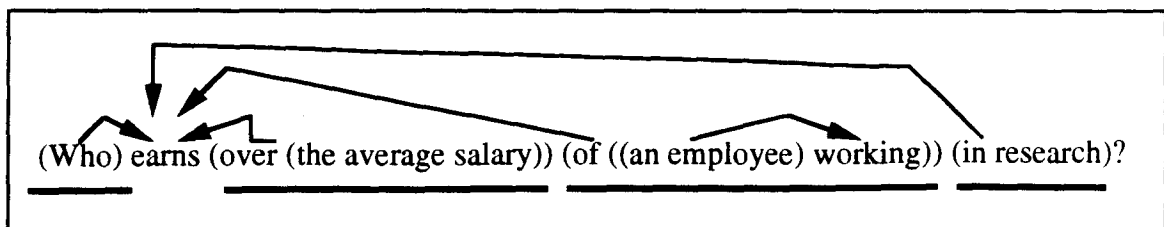
---

<sup>12</sup> - Row address is determined by ORACLE

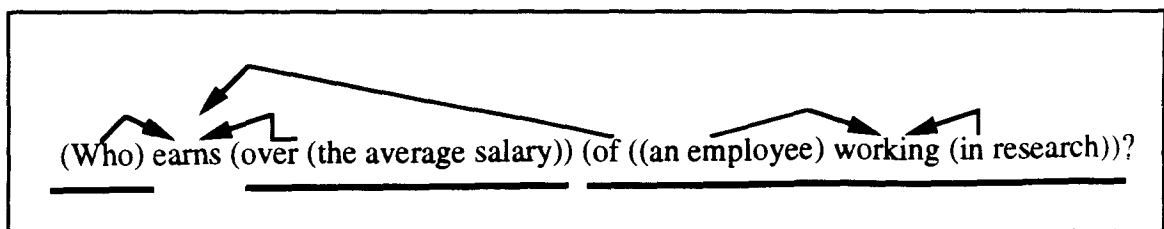
<sup>13</sup> - Not Implemented

<sup>14</sup> - Listed in appendix B

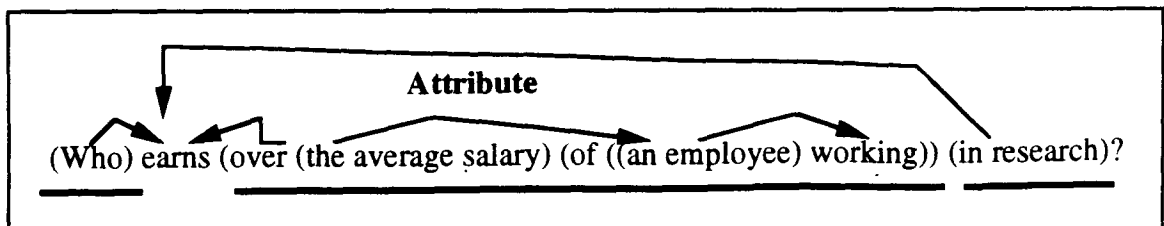
details on how to implement DCG based parsers can be found in ALLEN87. The following figures (Figure 6.11, Figure 6.12, Figure 6.13 and Figure 6.14) illustrate four different parses that could be produced if only syntactic information is used. Semantic information is used to reject non-meaningful parses and so there is only one parse that will be accepted for interpretation<sup>15</sup>. Note Figure 6.13 and Figure 6.14 which show "the average salary" as being associated with "employee" as an attribute. This is the result of the semantic grammar productions which were described in section 4.2.1.7.



**Figure 6.11 - Possible Sentence Structure**



**Figure 6.12 - Possible Syntactic Structure**



**Figure 6.13 - Possible Syntactic Structure**

<sup>15</sup> - Shown in Figure 6.14

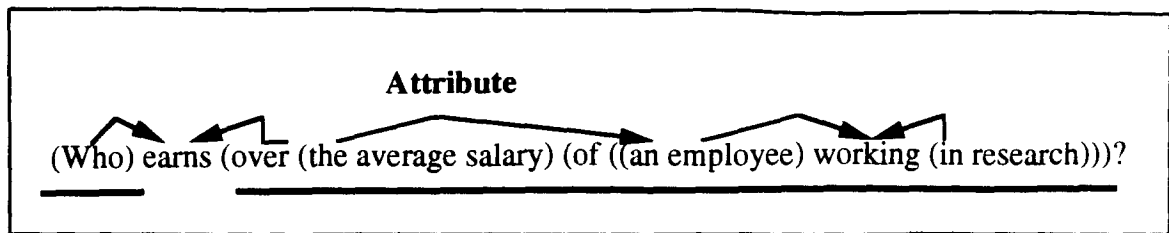


Figure 6.14 - Possible Syntactic Structure

Case Frame Marker: verbPhrase(2,auxiliaries([]),verb("EARNs",[]))	
Case Marker	Case Filler
1	nounPhrase(1,"WHO",[questionWord],[],[question])
2	range("OVER",compare(>),nounPhrase(3,"SALARY",[attribute],[groupFunctionModifier("AVG")],[definite]))
Case Frame Marker: verbPhrase(5,auxiliaries([]),verb("WORKING",[]))	
Case Marker	Case Filler
1	nounPhrase(4,"EMPLOYEE",[],[],[indefinite])
prep("IN")	nounPhrase(6,"RESEARCH",[node(location(DEPT,DNAME,00000250.0001.0004),value(1,raw(RESEARCH)),cost(other))],[],[[]])
Case Frame Marker: nounPhrase(4,"EMPLOYEE",[],[],[indefinite])	
Case Marker	Case Filler
Attribute	nounPhrase(3,"SALARY",[attribute],[groupFunctionModifier("AVG")],[definite])

Figure 6.15 - Cases of the Only Parse Accepted

Figure 6.15 presents detailed information on the parse that is finally accepted for interpretation. In this table, there are three frame markers. The first two markers are case frame markers which are represented by

verbPhrase( Position, Auxiliaries, Verb )

Figure 6.16 - verbPhrase/3

verbPhrase/3 predicates (Figure 6.16). The first term (Position) of this predicate is a

number indicating the position of the case frame marker in the input sentence. The position is not based upon the position of the word in the sequence of words but rather the position of the phrase in the sequence of phrases. In the example, the first verb phrase is in the second position after "who" and the second verb phrase is in the fifth position after "who", "earns", "over the average salary", and "of an employee". The second term of the verbPhrase/3 predicate is a list of the auxiliary verbs in the verb phrase. Currently this information is only used for detecting passives. The last term in the verbPhrase/3 predicate is the main verb information. This is stored in the verb/2 predicate. The first term of this predicate contains the verb found in the sentence and the second term is not used.

The last frame marker in the table is the nounPhrase/5

nounPhrase(Position, Noun, Meanings, Adjectives, Quantifiers)
---

**Figure 6.17 - nounPhrase/5**

predicate (Figure 6.17). The

first term is the position of the phrase in the sentence as described earlier. The second term is the head noun as it appears in the sentence. The third term, Meanings, is a list of different interpretations of the noun. This list could indicate that the noun appears in the database (using the node/3 predicate described), that the noun marks an object frame, or that the noun marks a case of some frame. No interpretation of the word "employee" is currently known and so the list of interpretations for employee is empty. The fourth term of nounPhrase/5, Adjectives, is a list of adjectives which were applied to the noun. The only adjectives allowed are group function modifiers such as "average", or "total". The last term contains quantifier information. The only quantifiers currently allowed are definite, indefinite, or questionWord markers.



Figure 6.15 also contains information about the cases recognized for each of the frames.

For the first case frame (for "earns"), there are two cases which are both marked by position. The subject is represented by the nounPhrase/5 for "who". The object is filled by a range/3 predicate. The range/3 predicate is used to mark a range of values. In this case, the range of values are defined using a comparison ('>') to a value, the nounPhrase/3 for salary. Range/3 phrases arise when there are rangeMarker/1 words in the sentence.

The second case frame (for "working") also has two cases. The first case which is marked by position is filled by the nounPhrase/5 for "employee". The second case is marked by the preposition "in". This case is filled by the nounPhrase/5 for "research". The last frame, which is marked by "employee", has only one case, "salary". Salary is an attribute of the object frame which is marked by the word "employee". In this case, the nounPhrase/5 for salary has information that indicates that the average of salary is to be used. The information in Figure 6.15 is passed along to the semantic interpreter (process (c) of 3.3).

### 6.1.3 Semantic Analysis

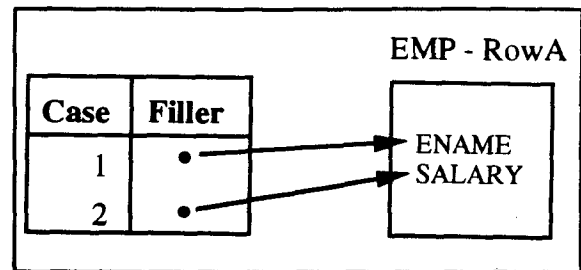
The semantic analysis stage of processing (process c of 3.3) begins after parsing. This section describes how this analysis proceeds for questions. The analysis of declarative sentences was described earlier in chapter 5.

For each parse that is produced, the system attempts an interpretation. Interpretation involves selecting frames for the frame markers and combining the results into a number of

interrelated GMSTs. This section illustrate this process for the one successful interpretation of the example sentence.

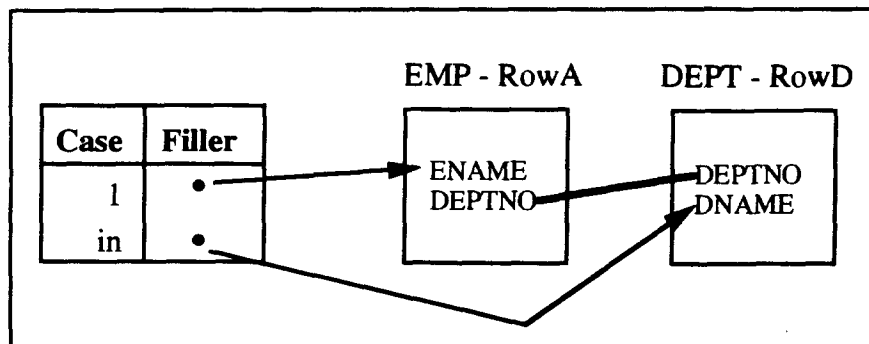
### 6.1.3.1 Applying the Frames

For the example, assume that the frames in Figure 6.18, Figure 6.19, and Figure 6.20 are in the dictionary. The case frame of Figure 6.18 could have been produced using the sentence, "Smith earns 800". The case



**Figure 6.18 - Case Frame for Earn**

frame of Figure 6.19 could have been produced with "Ford works in research". The frame in Figure 6.20 could have been produced from two sentences such as "Smith's manager is

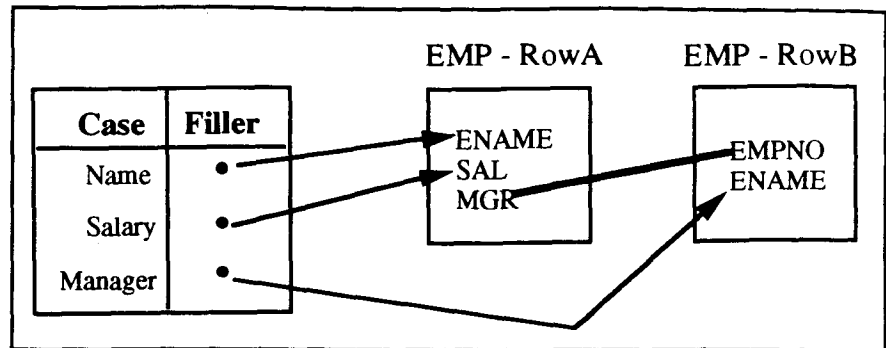


**Figure 6.19 - Case Frame for Work**

Ford" and "The salary of smith is 800".

These frames are different from the frames which were illustrated in earlier chapters because they contain the GMST data structure which is linked to the case information. The earlier diagrams did not contain the GMST data structure because when the diagrams were presented the GMSTs had not yet been defined. For these diagrams, the case fillers (nodes in the GMST) are marked by an arrow from the filler entry to the corresponding node. This

information provides  
the means to connect  
the components of the  
sentence to the GMST.

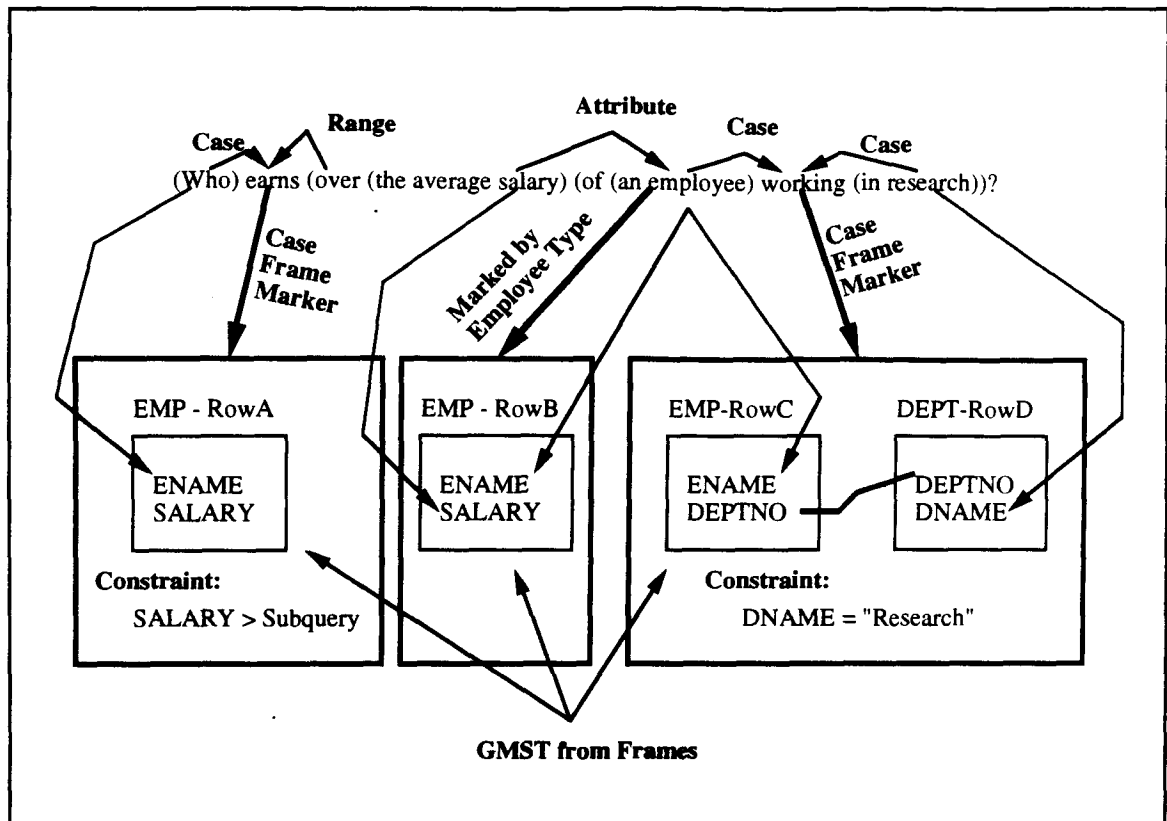


**Figure 6.20 - Object Frame for Employee**

The first step in the

process is to analyze the sentence using the information in the frames. This involves selecting a frame for each of the frame markers and then ensuring that the cases for the frame are compatible with the frame (Figure 6.21). Consider "earns". The systems knows that "earns" marks the "earn" case frame of Figure 6.18 so this frame is used for interpretation. Next, the case fillers must be considered. According to the parse information in the table of Figure 6.15, there are two case fillers for "earns". The first one is "who" which fills the subject case. According to the case frame information in Figure 6.18, the subject, "Who", is mapped to the ENAME node of the database graph. "Who" is allowed to refer to any type of node so this case filler is allowed by this case frame (indicated in Figure 6.21 by an arrow from "Who" in the sentence to the ENAME column in the GMST). The next filler is for the object case. According to the parse information (Figure 6.15), the object of "earns" is a comparison marked by "over". The system in this case does some type checking in order to determine compatibility of the case and the filler. The type information used is merely the data type of the filler (text, number or date). Only things of the same type may be compared (eg - numbers to numbers). More sophisticated type checking can be applied also (described later on in this section for the case marker "working"). For the object case, there is no filler in the sentence. The filler is left

unspecified. Instead a constraint is used. In this case, the SALARY node which is the object case, must be greater than "the average salary". This is represented in Figure 6.21 by the constraint SALARY > SALARY SUBQUERY. After determining the case fillers of the case frame the case to filler table is discarded.



**Figure 6.21 - Selecting Frames**

The application of the object frame for employees presents an interesting variation in the analysis. The mapping of sentence elements to the case frame occurs as just described. In this case however, the system does not yet know that the word "employee" marks the case frame for employees. The case frame for employees is selected using the knowledge that "salary" marks a case of the employee case frame. Additional support for this decision is that "employee" is also filling the subject frame of "working". In order to do this,

"employee" must refer to something in the ENAME column of the EMP table. But this is exactly what the object name in the object frame for employee (Figure 6.20) must refer to. This is an example of the second form of type information. The first form is the data type (text, number, or date) and the second form is the table and column that the data occurs in. The third form was to be the object frame and the fourth form was to be cases filled but these were not implemented. Arrows are used in Figure 6.21 to indicate that "salary" refers to SALARY column and "employee" refers to the ENAME column.

There is another wrinkle in the analysis. Notice that the GMST for the frame in Figure 6.20 contains more nodes than the object frame used in the interpretations for "employee" shown in Figure 6.21. The GMST used to interpret "the average salary of an employee" (Figure 6.21) is smaller than the GMST in the case frame (Figure 6.20). This happens because the system uses the  $GMST_1$  (relative to the cases present in the sentence) of the  $GMST_2$  in the frame. The difference is that the  $GMST_2$  is minimum relative to all of the cases mention in the case list for the frame. In Figure 6.20, there are three cases mentioned (the employee name, salary and manager) so  $GMST_2$  must span nodes for all of these value. In the sentence (Figure 6.21), however, only the employee name and salary are mentioned so the system selects the subset of the GMST of the frame ( $GMST_2$ ) that spans the values mentioned in the sentence. This amounts to using the GMST of the GMST.

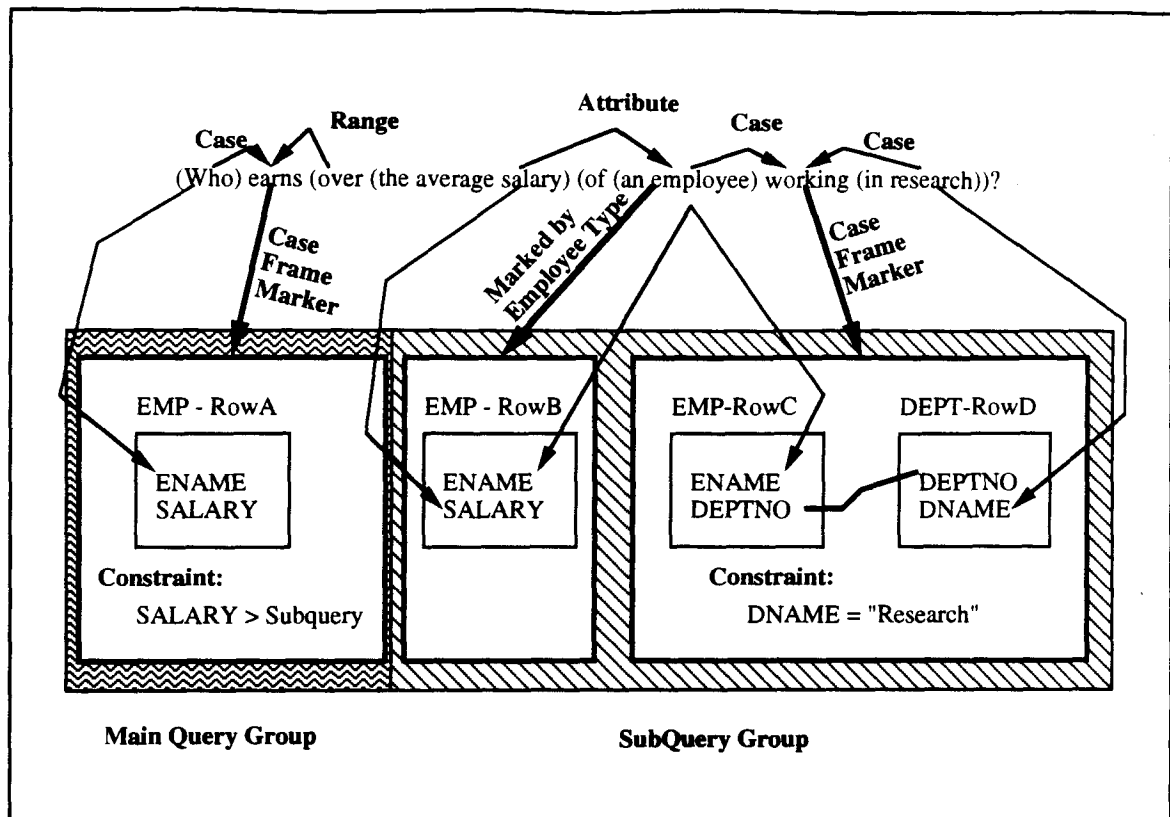
"Working" is the last case marker to be analyzed. The analysis occurs as before. The system accesses the case frame information for work (Figure 6.19). The case to case filler table is used to associate fillers from the parse table (Figure 6.15) with nodes in the GMST.

Again, the type information is used to verify that correct decisions are being made. The interpretation of "working" illustrates what occurs when the sentence contains phrases that occur in the database. For this example, "research" occurs in the DEPT table. To handle this, a constraint is added to the constraint list (Figure 6.21). The constraint is that the node corresponding to the appropriate case (DNAME) in the GMST must be equal to the value mentioned in the sentence ("research"), ie - DNAME = "Research". A better approach would be to constrain the case filler (DNAME) to be one of the nodes referred to by the word in the sentence ("research"). This would allow the system to function properly for words spelt incorrectly or morphological variations of words.

#### **6.1.3.2 Combining the Frames**

After using frame information to interpret the components of the sentence, the system combines the interpretations into one interpretation consisting of a number of GMSTs and constraints. There is one GMST for each group of GMSTs which have a node modified by a group function adjective. After determining how to group the GMSTs, the GMSTs in each group are combined into one GMST.

The first step of combining the GMSTs into one GMST is to form groups of GMSTs. The groups are determined based upon the words modified by group function adjectives (eg - "average", or "total"). For this example, there are two groups (Figure 6.22). One group contains the GMST and constraints for the "earns" case frame (main query group), and the other group contains the GMST and constraints for the "employee" case frame and the



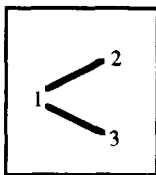
**Figure 6.22 - Query Groups**

"working" case frame (sub-query group).

For each sub-query group there is a phrase with a group function adjective in the sentence. In this example, there is only one phrase with a group function adjective ("the average salary"). Corresponding to each phrase with a group function adjective ("the average salary") there is a node in a GMST, (the SALARY node in the EMP table - Figure 6.22). Other GMST are added to the group containing this node by considering reachability from this node. In this case nodes are neighbours if they are reachable from within the same GMST or if there is a phrase in the sentence that refers to both nodes (the ENAME node in the EMP-RowB table and the ENAME node in the EMP-RowC table are both referred to by the word "employee" and so one is reachable from the other). This definition of

reachability implies that the GMSTs for "employee" and for "working" are connected and so they belong to the sub-query group for the "the average salary" phrase (diagonal lines box of Figure 6.22). All of the GMST that are not put in a sub-query group in this manner are put in the main query group (wiggly lines in Figure 6.22).

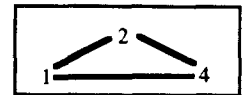
After the trees are divided into groups, the trees in each group are combined separately. A



**Figure 6.23**

- Graph 1

special case of graph union is used to combine them. In general, graph union is performed by combining the sets of nodes of the graphs and the sets of edges of the graph.

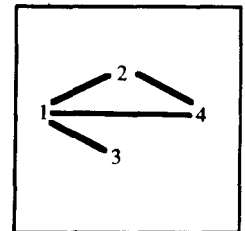


**Figure 6.24**

- Graph 2

The graph in Figure 6.25 is the result of combining the graphs in Figure 6.23 and Figure 6.24. In these graphs, each node is represented by a number.

Nodes with the same number are the same nodes. Node 2 in Figure 6.23 is the same node as node 2 in Figure 6.24 and they are represented by node 2 in Figure 6.25. Union of the GMST occurs in this manner with one exception.



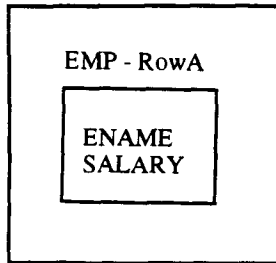
**Figure 6.25**

- Union

The exception is that the system must determine which nodes in the GMSTs should be treated as the same nodes. In the example of graph union, the labelling of the nodes was used to immediately determine which nodes were the same. For GMSTs, the system must determine which nodes should be the same. This is done by considering the relation between the phrases in the sentence and the nodes in the GMSTs. Consider the phrase, "an employee". This phrase refers to two different database graph nodes in two different GMSTs (Figure 6.22). Because one phrase refers to both these database graph nodes, the



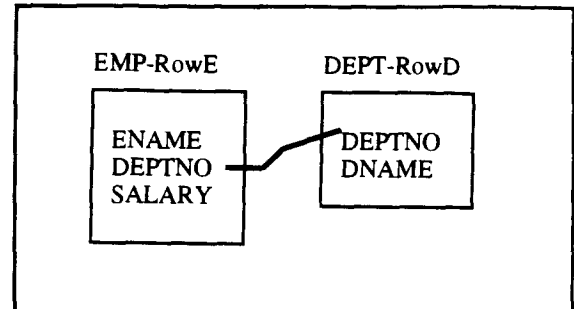
system decides that they should become the same database graph node. In order to make them the same database graph node, the symbolic row values which are used to distinguish



**Figure 6.26** - Main Query GMST

to different occurrences of a GMST node (RowB and RowC) are replaced by one

symbolic row value (in



**Figure 6.27** - Sub-query GMST

this case RowE). As a result the two GMST

nodes (EMP - RowB and EMP - RowC) become one. The result of combining the GMSTs

for the "average salary" sub-query group (Figure 6.22) is shown in Figure 6.27. The

EMP - RowB and EMP - RowC GMST nodes have been combined into one node marked by EMP - RowE<sup>16</sup>. This is the only way that nodes are combined during this analysis. By

combining nodes in this manner, the system can determine that an interpretation is incorrect and the SQL that is generated can be more efficient. No work needs to be done to the main query group because there is only one GMST in it. The GMST for the main query is shown in Figure 6.26.

### 6.1.3.3 Producing the SQL Query

After the groups are combined, the SQL query can be formulated. Each group is converted separately to an SQL query. Consider the sub-query group corresponding to the phrase, "the average salary of an employee working in research" (Figure 6.27). For each GMST node

<sup>16</sup> - Arbitrarily chosen unique value

(identified by a table and row) mentioned in the GMST there is a table alias in the SQL (see FROM list of Figure 6.28). For this case, there are two GMST nodes, EMP-RowE and DEPT-RowD. Each of the symbolic rows (RowE and RowD) are mapped to a unique number. This number is appended to the table name to form an alias for the table. In this case RowE is mapped to zero and RowD is mapped to zero. The resulting aliases are DEPT0 and EMP0.

```
SELECT AVG(EMP0.SAL)
FROM DEPT DEPT0, EMP EMP0
WHERE DEPT0.DNAME = 'RESEARCH' AND DEPT0.DEPTNO = EMP0.DEPTNO
```

**Figure 6.28 - SQL for the Sub-query**

During the analysis of the sentence, a number of constraints arose. For the sub-query group, there was only one constraint, DNAME = "Research" (Figure 6.22). This constraint is placed in the WHERE list of the SQL. Notice that the alias corresponding to the correct GMST node is used to identify the DNAME column properly (Figure 6.28). The join conditions are also put into the WHERE list of the SQL. Remember from chapter 5 that the join edges in the GMST represent the join condition of two tables. For this system, the join condition is that the nodes connected by the "join" edge must have the same value. In the GMST for the sub-query (Figure 6.27), there is only one "join" edge which connects the DEPTNO column of the EMP table to the DEPTNO column of the DEPT table. In the SQL, the join condition is represented by the constraint that DEPT0.DEPTNO = EMP0.DEPTNO. Notice that the correct table aliases have been applied.

The last component of the SQL query to describe is the SELECTION list. For sub-queries, the column(s) (SAL) that the function word modifies is placed in the SELECTION list. For this case, "average" modifies "salary" and so EMP0.SAL is placed in the SELECTION list with the appropriate SQL function (Figure 6.28).

The process of producing the SQL for the main query is essentially the same as for the sub-query. There is one exception. During the analysis of the sentence, a list of nodes that are to be printed as an answer is produced (the list contains all words that are marked as questionWords<sup>17</sup>). In this case, the list only contains the node referred to by "Who". This list is used when producing the main query. All of the nodes in the list are put in the SELECTION list of the main query. As with the sub-query, the constraints and join conditions are placed in the WHERE clause. For this main query, there is a comparison that references the sub-query. The SQL for the sub-query is inserted in the appropriate place at this time. The final form of the SQL for the main query is shown in Figure 6.29.

```
SELECT EMP0.ENAME
FROM EMP EMP0
WHERE EMP0.SAL >
  any (SELECT AVG(EMP0.SAL)
        FROM DEPT DEPT0, EMP EMP0
        WHERE DEPT0.DNAME = 'RESEARCH' AND DEPT0.DEPTNO = EMP0.DEPTNO)
```

**Figure 6.29 - SQL for the Main Query**

The query is sent to Oracle for processing and the results are printed by the system.

---

<sup>17</sup> - see section 6.1

## 6.2 Response Generation

There is no sophisticated response generation. Tuples that satisfy the SQL command are merely listed.

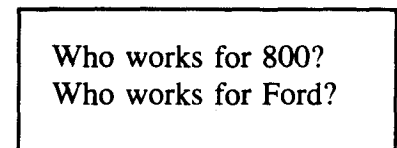
## 6.3 Learning Word Meanings

Word meanings can be learned when the system is interpreting questions. This was shown in the example for the word "employee". There are other types of words that the system can learn during question answering. For the sentence, "Which employees work in Research?", the system learns that "employee(s)" is an noun marking the subject case for the "Work" case frame. After the question "Which employee's salary is 800", the system knows that the object frame referenced in this sentence is marked by the noun "employee". Attribute names can be learnt as well. The system determines what the words refer to during sentence analysis as described earlier. If the system did not know that the word could have such a referent then this information is added to the lexicon for use later. Extensions to the system based upon exploiting this type learning are presented in the final chapter.

## 6.4 Handling Ambiguity

Type information is used by the system to resolve ambiguity. Currently type information is the ORACLE data type (numbers, dates or text), and the table and column that the data appears in. The object frames and the case markers were to be used as type information but this was not developed. Type information is used to mark object frames and can be used to resolve ambiguity when syntactic and case markers do not.

As an example of this, imagine that the works case frame has two cases marked by "for", one is a manager and the other is a salary. The type information can be used in Figure 6.30 to determine which case is appropriate. For the



**Figure 6.30** - Possible Ambiguity

first sentence, 800 is a number. Only the salary case marked by "for" can be filled by a number. For the second sentence, "Ford" is known to be in the ENAME column and to be text. There is only one case marked by the preposition "for" that allows such a filler. The type information has been used here to eliminate ambiguity. If the words "manager" and "salary" were known to mark the cases then these words could be used to eliminate ambiguity (implemented for object frames but not for case frames).

## 6.5 Controlling the Dialogue

The dialogue is maintained using a control structure patterned after an ATN. The dialogue manager does

not handle

ellipsis or

anaphora. It

only controls

the

presentation

and selection of alternatives. In some situations, a fixed number of responses from the user is expected. For these, the system uses a pattern matching approach to analyze the

sentences. This provides a more robust way of analyzing sentences when the number of

different interpretations is limited. At the following point in the dialogue (Figure 6.31),

pattern matching is used to interpret the

user's response. The pattern matcher will

accept a large number of sentences that

the parser will not (Figure 6.32).

```
There are no more alternative interpretations.
All alternatives have been listed.3 Interpretations. They
Are:
    Option #1
        "58" refers to ACTUALPRICE in the ITEM table
    Option #2
        "58" refers to ITEMTOT in the ITEM table
    Option #3
        "58" refers to STDPRICE in the PRICE table

You may select an option as a correct interpretation,
ask for more options, or decide to stop this definition
```

**Figure 6.31 - Asking for Input**

```
gimme 1.
1.
of these choices the only good one
is 1.
number 1.
numbr 1
no 1.
1's the best.
```

**Figure 6.32 - Selection Option #1**

## 6.6 Summary

The chapter described the use of spanning trees during the question answering process by presenting a detailed example. Tokenization of the question is performed taking advantage of the TRIE structure of the inverted index which contains the lexicon. The result of parsing is a number of cases and associated fillers which are analyzed by the semantic interpreter. Semantic analysis has two main stages. For the first stage, the spanning trees in the case frames are accessed and related to phrases in the sentence. The second stage is to combine the spanning trees. After the semantic analysis, the interpretation is converted to SQL and the results of the SQL query are presented to the user.

# **Chapter 7**

## **Conclusion**

### **7.1 Contributions of this Thesis**

This paper describes a method that can be used to obtain a higher degree of domain portability for natural language interfaces. Early natural language interfaces required a domain and natural language expert in order to be configured. Later systems increased domain portability by providing facilities that a DBA could use in order to configure the natural language interface. The approach presented in this paper produces a system that could be almost entirely configured by a naive user (no knowledge of linguistics or database management systems). This represents a significant increase in the portability of natural language interfaces. A number of benefits arise because each user configures the system for himself or herself.

The first benefit is for the DBA. One can imagine that when the DBA configures the interface, there is a great deal of work to be done. First, meetings must be held with all the users to determine which words they will want to use. These words must be compiled and defined in terms of the database. Problems will arise if different users have different



interpretations of the same word. This will require compromise on the part of the users or there will be words whose interpretation is ambiguous relative to the grammar of all the users. After the system is configured by the DBA, new users will want to use new words. Then the DBA will have to combine these new words with the old and iron out any conflicts. A number of these problems can be avoided when the users configure the system for themselves.

Each user will have his or her own grammar. This will mean that words that are ambiguous relative to everyone's grammar will not necessarily be ambiguous to a particular user's grammar. As a result, the user will not be asked to select from alternative interpretations when relative to the user's grammar there is only one interpretation. This will also increase the processing speed of the system by reducing the size of the dictionaries. If the users discover that they have not configured the system for a word that they want to use, then configuration can be done immediately without having to wait for the DBA to get around to doing it. For the DBA, the main advantage is that very little time need be spent helping the users configure the systems. This can amount to a great reduction in work if you consider that, for some applications, users are only allowed to access the database through a set of customized views. In this case, the NLI would have to be set up once for each of the users' different views of the database. This could be a time consuming process depending on how different the views are and how much is transferrable between them.

A program was written to demonstrate that the method of configuration described in this paper is effective. The program was not developed to be a sophisticated natural language

interface. The system has a rudimentary question answering facility whose purpose was to show that the system had, in fact, correctly configured itself. When looking at the program demonstrations, one should not ask whether or not the program is a sophisticated natural language interface but rather one should ask does the program adequately demonstrate that the approach described in this paper is effective. We believe so.

Although the system was not developed to the point of being a sophisticated natural language interface, it could serve as a basis for one. The remainder of this chapter describes enhancements that could be made to the system in order to produce a powerful NLI. These enhancements include descriptions on improvements to the methods used to configure the system.

## **7.2 Possible Enhancements**

The enhancements presented here are not described in great detail. The purpose is to provide brief notes indicative of straightforward directions for improvements.

### **7.2.1 Enhancements to Noun Phrase Interpretation**

#### **7.2.1.1 Adjectives**

The system currently does not handle adjectives. This section describes a number of special cases that could be added to the system.

#### **7.2.1.1.1 Related Adjective - "red ford"**

Sometimes adjectives are nouns that refer to something related to the noun being modified. For "red Ford", red is the color of a Ford. The relation could be represented as a GMST in the object frame for "ford". Alternatively, inference could be performed using the object frames and case frames to find a connection between "red" and "ford" that already exists. For example, "red" could be the color attribute in the object frame for "Ford".

#### **7.2.1.1.2 Explicitly Defined - "Rich employees earn over 20000"**

Another type of adjective is one that is defined explicitly by the user. The approaches used to handle other problems in the system could be used to handle this one. The first step after getting a sentence that is used to define an adjective, would be to produce the GMST for the sentence. The adjective whose meaning is not known could be ignored. After the GMST is produced, the adjective ("rich") would be defined by noting that it ("rich") referred to the same database graph node that is referred to by the noun ("employee") it modified. Later when the adjective is used in a sentence, the system could access the GMST associated with the adjective. This GMST would be combined with the GMST for the head noun of the sentence using the GMST union<sup>18</sup> operation. For this union, the node referred to by the adjective in the adjective's GMST would be set to be the same node as the node referred to by the head noun in the head noun's GMST. Thus, application of the meaning of the adjective would occur through graph union.

---

<sup>18</sup> - see section 6.1.3.2

#### **7.2.1.1.3 Anaphora - "part number A37"**

Anaphora is another form of expression employed by users that should be handled. For anaphora, a noun that marks an object frame and a noun referring to something in the database are listed one after the other. The first noun could be interpreted as a marker for an object frame (or a case of a case frame). The second noun could then be interpreted using the marked frame.

#### **7.2.1.2 Semantic Grammar Categories**

In this system, the only semantic categories that were used are attribute, and object. The system's performance could be improved greatly if more semantic categories were used. Names is one example of a useful category. Sometimes objects have a number of names (first, middle or last name). Names can also be abbreviated (I.B.M.). Object can also have more than one distinct name. In order to function effectively, a natural language system must be able to work with names. Another semantic category that is useful is units. Units have a hierarchy of different type of units. Examples of units include measurements (temperatures, distances, weights) and counts (currency). The TEAM system provides a good source for useful semantic categories. Once new categories are defined, the grammar could be extended to recognize them.

### **7.2.1.3 Pronouns**

The system would be enhanced greatly by being able to handle pronouns and more complicated forms of anaphora.

### **7.2.1.4 Words not in the Database**

Words that are not in the database are not handled properly by the system. The system should be extended to handle this in order to provide intelligible feedback to questions involving unknown words.

### **7.2.1.5 Learning Object Class**

An object class is a semantically defined set of objects. This is commonly represented in systems using the isa relation. In order to handle a wide range of questions the system should be extended to cover isa relations. This section describes the different ways that the isa hierarchy can be implicitly represented in the database.

#### **7.2.1.5.1 Column Membership**

Case fillers and objects must be represented by values that appear in the same columns. For example, for "Smith works in research", the subject is allowed to be any word in the ENAME column of the EMP table. This works well for most cases. However, there are

several ways of representing data in the database which this will not work for.

Sometimes classes of objects are marked by constraints on other columns. For example, in Figure 7.1, the clerks are objects in the EMP table which have JOB = "Clerk". This is not an unreasonable way to represent data but the system as it is set up will not handle this.

EMP						
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800	20
7876	ADAMS	CLERK	7788	12-JAN-83	1100	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	20

**Figure 7.1 - Job Class**

#### 7.2.1.5.2 Groups of Columns - (Job1, Job2, ... )

Another way of representing groups of objects is by having a number of columns with the same prefix and a different number for a suffix. For

S.I.N.	Name	Job1	Job2	Job3
989-776-090	Jones	MGR776	PGR700	
334-987-909	Smith	CLK223		

**Figure 7.2 - Another Form of Classes**

examples one could imagine a table where the employees could have up to three jobs and the different jobs would be listed in columns called JOB1, JOB2, and JOB3 (Figure 7.2).

The program set up does not recognize groups such as these.

### **7.2.1.5.3 Through Joins and Restriction**

Another way of representing groups is by using table joins to restrict the values selected.

This is closely related to the discussion in section 7.2.1.5.1 for the representation of clerks.

In this case, a table join must be performed first before the value that defines the group can be accessed.

### **7.2.1.5.4 Information contained in file name or paths**

For some systems, filenames and directory paths are used to encode information which the system must access in order to function properly. For example, data for different years could be stored in different directories or filenames could be used to encode the name of different sites from which data is collected. Access to the directory paths and file names is sometimes necessary.

### **7.2.1.6 Plural Nouns**

The system does not handle plural nouns. Extending the system to handle them is relatively straightforward since the morpher will provide information about whether or not a noun is plural.

## **7.2.2 Learning Grammatical Structures**

An interesting extension to the system would be to have it learn semantic grammar rules. The semantic grammar rule itself could be used as a marker for a case frame that the system learned. The meaning of the semantic grammar rule could be determined by using the spanning tree approach. The grammar could be extended to include the new semantic grammar rule and the corresponding spanning tree.

### **7.2.2.1 "Voltage across part is 29" instead of Attribute of Object is Value**

The system currently does not allow prepositional phrases (other than 'of' phrases) to modify a noun directly. Adding them would be relatively straightforward. The object frame could be used to store information about prepositional phrases that are allowed to modify each object. The parser would have to be extended to allow such modifier structure to be recognized.

## **7.2.3 Enhancements to Verb Phrase Interpretation**

### **7.2.3.1 Employer/Employee**

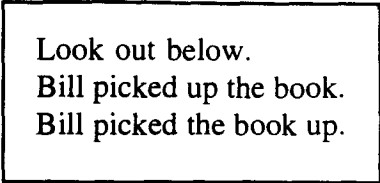
A simple improvement to the system would be to calculate employer and employee from the verb employ. The GMST from employ could be used and the word (employer or employee) could be set to refer to one of the database graph nodes in the GMST. Then the



GMST could be combined with the rest of GMSTs of the sentence as described earlier.

### 7.2.3.2 Particles

Sometimes verbs can be combined with a particle to yield a verb with a new meaning. The role of particle is filled by prepositions. When used as particles, preposition do not mark cases of the verb but instead are part of the verb.



Look out below.  
Bill picked up the book.  
Bill picked the book up.

**Figure 7.3** - Preposition used as Particles

Figure 7.3 contains examples of sentences where the preposition is used as a particle.

Notice that the particle immediately follows the verb or the object of the verb. The system does not currently handle particles.

### 7.2.4 Enhancements of the Parser

The parser does not take advantage of the fact that the system is based on case frames. A parser that was constructed to analyze sentences using the case frames directly might be more robust and flexible (HAYMOU80, and TAYROS75).

### 7.2.5 Enhancements Using Learning

It is possible to set the system up so that it can learn the meaning of words from sentences that do not directly reference the database. For example, the system could learn the meaning of "earns" from the sentence "Smith's manager earns 3000". This can be done

because the algorithms for calculating GMSTs expect a list of nodes that are referred to by the values in the sentence. In order to handle these type of sentences, the step that calculates the mapping of the values in the sentence to the database graph nodes must be extended.

#### **7.2.5.1 Correcting things that were learned incorrectly**

There is currently no way to correct things that have been learned incorrectly. Handling this elegantly is a non-trivial problem.

#### **7.2.6 Enhancements Using Frames**

##### **7.2.6.1 Meta-Questions - "What is in the database"**

For meta-questions such as "What is in the database?" or "Tell me about employees", the frames that the system has could be used to provide a reasonable answer.

##### **7.2.6.2 Report Frames**

The system currently responds to questions. This is not enough. A facility for dealing with reports should be added. Frames could be used to keep track of different types of reports and different instances of reports created by the user. The report frames could be used to answer meta-question or to help the system understand how the user conceptualizes the

domain. The second use would be interesting to investigate.

## **7.2.7 Enhancements to the Database Interface**

### **7.2.7.1 What if underlying database structures are changed?**

One problem, not dealt with here, is what happens when the underlying database structures are changed (eg - new tables, new columns, deletion of tables). Is there a way to automate the conversion of the system?

## **7.2.8 Inferences**

Adding inference to the system would enhance the performance of the system in a number of ways. This section describes a few.

### **7.2.8.1 Filling in the Blanks - "average salary in research"**

Sometime when people speak they leave information out of the sentence when they believe it can be reasonably inferred. For example, one might ask for "the average salary in research" and expect the system to infer that one is talking about "the average salary of an employee who works in research". This could be performed by using the frames that the system has to determine connections among the frames (or cases) that are mentioned.

#### **7.2.8.2 Ellipsis**

It was hoped that by having many types of information in the system and by using semantic grammars that the system would be able to handle ellipsis as effectively as semantic grammars of the 1970's. Unfortunately, no work was done on ellipsis in this system.

### **7.3 Summary**

The goal of this thesis is to provide a means to increase the portability of natural language interfaces. This has been achieved by using the spanning tree approach. A system was implemented to demonstrate the effectiveness of this configuration method. The system allows a naive end user to configure the interface in a simple and straightforward way. There is much room left for development of this approach to configuration as well as to the program itself.

## Bibliography

**ALLEN87** Allen, J., *Natural language understanding*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California 1987.

**BERWICK85** Berwick, R., *The Acquisition of Syntactic Knowledge*, MIT Press, Cambridge, MA, 1985

**BOOTH83** Booth, D. A., *Designing a Portable Natural Language Database Interface*, M.Sc. Thesis, University of British Columbia, Vancouver, 1983.

**BRADLEY82** Bradley, J., *File and data base techniques*, CBS College Publishing, New York, NY, 1982.

**BROWSE77** Roger Browse, *A Knowledge Identification Phase of Natural Language Analysis*, M.Sc. Thesis, University of British Columbia, Vancouver, 1977.

**BUBR79** Burton, R. R. and Brown, J. S., "Toward a Natural-Language Capability for Computer-Assisted Instruction," reprinted in *Readings in Natural Language Processing*, editors Grosz, B. J., Sparck Jones, K., and Webber, B. L., Morgan Kaufmann Publishers, 1986 pp 605-626.

**CARHAY83** Carbonell, J. G., and Hayes, P. J., "Recovery Strategies for Parsing Extragrammatical Language," *American Journal of Computational Linguistics*, Vol 9, No 3-4, 1983.

**COLBY71** Colby, K., Weber, S., and Hilf, F., "Artificial Paranoia," *Artif. Intell* 2, 1, pp 1-25 (1971)

**DATE86** Date, C.J., *An introduction to database systems*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1986.

**FILLMORE71** Fillmore, C., "Types of lexical information," in *Semantics: An Interdisciplinary Reader*, Steinber and Jakobovits, eds., Cambridge University Press, London, 1971

**HARRIS77** Harris, L. R., "User oriented data base query with the ROBOT natural language query system," *Int. J. Man-Machine Studies* 9, 1977, pp 697-713

**HARRIS84** Harris, L. R., "Experience with INTELLECT," *The AI Magazine*, Vol. V, No. 2, 1984

**HAYMOU80** Hayes, P.J., and Mouradian, G.V., "Flexible Parsing.", in *Proceedings of 18<sup>th</sup> Annual Meeting of the Assoc. for Comput. Ling*, Philadelphia, June 1980, pp 97-103.

**HOBBS78** Hobbs, J. R., "Resolving Pronoun References," in *Readings in Natural Language Processing*, editors Grosz, B. J., Sparck Jones, K., and Webber, B. L., Morgan Kaufmann Publishers, Inc. 1986, pp 339-352

**KAPLAN84** Kaplan, S. J., "Designing a Portable Natural Language Database Query System," *ACM Transactions on Database Systems*, Vol 9, No. 1, March 1984, Pages 1-19.

**LEHMAN89** Lehman, J. F., *Adaptive Parsing: Self-extending Natural Language Interfaces*, Kluwer Academic Publishers, MA, 1989

**LIFER77** Hendrix, G., Sacerdoti, E., Sagalowicz, D., and Slocam, J., "Developing a Natural Language Interface to Complex Data," *ACM Trans. Database Sys*, #(2), 1978, pp 105-147

**MAP83** Martin, P., Appelt, D., and Pereira, F., "Transportability and Generality in a Natural-Language Interface System," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany*, Los Altos:William Kaufmann Inc., 1983, pp 573-581.

**MARCUS80** Marcus, M., *A Theory of Syntactic Recognition for Natural Language*, MIT Press, Cambridge, MA, 1980

**PERWARR80** Pereira, F., Warren, D., "Definite Clause Grammars for Language Analysis", *Artificial Intelligence 13*, North-Holland Publishing Company, 1980

**PG87** Perrault, C. R., and Grosz, B. J., "Natural Language Interfaces" in *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, editor, Shrobe, H. E., Morgan Kaufmann Publishers, 1987, pp 133-172

**PLANES76** Waltz, D. L., Finin, T., Green, F., Conrad, F., Goodman, B., and Hadden, G., "The Planes System: Natural Language Access to a Large Data Base," *Technical Report T-34*, University of Illinois, Urbana Illinois, November 1976

**SEDGE90** Sedgewick, R., *Algorithms in C*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1990

**TARJAN83** Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

**TAYROS75** Taylor, B. H., and Rosenberg, R. S., "A Case-Driven Parser for Natural Language", *American Journal for Computational Linguistics*, AJCL Microfiche 31.

**THOMP85** Thompson, B. H., and Thompson, F. B., "ASK is Transportable in Half a Dozen Ways," *ACM Trans. Off. Inf. Syst.*, 3, 2 (April 1985), pp 185-203

**WEIZEN66** Weizenbaum, J., "ELIZA - A Computer Program For the Study of Natural Language Communication Between Man and Machine," *Communications of the ACM*, Vol 9, No 1, January 1966

**WEIZEN67** Weizenbaum, J., "Contextual Understanding by Computers," *Communications of the ACM*, Vol 10, No 8, August 1967

**WHITE85** White, S.J., *A Portable Natural Language Database Query System*, M.Sc. Thesis, University of British Columbia, Vancouver, 1985.

**WINOGRAD75** Winograd, T., *Understanding Natural Language*, Academic Press, New York, NY, 1972.

**WINOGRAD83** Winograd, T. *Language as a Cognitive Process. Vol 1: Syntax.*, Addison-Wesley Reading, MA, 1983

**WINSTON91** Winston, P. H., *Artificial intelligence 3rd ed.*, Addison-Wesley Pub. Co., Reading, MA, 1992

**WOODS70** Woods, W. A., "Transition Network Grammers for Natural Language Analysis," *Commun. ACM*, 13, pp 591-602

**WOODS85** Woods, W. A., "Semantics and Quantification in Natural Language Question Answering," reprinted in *Readings in Natural Language Processing*, editors Grosz, B. J., Sparck Jones, K., and Webber, B. L., Morgan Kaufmann Publishers, 1985.



# Appendices

## A Relational Database Terminology

This appendix briefly describes some aspects of relational database theory. For a detailed discussion see DATE86.

"A relational database is a database that is perceived by its users as a collection of tables (and nothing but tables)"<sup>19</sup>. A table is a data structure made up of columns each of which has a name. A table also contains a number of rows of data. Two tables are shown in Figure A1.1.

---

DEPT		
DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS

EMP						
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800	20
7876	ADAMS	CLERK	7788	12-JAN-83	1100	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	20

---

**Figure A1.1 - Oracle Sample Database**

---

<sup>19</sup> - CODD86, page 96

The department table contains information about departments. It has three columns - DEPTNO, DNAME, and LOC. A value that uniquely identifies each department is stored in the DEPTNO column. The name of the department is contained in the DNAME column and the city that the department is located in is stored in the LOC column. Each row contains information describing a single department.

The employee table contains information about each employee. It has eight columns which are called EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, and DEPTNO. As with the department table, there is a column (EMPNO) which contains values that uniquely identify each employee. The name of each employee is stored in the ENAME columns. The employee's job description is in the JOB column. The MGR column contains the EMPNO of the manager of the employee. The HIREDATE, and SAL columns contain the date of hiring, and salary of the employee, respectively. The last column, DEPTNO, contains the department number of the employee. Each row in the employee table contains information about one employee.

Under the relational model, information is not only derived directly from each table but also from the relationships that exists among tables. Consider the employee table. There is a column that contains the department number of the employee's department (DEPTNO) but the employee table does not contain the department name. The name can be accessed by looking in the department table at the row with the corresponding department number. For example, the first employee, SMITH, is in department number 20. By looking up department number, 20 in the DEPT table, the department name can be found, ie -

RESEARCH. This approach of connecting rows in different tables by matching columns values is a common method of accessing information in relational databases. This method of connecting tables is called a (natural) *join*.

Key columns are important in order to join tables effectively. The *key* of a table is a column or number of columns that are guaranteed to contain a unique value or combination of values for each row in a table. For example, the EMPNO column of the employee table is a key. Note that the relational model does not require that each table have a key. When a column of a table can contain only values that are in the key of another table than this column is called a *foreign key*. For example, the DEPTNO column of the employee table is a foreign key of the employee table because all of the values in the column are in the DEPTNO key column of the department table.

A number of other important concepts associated with the relational database model can be found in DATE86.

## B Grammar

This appendix contains a listing of the grammar taken directly from the source code. The grammar is implemented using the DCG feature of Quintus Prolog.

```

sentence(parseInfo(PIIn,PIOut), CaseMappings) -->
  noiseWords, {PI1 = PIIn, CM1 = CM1T-CM1T},
  present(parseInfo(PI1,PI2), CM2 ),
  {
    parseInfo_send( position, PI2, SubjPosition, PI3 ),
    parseInfo_send( position, PI3, VPPosition, PI4 ),
    parseInfo_send( mood, PI4, question, PI5 )
  },
  unmarkedCase(parseInfo(PI5,PI6),Case,CM3),
  terminator(parseInfo(PI6,PI7)),
  { % fake a what is <np> question
    VerbPhrase =
      verbPhrase(VPPosition, auxiliaries([]), verb( "BE", [] )),
    parseInfo_send( marker(VPPosition), PI7, Marker1, PI8 ),
    CM4 = [noun(SubjPosition,"WHAT",[questionWord],[],[question])-(Marker1-VerbPhrase)|T4]-T4,
    case_object( Case, Object ),
    parseInfo_send( marker(VPPosition), PI8, Marker2, PI9 ),
    parseInfo_send( mainVerb, PI9, VerbPhrase, POut ),
    CM5 = [Object-(Marker2-VerbPhrase)|T5]-T5,
    diffList_appends( [CM1,CM2,CM3,CM4,CM5], CaseMappings )
  }.

sentence(parseInfo(PIIn,PIOut), H-T) -->
  noiseWords,
  {PI1 = PIIn},
  unmarkedCase(parseInfo(PI1, PI2), Case, L0), % What does questions
  {parseInfo_send(level(carry), PI2, Case, PI3)},
  {
    auxilliary(parseInfo(PI3,PI4)),
    {
      parseInfo_send( mood, PI4, question, PI5 ),
      parseInfo_send( mainVerbDefined, PI5, yes, PI6 )
    },
    unmarkedCase(parseInfo(PI6,PI7), Subject, L1),
    vp(parseInfo(PI7,PI8), VerbPhrase)
  },
  {
    parseInfo_send( level(takeCarry), PI3, Subject, PI4 ),
    L1 = L1T-L1T
  },
  vp(parseInfo(PI4,PI8), VerbPhrase)
},
{
  {
    {
      verbPhrase_position( VerbPhrase, VPPos ),
      parseInfo_get( voice(VPPos), PI8, Voice ),
      {
        Voice = passive
        -> parseInfo_send( marker(VPPos), PI8, _, PI10 ); % Skip Subject
        Voice = possiblePassive
        -> {
          parseInfo_send( voice( VPPos ), PI8, passive, PI9 ),
          parseInfo_send( marker(VPPos), PI9, _, PI10 ); % Skip Subject
          PI10 = PI8
        };
        PI10 = PI8
      },
      parseInfo_send( marker(VPPos), PI10, Marker, PI11 ),
      case_object( Subject, SObject ),
      L2 = [SObject-(Marker-VerbPhrase)|L2T]-L2T
    }
  }
}

```

```

    ),
    unmarkedObjects(parseInfo(PI11,PI12), VerbPhrase, L3 ),
    markedObjects(parseInfo(PI12,PI13), VerbPhrase, L4 ),
    terminator(parseInfo(PI13,PI14)),
    {
      {
        parseInfo_send( level(takeCarry), PI14, Carry, PI15 )
        --> {
          verbPhrase_position( VerbPhrase, VPPos ),
          parseInfo_send( marker(VPPos), PI15, ObjMarker, PI16 ),
          {
            verbPhrase_verb( VerbPhrase, Verb ),
            (Verb="BE"; Verb="HAVE"; Verb="DO")
            --> ObjMarker < 3;
            true
          },
          case_object( Carry, CarryObject ),
          L5 = [CarryObject-(ObjMarker-VerbPhrase)]L5T]-L5T
        );
        L5 = L5T-L5T, PI16 = PI14
      },
      diffList_appends( [L0, L1, L2, L3, L4, L5], H-T ),
      parseInfo_send( mainVerb, PI16, VerbPhrase, PI17 ),
      PIOut = PI17
    }
  ).

terminator( parseInfo(PIIn,PIOut) ) -->
  ["."-_ _], !, {PIOut = PIIn} |
  [{"?"-_ _], !, {parseInfo_send( mood, PIIn, question, PIOut )}}.

auxilliary(parseInfo(PIIn,PIOut)) -->
  % ----- MODAL -----
  [_Modal-ModalAlts-],
  {
    memberchk( auxilliary(modal, _ _ _), ModalAlts ),
    parseInfo_send( auxilliary(ModalAlts), PIIn, PIOut )
  }
  |
  % ----- HAVE -----
  [_Have-HaveAlts-],
  {
    memberchk( auxilliary(have, _ _ _ , tense(_HaveTense)), HaveAlts ),
    parseInfo_send( auxilliary(HaveAlts), PIIn, PIOut )
  }
  |
  % ----- BE 1 -----
  [_-BelAlts-],
  {
    memberchk( auxilliary(be, _ _ _ , tense(_BelTense)), BelAlts ),
    parseInfo_send( level(up(be)), PIIn, PI1 ),
    parseInfo_send( auxilliary(BelAlts), PI1, PIOut )
  },
  !
  |
  % ----- DO -----
  [_-DoAlts-],
  {
    memberchk( auxilliary(do, _ _ _ , tense(_DoTense)), DoAlts ),
    parseInfo_send( auxilliary(DoAlts), PIIn, PIOut )
  }.

vp(parseInfo(PIIn,PIOut), verbPhrase(Position, auxilliaries(Auxs), verb( Verb, VerbInfo )) )
-->
  % ----- MODAL -----
  (
    [_Modal-ModalAlts-],
    {
      memberchk( auxilliary(modal, _ _ _), ModalAlts ),
      parseInfo_send( auxilliary(ModalAlts), PIIn, PI1 )
    }
  )
  |
  [], {PI1 = PIIn}
),
  % ----- HAVE -----
  (
    [_Have-HaveAlts-],
    {
      memberchk( auxilliary(have, _ _ _ , tense(_HaveTense)), HaveAlts ),
      parseInfo_send( auxilliary(HaveAlts), PI1, PI2 )
    }
  )

```

```

    }
    |
    [], {PI2 = PI1}
),
% ----- BE 1 -----
(
    [_-Be1Alts-],
    {
        memberchk( auxilliary(be, _, _, tense(_Be1Tense)), Be1Alts ),
        parseInfo_send( level(up(be)), PI2, PI3 ),
        parseInfo_send( auxilliary(Be1Alts), PI3, PI4 )
    }
    |
    [], {PI4=PI2}
),
% ----- BE 2 -----
(
    [_-Be2Alts-],
    {
        memberchk( auxilliary(be, _, _, tense(_Be2Tense)), Be2Alts ),
        parseInfo_send( level(up(be)), PI4, PI5 ),
        parseInfo_send( auxilliary(Be2Alts), PI5, PI6 )
    }
    |
    [], {PI6=PI4}
),
verb(Verb, VerbInfo),
(
    parseInfo_send( position, PI6, Position, PI7 ),
    (
        parseInfo_get( level(be), PI7, BeCount )
        -> true; BeCount = 0
    ),
    parseInfo_get( auxilliarities, PI7, Auxs ),
    (
        BeCount > 0
        -> (
            BeCount = 2
            -> parseInfo_send( voice(Position), PI7, passive, PI8 )
            ;
            (
                (
                    morph_irregular( Verb, [_Root-MorphInfo|Tail] )
                    -> true;
                    morph_roots( Verb, [_Root-MorphInfo|Tail] )
                ),
                Tail = [],
                member( tense(Tenses), MorphInfo )
            )
            -> (
                memberchk( pastpart, Tenses )
                -> parseInfo_send( voice(Position), PI7, passive, PI8 );
                PI8 = PI7
            );
            parseInfo_send( voice(Position), PI7, possiblePassive, PI8 )
        )
    );
    PI8 = PI7
),
(
    % Example sentences do not need the main verb defined but
    % questions do
    parseInfo_get( mainVerbDefined, PI8, _ )
    -> (
        Verb = "BE"
        -> Roots = ["BE"-[]];
        Verb = "HAVE"
        -> Roots = ["HAVE"-[]];
        Verb = "DO"
        -> Roots = ["DO"-[]];
        r_get( caseFrameDictionary, entry( Verb ), _ )
        -> Roots = [Verb-[]];
        morph_irregular( Verb, Roots ),
        someChk( Root_ in Roots,
            r_get( caseFrameDictionary, entry( Root ), _ ) )
        -> true;
        morph_roots( Verb, Roots ),
        someChk( Root_ in Roots,
            r_get( caseFrameDictionary, entry( Root ), _ ) )
        -> true
    );
    true

```

```

),
(
  % some simple tense checking stuff for the main verb
  Auxs = [],
  \+ parseInfo_get( level(embedded), PI8, _ ),
  (
    r_get( caseFrameDictionary, entry( Verb ), _ )
    -> Roots = [Verb-[]];
    morph_irregular( Verb, Roots ),
    someChk( Root-_ in Roots,
      r_get( caseFrameDictionary, entry( Root ), _ ) )
    -> true;
    morph_roots( Verb, Roots ),
    someChk( Root-_ in Roots,
      r_get( caseFrameDictionary, entry( Root ), _ ) )
    -> true
  )
  -> (
    mapList( _-MorphInfo in Roots,
      (
        memberchk( tense(Tenses), MorphInfo ),
        (member( prespart, Tenses );member( pastpart, Tenses ))
      )
    )
    -> fail;
    true
  );
  true
),
(
  % For Reduced Relative Clauses
  PI9 = PI8
),
PIOut = PI9
).

objects(parseInfo(PIIn,PIOut), VerbPhrase, CaseMappings) -->
object(parseInfo(PIIn,PI1), Case, CM0),
{
  (
    verbPhrase_position( VerbPhrase, VPos ),
    (
      case_type( Case, marked )
      -> (
        PI2 = PI1,
        case_marker( Case, Marker1 ),
        (
          parseInfo_get( voice(VPos), PI1, passive ),
          Marker1 = prep("BY")
          -> Marker = 1;
          Marker = Marker1
        )
      );
      parseInfo_send( marker(VPos), PI1, Marker, PI2 )
    )
  ),
  case_object( Case, CObject ),
  CM1 = [CObject-(Marker-VerbPhrase)|CM1T]-CM1T,
  diffList_appends( [CM0, CM1, CM2], CaseMappings )
},
objects(parseInfo(PI2,PIOut), VerbPhrase, CM2);
[], {CaseMappings = T-T, PIIn=PIOut}.

markedObjects(parseInfo(PIIn,PIOut), VerbPhrase, CaseMappings) -->
markedCase(parseInfo(PIIn,PI1), Case, CM0),
{
  verbPhrase_position( VerbPhrase, VPos ),
  PI2 = PI1,
  case_marker( Case, Marker1 ),
  (
    parseInfo_get( voice(VPos), PI1, passive ),
    Marker1 = prep("BY")
    -> Marker = 1;
    Marker = Marker1
  ),
  case_object( Case, CObject ),
  CM1 = [CObject-(Marker-VerbPhrase)|CM1T]-CM1T,
  diffList_appends( [CM0, CM1, CM2], CaseMappings )
},
markedObjects(parseInfo(PI2,PIOut), VerbPhrase, CM2);
[], {CaseMappings = T-T, PIIn=PIOut}.

unmarkedObjects(parseInfo(PIIn,PIOut), VerbPhrase, CaseMappings) -->
unmarkedCase(parseInfo(PIIn,PI1), Case, CM0),

```



```

{
  verbPhrase_position( VerbPhrase, VPos ),
  parseInfo_send( marker(VPos), PI1, Marker, PI2 ),
  {
    verbPhrase_verb( VerbPhrase, Verb ),
    (Verb="BE"; Verb="HAVE"; Verb="DO")
    -> Marker < 3;
    true
  },
  case_object( Case, CObject ),
  CM1 = [CObject-(Marker-VerbPhrase)|CM1T]-CM1T,
  diffList_appends( [CM0, CM1, CM2], CaseMappings )
},
{
  {case_object( Case, range( _, _ , _ ) )}
  -> [], {CM2 = T-T, PIOut=PI2};
  unmarkedObjects(parseInfo(PI2,PIOut), VerbPhrase, CM2)
};
[], {CaseMappings = T-T, PIIn=PIOut}.

object(parseInfo(PIIn,PIOut), Case, CaseMappings) -->
markedCase(parseInfo(PIIn,PIOut), Case, CaseMappings), !;
unmarkedCase(parseInfo(PIIn,PIOut), Case, CaseMappings).

unmarkedCase(parseInfo(PIIn,PIOut),
  case(unmarked, 'NA', NounInfo),
  CaseMappings )
--> nounPhrase(parseInfo(PIIn,PIOut), NounInfo, CaseMappings);
nounPhrase_range(parseInfo(PIIn,PIOut), NounInfo, CaseMappings).

markedCase( parseInfo(PIIn,PIOut), Case, CaseMappings ) -->
preposition(Preposition),
{
  {
    nounPhrase(parseInfo(PIIn,PIOut), NounInfo, CaseMappings),
    {
      Case = case(marked, prep(Preposition), NounInfo)
    }
  } |
  {
    nounPhrase_range(parseInfo(PIIn,PIOut), Range, CaseMappings),
    {
      Case = case(marked, prep(Preposition), Range)
    }
  } |
  {
    {
      nounPhrase(parseInfo(PIIn,PIOut), NounInfo, CaseMappings);
      nounPhrase_range(parseInfo(PIIn,PIOut), Range, CaseMappings)
    }
    -> {fail};
    {
      {
        parseInfo_send( level(takeCarry), PIIn, case(unmarked, _Marker, NounInfo ), PI1
      )
      -> {
        Case = case(marked,prep(Preposition),NounInfo)
      };
      PI1 = PIIn
    },
    PIOut = PI1,
    CaseMappings = T-T
  }
}
).

%
% Possessive = possessive then head noun must be possessive
% Possessive = anything else for not possessive
%

nounPhrase( parseInfo(PIIn,PIOut), HeadNounInfo, CaseMappings ) -->
questionWord( parseInfo(PIIn, PI1), HeadNounInfo ),
nounPhrase_qualifiers(parseInfo(PI1,PIOut),HeadNounInfo,CaseMappings)
|
nounPhrase_other( parseInfo(PIIn, PI1), NounInfo1, CM1 ),
{
  [possessiveMarker],
  nounPhrase_possessive( parseInfo(PI1,PIOut), NounInfo1, HeadNounInfo, CM2 ),
  {diffList_appends( [CM1, CM2], CaseMappings )};
}

```

```

    (HeadNounInfo = NounInfo1, CaseMappings = CM1, PIOut = PI1)
  ).
nounPhrase_qualifiers( parseInfo( PIIn, PIOut ), NounInfo, CaseMappings ) -->
  nounPhrase_relativeClause(parseInfo( PIIn, PIOut ), NounInfo, CaseMappings)
|
  (
    % Followed by a marked Case (OF for attributes)
    markedCase( parseInfo(PIIn,PI1), ObjectCase, CM1 ),
    {
      case_marker( ObjectCase, OMarker ),
      OMarker = prep("OF"),
      case_noun( ObjectCase, ObjectNoun ),
      CM2 = [NounInfo-(OMarker-ObjectNoun)|CM2T]-CM2T,
      %nounPhrase(position(OPos),OWord,OMeanings)|CM2T]-CM2T,
      diffList_append( [CM1, CM2], CaseMappings ),
      PIOut = PI1
    }
  )
|
  % Just not followed by anything directly related
  [], {CaseMappings = T-T, PIOut=PIIn}.
nounPhrase_relativeClause( parseInfo( PIIn, PIOut ), NounInfo, CaseMappings ) -->
  {
    parseInfo_send( push, PIIn, PI1 ),
    parseInfo_send( level(carry), PI1, case(unmarked, 'NA', NounInfo), PI2 )
  },
  {
    relativeIntro( parseInfo(PI2,PI3) )
    -> {true};
    {
      {
        % not quite right because plural form counts as
        % a determiner
        noun_determiner( NounInfo, [_] ),
        % Unmarked relative clause must be for
        % things with determiners
        parseInfo_send( level(embedded), PI2, yes, PI3 )
        %PI3 = PI2
      }
    },
    relativeClause( parseInfo(PI3,PI4), CaseMappings ),
    {
      parseInfo_send( pop, PI4, PIOut )
    }
  },
nounPhrase_other( parseInfo(PIIn,PIOut), NounInfo, CaseMappings ) -->
  {
    determiner(parseInfo(PIIn, PI1), _Determiner, Modifiers1, DeterminerInfo);
    [],{PI1=PIIn, DeterminerInfo = [], Modifiers1 = []}
  },
  { % Modifiers
    nounModifiers(parseInfo(PI1,PI2), Modifiers2, CM1 ),
    {append( Modifiers1, Modifiers2, Modifiers )}
  },
  {
    parseInfo_send( position, PI2, Position, PI3 ),
    NounInfo = noun( Position, Noun, Meanings, Modifiers, DeterminerInfo )
  },
  {
    commonNoun( not_possessive, parseInfo(PI3,PI5), Noun, Meanings );
    {
      {
        parseInfo_get( and, PI3, AndCount ),
        AndCount > 0
      }
      -> {
        attributes( parseInfo(PI3, PI4), Attributes, Meanings ),
        {
          Noun = Attributes,
          parseInfo_send( down(and), PI4, PI5 )
        }
      }
    }
  },
  nounPhrase_qualifiers( parseInfo( PI5, PIOut ), NounInfo, CM2 ),
  {
    diffList_append( CM1, CM2, CaseMappings )
  },
nounModifiers3( parseInfo(PI,PI), Modifiers, CM-CM ) -->

```

```

[ _Word-Meanings- _Lex],
{
  member(groupFunctionModifier(Mod), Meanings)
  -> Modifiers = [groupFunctionModifier(Mod)]
}
[], {Modifiers = []}.

nounModifiers( PI, Modifiers, CM-CM ) -->

  andList( PI,
    [Parse, TheWord, TheMeanings]-nounModifier(Parse, TheWord, TheMeanings),
    _Words, Modifiers )
;
  nounModifier( PI, _Word, Modifier ),
  {Modifiers = [Modifier]}
;
  [], {PI = parseInfo(P,P), Modifiers = []}.

nounModifier( parseInfo(PI,PI), Word, Modifier ) -->
[Word-Meanings- _Lex],
{
  member(groupFunctionModifier(Mod), Meanings)
  -> Modifier = groupFunctionModifier(Mod)
}.

andList(parseInfo(PIIn,PIOut), Component, Words, Meanings) -->
{
  parseInfo_get( and, PIIn, AndCount ),
  AndCount > 0,
  parseInfo_send( down(and), PIIn, PI1 )
},
andList( parseInfo(PI1,PIOut), Component, Words-[], Meanings-[], first ).

andList(parseInfo( PIIn, PIOut ),Component,Comp,Mean,First ) -->
["AND"-[]-[]], {\+ First = first}
-> {
  copy_term( Component, CompForm ),
  [parseInfo( PIIn, PIOut ),Comp1, Mean1]-CompCall = CompForm
},
CompCall,
{
  Comp = [Comp1|CT]-CT,
  Mean = [Mean1|M1]-M1
};
(["", "-_ _"] -> {true}; {true})
-> {
  copy_term( Component, CompForm ),
  [parseInfo( PIIn, PI1 ),Comp1, Mean1]-CompCall = CompForm
},
CompCall,
{
  diffList_append( [Comp1|C1T]-C1T, Comp2, Comp ),
  diffList_append( [Mean1|M1T]-M1T, Mean2, Mean )
},
andList(parseInfo( PI1, PIOut ), Component, Comp2, Mean2, notFirst).

%
% range = range( Marker, Compare, NounPhrase )
%
nounPhrase_range( ParseInfo, range(Marker,Compare,HeadNoun), CaseMappings ) -->
  rangeMarker( Marker, Compare ),
  nounPhrase( ParseInfo, HeadNoun, CaseMappings ).

rangeMarker( Marker, Compare ) -->
[Marker-WordMeanings- _LexInfo],
{
  memberchk( rangeMarker(Compare), WordMeanings )
}.

nounPhrase_possessive( parseInfo(PIIn,PIOut), LastPossessiveNoun,
  HeadNoun, CaseMappings ) -->
  % Possessives
  nounPhrase_other( parseInfo(PIIn,PI1), NounInfo, CM1 ),
  {
    CM2 = [NounInfo-(possessive-LastPossessiveNoun)|CM2T]-CM2T,
    diffList_appends( [CM1, CM2, CM3], CaseMappings )
  },
  {
    [possessiveMarker]
    -> nounPhrase_possessive( parseInfo( PI1, PIOut ),
      NounInfo, HeadNoun, CM3 );
  }

```

```

    {
        NounInfo = noun( IPosition, INoun, IMeanings, IMods, IDeterminerInfo ),
        HeadNoun = noun( IPosition, INoun, IMeanings, IMods, [possessive|IDeterminerInfo] ),

        CM3 = CM3T-CM3T,
        PIOut = PI1
    }
}.

determiner( parseInfo(PIIn,PIOut), Determiner, Modifiers, Info ) -->
    determiner_questionWord( parseInfo(PIIn,PI1), Determiner, Modifiers, Info )
    -> {parseInfo_send( mood, PI1, question, PIOut )};
    determiner_article( parseInfo(PIIn, PIOut), Determiner, Modifiers, Info ).

determiner_article( parseInfo( PI, PI ), Word, [], Info ) -->
    [Word--_],
    {
        atom_chars( Atom, Word ),
        article( Atom, Info )
    }.

article( 'A', [indefinite] ).
article( 'AN', [indefinite] ).
article( 'THE', [definite] ).

relativeIntro(parseInfo(PI,PI)) -->
    [Word--_],
    {
        atom_chars( Atom, Word ),
        relativeIntro( Atom )
    }.

relativeIntro( 'WHO' ).
relativeIntro( 'WHICH' ).
relativeIntro( 'THAT' ).

determiner_questionWord( parseInfo(PIIn,PIOut), Word, Modifiers, [question] ) -->
    [Word-WordMeanings-_],
    {
        member( determiner_questionWord(Modifiers), WordMeanings ),
        parseInfo_send( mood, PIIn, question, PIOut )
    }.

relativeClause(parseInfo(PIIn,PIOut), CaseMappings) -->
    {
        unmarkedCase(parseInfo(PIIn,PI1), Subject, L1),
        vp(parseInfo(PI1,PI2), VerbPhrase)
    };
    {parseInfo_send( level(takeCarry), PIIn, Subject, PI1 ), L1 = L1T-L1T},
    vp(parseInfo(PI1,PI2), VerbPhrase)
},
    {
        verbPhrase_verbInfo(VerbPhrase,VPInfo),
        \+ member( auxilliary(_,_,_,_), VPInfo )
    },
    {
        {
            {
                verbPhrase_position( VerbPhrase, VPPos ),
                parseInfo_get( voice(VPPos), PI2, Voice ),
                PI3 = PI2,
                {
                    Voice = passive
                    -> parseInfo_send( marker(VPPos), PI3, _, PI5 ); % Skip Subject
                    Voice = possiblePassive
                    -> {
                        parseInfo_send( voice( VPPos ), PI3, passive, PI4 ),
                        parseInfo_send( marker(VPPos), PI4, _, PI5 ); % Skip Subject
                        PI5 = PI2
                    };
                    PI5 = PI2
                },
                parseInfo_send( marker(VPPos), PI5, Marker, PI6 ),
                case_object( Subject, SObject ),
                L2 = [SObject-(Marker-VerbPhrase)|L2T]-L2T
            }
        },
        unmarkedObjects(parseInfo(PI6,PI7), VerbPhrase, L3),
        markedObjects(parseInfo(PI7,PI8), VerbPhrase, L4),
        {
            {

```

```

    parseInfo_send( level(takeCarry), PI8, Carry, PI9 )
    -> {
        verbPhrase_position( VerbPhrase, VPPos ),
        parseInfo_send( marker(VPPos), PI9, ObjMarker, PI10 ),
        case_object( Carry, CarryObject ),
        L5 = [CarryObject-(ObjMarker-VerbPhrase)|L5T]-L5T
    };
    L5 = L5T-L5T, PI10 = PI8
),
diffList_appends( [L1, L2, L3, L4, L5], CaseMappings ),
PIOut = PI10
)
).

questionWord( parseInfo(PIIn,PIOut),
    noun( Position, Word, [questionWord], [], [question] ) ) -->
[Word-WordMeanings-LexInfo],
{
    parseInfo_send( position, PIIn, Position, PI1 ),
    memberchk( questionWord, WordMeanings )
    -> parseInfo_send( mood, PI1, question, PIOut )
}.

% Meaning = attributes( ["Word", "Word", "Word", ... ], OfObjects )

attributes(PI, Attributes, attributes(Attributes, OfObjects)) -->
attributes1( PI, Attributes-[], OfObjects-[], first ).

attributes1(parseInfo( PIIn, PIOut ), Attributes, OfObjects, First ) -->
([",", "-_"] -> {true}; [{"AND", "-_"] -> {fail}; {true})))
-> attribute( parseInfo( PIIn, PI1 ), Attribute1, OfObjects1 ),
{
    diffList_append( [Attribute1|A1T]-A1T, Attributes2, Attributes ),
    diffList_append( [OfObjects1|O1T]-O1T, OfObjects2, OfObjects )
},
attributes1(parseInfo( PI1, PIOut ), Attributes2, OfObjects2, notFirst);
["AND"-[]-[]], {\+ First = first}
-> attribute( parseInfo( PIIn, PIOut ), Attribute, OfObjects1 ),
{
    Attributes = [Attribute|AT]-AT,
    OfObjects = [OfObjects1|O1]-O1
}.

attribute( parseInfo( PIIn, PIIn ), Word, OfObjects) -->
[Word-_-],
{
    (
        (
            r_get( attributeToObjectFrame_Dictionary, entry(Word), OfObjects1 )
            -> true;
            OfObjects1 = []
        ),
        (
            morph_roots( Word, Roots )
            -> mapSome( Root-_- in Roots, OfObjects2 in OfObjectss1,
                r_get( attributeToObjectFrame_Dictionary, entry(Root), OfObjects2 )
            );
            OfObjectss1 = []
        ),
        OfObjects = [_|_],
        ord_union( [OfObjects1 | OfObjectss1], OfObjects )
    ).
}.

commonNoun(Possessive, parseInfo(PI,PI), Word, WordMeanings) -->
[Word-WordMeanings-LexInfo],
{
    (
        Possessive = possessive
        -> memberchk( possessive, LexInfo );
        \+ memberchk( possessive, LexInfo )
    ),
    (
        memberchk( questionWord, WordMeanings )
        -> (
            member( questionWord, WordMeanings )
            -> fail;
            true
        );
        (
            (
                member( Alt, WordMeanings ),
                (

```

```

        functor( Alt, auxilliary, _ );
        functor( Alt, preposition, _ );
        Alt = groupFunctionModifier(_);
        Alt = determiner_questionWord(_);
        morph_roots( Word, Roots ),
        member( _Root-LexI, Roots ),
        (
            member( prespart, LexI );
            member( pastpart, LexI )
        )
    );
    (
        WordMeanings = [],
        \+ ( \+ ascii_isPunctuation(Word),
            atom_chars( Atom, Word ),
            \+ Atom = 'WHICH',
            \+ Atom = 'THAT',
            \+ Atom = 'MANY',
            \+ Atom = 'AND',
            \+ article( Atom, _ ),
            \+ member( preposition(_), WordMeanings ),
            \+ member( possessive, LexInfo ) )
        )
    ) -> fail;true
),
Word = [Ch|_], \+ is_punct( Ch )
}.

% jan 12 93
% january 12 93
% ? 12th
% year = 93 1993 none
% month jan 1-12 january
% day 1 1st
% separated by -, /, ' '
% order mmm dd yyyy      dd mmm yyyy
%date( ParseInfo, Date-[ ] ) -->
    %[Word],

verb(_Verb, _) --> preposition( _ ), !, {fail}.
verb(VerbOut, Info) -->
    [Verb-Info-LexInfo],
    (
        \+ ascii_isPunctuation(Verb),
        \+ ascii_isNumber(Verb),
        atom_chars( Atom, Verb ),
        \+ relativeIntro( Atom ),
        \+ determiner_questionWord( Atom ),
        \+ member( determiner_questionWord( _ ), Info ),
        \+ Atom = 'MANY',
        \+ Atom = 'AND',
        \+ article( Atom, _ ),
        \+ member( preposition(_), Info ),
        \+ member( possessive, LexInfo )
    ),
    (
        Info = [], VerbOut = Verb, !;
        member( VerbInfo, Info ),
        \+ auxilliary_modal( VerbInfo ),
        auxilliary_root( VerbInfo, VerbOut ),
        !
    ).

% auxilliary( RootAtom, RootWord, AtomForm, PerNo, Tense )

auxilliary_root(auxilliary(_RootAtom,RootWord,_Form,_PerNo,_Tense),RootWord ).
auxilliary_modal( auxilliary(modal, _RootWord, _AtomForm, _Type) ).

preposition( Preposition ) -->
    [Preposition-Meanings-_LexInfo],
    {member(preposition(_), Meanings)}.

sentence_yes -->
    [_-Meanings-], {memberchk( yesWord(_), Meanings )}
    |
    [_], sentence_yes.

sentence_no -->
    [_-Meanings-], {memberchk( noWord(_), Meanings )}
    |

```

```

    [ _ ], sentence_no.

parser_anything -->
    [ _ ], !, parser_anything
    |
    [ ].

% [DayName] Year    [ | - | / ] Month    [ | - | / ] Day
%                Month      Day      [Year]
%                Day        {Month    [Year]}
% Today
% Yesterday
% Tommorrow
% 1st Saturday in Month
% Time Units [[from|before|after] Date
% units = Month | Day | Year

% Year = yyyy | yy
% Month = mm | mmm | mmmmmmmmmmm
% Day = dd | dd_th

parser_date( Date ) -->
    yymdd( Date ) | mmdyy( Date ) | ddmmyy( Date ).

% [DayName] Year    [ | - | / ] Month    [ | - | / ] Day
%                Month      Day      [Year]
%                Day        {Month    [Year]}

date1( Date ) -->
    yymdd( Date )
    |
    mmdyy( Date )
    |
    ddmmyy( Date ).

yymdd( date( Year, Month, Day ) ) -->
    year( Year ),
    separator( Sep ),
    month( Month ),
    separator( Sep ),
    day( Day ).

mmdyy( date( Year, Month, Day ) ) -->
    month( Month ),
    separator( Sep ),
    day( Day ),
    (
        separator( Sep ),
        year( Year )
        -> [ ];
        {Year=none}
    ).

ddmmyy( date( Year, Month, Day ) ) -->
    day( Day ),
    separator( Sep ),
    month( Month ),
    (
        separator( Sep ),
        year( Year )
        -> [ ];
        {Year=none}
    ).

separator( Sep ) -->
    [Sep],
    {
        Sep = "-"
        -> true;
        Sep = "/"
        -> true
    } -> {true};
    spaces.

spaces -->
    " " -> spaces; [ ].

% A.D. or B.C.
year( Year ) -->
    parser_number( Ascii, Number ),
    {

```

```

    mapList( A in Ascii, is_digit( A ) ), % integer test
    (
        Number = [_,_]
        -> Year is Number;
        Year is Number - 1900
    )
}.

month( Month ) -->
monthAbbrev( Month )
-> {true};
parser_number( Ascii, Month )
-> {
    mapList( A in Ascii, is_digit(A) ), % integer test
    Month <= 12, Month >= 1
}.

monthAbbrev( Month ) -->
(
    "JAN" -> {Month = 1};
    "FEB" -> {Month = 2};
    "MAR" -> {Month = 3};
    "APR" -> {Month = 4};
    "MAY" -> {Month = 5};
    "JUN" -> {Month = 6};
    "JUL" -> {Month = 7};
    "AUG" -> {Month = 8};
    "SEP" -> {Month = 9};
    "OCT" -> {Month = 10};
    "NOV" -> {Month = 11};
    "DEC" -> {Month = 12}
),
{([Char],(is_alpha(Char)) -> {fail}; {true})}.

day( Day ) -->
parser_number( Ascii, Day ),
(
    mapList( A in Ascii, is_digit(A) ),
    Day >= 0, Day <= 31
),
{(["ST"--_ _] -> {true}; ["ND"--_ _] -> {true}; ["TH"--_ _] -> {true}; []).

dateVerify( date(_Year, Month, Day) ) :-
    monthDays( Month, DaysInMonth ),
    Day <= DaysInMonth.

monthDays( 1, 31 ).
monthDays( 2, 28 ).
monthDays( 3, 31 ).
monthDays( 4, 30 ).
monthDays( 5, 31 ).
monthDays( 6, 30 ).
monthDays( 7, 31 ).
monthDays( 8, 31 ).
monthDays( 9, 30 ).
monthDays( 10, 31 ).
monthDays( 11, 30 ).
monthDays( 12, 31 ).

% show to me show me show
% list for me list
% give to me give me give
% print for me print
% display for me display
present(parseInfo(PI,PI),C-C ) -->
(
    (["LIST"--_ _]|["PRINT"--_ _]|["DISPLAY"--_ _]),
    (["FOR"--_ _],["ME"--_ _]|[])
    |
    (["GIVE"--_ _]|["SHOW"--_ _]),
    (((["TO"--_ _];[]), ["ME"--_ _]); [])
),
!.

% This structure derived from Steven John White - A Portable Natural
% Language Database Query System (1980) pp 142
% i want you to ....

noiseWords -->
(
    (

```



```

        (["WOULD"--_] | ["COULD"--_] | ["CAN"--_]), ["YOU"--_]
    |
    ["I"--_], (["WANT"--_] | ["WOULD"--_], ["LIKE"--_]), ["YOU"--_], ["TO"--_]
    )
    |
    []
    ),
    noise_please,
    !.

noise_please -->
    ["PLEASE"--_], noise_please;
    ["LIKE"--_], noise_like;
    (["TELL"--_] | ["LET"--_] | ["INFORM"--_]), noise_tell;
    ["SAY"--_], noise_mid;
    [].

noise_like -->
    ["TO"--_].

noise_tell -->
    (["ME"--_] | ["US"--_]),
    noise_mid.

noise_mid -->
    (["PLEASE"--_] | ["KNOW"--_]), noise_mid
    |
    (["IF"--_] ; ["WHETHER"--_]), []
    |
    [].

```

## C Database Listing

```
=====
DEPT
=====
```

```
-----
DEPTNO DNAME          LOC
-----
10 ACCOUNTING        NEW YORK
20 RESEARCH           DALLAS
30 SALES              CHICAGO
40 OPERATIONS         BOSTON
```

```
=====
EMP
=====
```

```
-----
EMPNO ENAME          JOB          MGR  HIREDATE   SAL      COMM      DEPTNO
-----
7369 SMITH           CLERK        7902 17-DEC-80   800             20
7499 ALLEN           SALESMAN     7698 20-FEB-81  1600           300         30
7521 WARD            SALESMAN     7698 22-FEB-81  1250           500         30
7566 JONES           MANAGER      7839 02-APR-81  2975             20
7654 MARTIN          SALESMAN     7698 28-SEP-81  1250          1400         30
7698 BLAKE           MANAGER      7839 01-MAY-81  2850             30
7782 CLARK           MANAGER      7839 09-JUN-81  2450             10
7788 SCOTT           ANALYST      7566 09-DEC-82  3000             20
7839 KING            PRESIDENT    17-NOV-81  5000             10
7844 TURNER          SALESMAN     7698 08-SEP-81  1500            0          30
7876 ADAMS           CLERK        7788 12-JAN-83  1100             20
7900 JAMES           CLERK        7698 03-DEC-81   950             30
7902 FORD            ANALYST      7566 03-DEC-81  3000             20
7934 MILLER          CLERK        7782 23-JAN-82  1300             10
```

=====

ORD

=====

ORDID	ORDERDATE	C	CUSTID	SHIPDATE	TOTAL
610	07-JAN-87	A	101	08-JAN-87	101.4
611	11-JAN-87	B	102	11-JAN-87	45
612	15-JAN-87	C	104	20-JAN-87	5860
601	01-MAY-86	A	106	30-MAY-86	2.4
602	05-JUN-86	B	102	20-JUN-86	56
604	15-JUN-86	A	106	30-JUN-86	698
605	14-JUL-86	A	106	30-JUL-86	8324
606	14-JUL-86	A	100	30-JUL-86	3.4
609	01-AUG-86	B	100	15-AUG-86	97.5
607	18-JUL-86	C	104	18-JUL-86	5.6
608	25-JUL-86	C	104	25-JUL-86	35.2
603	05-JUN-86		102	05-JUN-86	224
620	12-MAR-87		100	12-MAR-87	4450
613	01-FEB-87		108	01-FEB-87	6400
614	01-FEB-87		102	05-FEB-87	23940
616	03-FEB-87		103	10-FEB-87	764
619	22-FEB-87		104	04-FEB-87	1260
617	05-FEB-87		105	03-MAR-87	46370
615	01-FEB-87		107	06-FEB-87	710
618	15-FEB-87	A	102	06-MAR-87	3510.5
621	15-MAR-87	A	100	01-JAN-87	730
610	07-JAN-87	A	101	08-JAN-87	101.4
611	11-JAN-87	B	102	11-JAN-87	45
612	15-JAN-87	C	104	20-JAN-87	5860
601	01-MAY-86	A	106	30-MAY-86	2.4
602	05-JUN-86	B	102	20-JUN-86	56
604	15-JUN-86	A	106	30-JUN-86	698
605	14-JUL-86	A	106	30-JUL-86	8324
606	14-JUL-86	A	100	30-JUL-86	3.4
609	01-AUG-86	B	100	15-AUG-86	97.5
607	18-JUL-86	C	104	18-JUL-86	5.6
608	25-JUL-86	C	104	25-JUL-86	35.2
603	05-JUN-86		102	05-JUN-86	224
620	12-MAR-87		100	12-MAR-87	4450
613	01-FEB-87		108	01-FEB-87	6400
614	01-FEB-87		102	05-FEB-87	23940
616	03-FEB-87		103	10-FEB-87	764
619	22-FEB-87		104	04-FEB-87	1260
617	05-FEB-87		105	03-MAR-87	46370
615	01-FEB-87		107	06-FEB-87	710
618	15-FEB-87	A	102	06-MAR-87	3510.5
621	15-MAR-87	A	100	01-JAN-87	730

=====

SALGRADE

=====

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

=====					
ITEM					
=====					
ORDID	ITEMID	PRODID	ACTUALPRICE	QTY	ITEMTOT
610	3	100890	58	1	58
611	1	100861	45	1	45
612	1	100860	30	100	3000
601	1	200376	2.4	1	2.4
602	1	100870	2.8	20	56
604	1	100890	58	3	174
604	2	100861	42	2	84
604	3	100860	44	10	440
603	2	100860	56	4	224
610	1	100860	35	1	35
610	2	100870	2.8	3	8.4
613	4	200376	2.2	200	440
614	1	100860	35	444	15540
614	2	100870	2.8	1000	2800
612	2	100861	40.5	20	810
612	3	101863	10	150	1500
620	1	100860	35	10	350
620	2	200376	2.4	1000	2400
620	3	102130	3.4	500	1700
613	1	100871	5.6	100	560
613	2	101860	24	200	4800
613	3	200380	4	150	600
619	3	102130	3.4	100	340
617	1	100860	35	50	1750
617	2	100861	45	100	4500
614	3	100871	5.6	1000	5600
616	1	100861	45	10	450
616	2	100870	2.8	50	140
616	3	100890	58	2	116
616	4	102130	3.4	10	34
616	5	200376	2.4	10	24
619	1	200380	4	100	400
619	2	200376	2.4	100	240
615	1	100861	45	4	180
607	1	100871	5.6	1	5.6
615	2	100870	2.8	100	280
617	3	100870	2.8	500	1400
617	4	100871	5.6	500	2800
617	5	100890	58	500	29000
617	6	101860	24	100	2400
617	7	101863	12.5	200	2500
617	8	102130	3.4	100	340
617	9	200376	2.4	200	480
617	10	200380	4	300	1200
609	2	100870	2.5	5	12.5
609	3	100890	50	1	50
618	1	100860	35	23	805
618	2	100861	45	11	50
618	3	100870	45	10	450
621	1	100861	45	10	450
621	2	100870	2.8	100	280
615	3	100871	5	50	250
608	1	101860	24	1	24
608	2	100871	5.6	2	11.2
609	1	100861	35	1	35
606	1	102130	3.4	1	3.4
605	1	100861	45	100	4500
605	2	100870	2.8	500	1400
605	3	100890	58	5	290
605	4	101860	24	50	1200
605	5	101863	9	100	900
605	6	102130	3.4	10	34
612	4	100871	5.5	100	550
619	4	100871	5.6	50	280

=====

PRODUCT

=====

-----

PROID	DESCRIP
100860	ACE TENNIS RACKET I
100861	ACE TENNIS RACKET II
100870	ACE TENNIS BALLS-3 PACK
100871	ACE TENNIS BALLS-6 PACK
100890	ACE TENNIS NET
101860	SP TENNIS RACKET
101863	SP JUNIOR RACKET
102130	RH: "GUIDE TO TENNIS"
200376	SB ENERGY BAR-6 PACK
200380	SB VITA SNACK-6 PACK

-----

=====

PRICE

=====

PROID	STDPRICE	MINPRICE	STARTDATE	ENDDATE
100871	4.8	3.2	01-JAN-85	01-DEC-85
100890	58	46.4	01-JAN-85	
100890	54	40.5	01-JUN-84	31-MAY-84
100860	35	28	01-JUN-86	
100860	32	25.6	01-JAN-86	31-MAY-86
100860	30	24	01-JAN-85	31-DEC-85
100861	45	36	01-JUN-86	
100861	42	33.6	01-JAN-86	31-MAY-86
100861	39	31.2	01-JAN-85	31-DEC-85
100870	2.8	2.4	01-JAN-86	
100870	2.4	1.9	01-JAN-85	01-DEC-85
100871	5.6	4.8	01-JAN-86	
101860	24	18	15-FEB-85	
101863	12.5	9.4	15-FEB-85	
102130	3.4	2.8	18-AUG-85	
200376	2.4	1.75	15-NOV-86	
200380	4	3.2	15-NOV-86	

=====

CUSTOMER

=====

CUSTID	NAME	ADDRESS	CITY	ST	ZIP	AREA	PHONE	REPID	CREDITLIMIT
100	JOCKSPORTS	345 VIEWRIDGE	BELMONT	CA	96711	415	598-6609	7844	5000
101	TKB SPORT SHOP	490 BOLI RD.	REDWOOD CITY	CA	94061	415	368-1223	7521	10000
102	VOLLYRITE	9722 HAMILTON	BURLINGAME	CA	95133	415	644-3341	7654	7000
103	JUST TENNIS	HILLVIEW MALL	BURLINGAME	CA	97544	415	677-9312	7521	3000
104	EVERY MOUNTAIN	574 SURRY RD.	CUPERTINO	CA	93301	408	996-2323	7499	10000
105	K + T SPORTS	3476 EL PASEO	SANTA CLARA	CA	91003	408	376-9966	7844	5000
106	SHAPE UP	908 SEQUOIA	PALO ALTO	CA	94301	415	364-9777	7521	6000
107	WOMENS SPORTS	VALCO VILLAGE	SUNNYVALE	CA	93301	408	967-4398	7499	10000
108	NORTH WOODS HEALTH AND FITNESS SUPPLY CENTER	98 LONE PINE WAY	HIBBING	MN	55649	612	566-9123	7844	8000

```
=====
MVDIR
=====
```

```
-----
DIRNO DIRNAME
-----
```

```
D01 STEVEN SPIELBERG
D02 PENNY MARSHALL
D03 SPIKE LEE
```

```
=====
MVSTAR
=====
```

```
-----
STNO STNAME
-----
```

```
S01 MADONNA
S02 TOM HANKS
S03 GEENA DAVIS
S04 JEFF GOLDBLOOM
S05 DENZEL WASHINGTON
```

```
=====
MVMOVIE
=====
```

```
-----
MVNO MVNAME
-----
```

```
M01 JURASSIC PARK
M02 LEAGUE OF THEIR OWN
M03 MALCOLM X
```

```
=====
MVSTARSIN
=====
```

```
-----
MVNO STNO
-----
```

```
M02 S01
M02 S02
M02 S03
M01 S04
M03 S05
```

```
=====
MVDIRECTS
=====
```

```
-----
MOVIENO DIRNO
-----
```

```
M01 D01
M02 D02
M03 D03
```

## D Database Interface

The interface of the Prolog part of the system to the C server is contained in one small Prolog module (where.pl). The interface is implemented in the form of four predicates.

The first predicate is **whereLocation/5** (Figure D.1) The purpose of this predicate is to determine the database graph node at a particular table, row, and column position in the database. The first term of the predicate (**location/3**) specifies the table, column and row that identifies the database graph node. The system accesses the database through ORACLE in order to determine the second term, (the **node/3** predicate). This predicate is used when the system is producing the spanning trees to access nodes in the database graph that are in the same row.

```
whereLocation( +location(Table,Column,Row), ?node( Location, Value, cost(Cost) )-)
```

**Figure D.1 - whereLocation/2**

The second predicate, **where\_prefix/3**, is used during tokenization of a sentence (see section 6.1.1). The first term, **String**, is the list of ascii codes that comprise the input string<sup>20</sup>.

```
where_prefix( String, Prefix-Alternatives, Suffix )
```

**Figure D.2 - where\_prefix/3**

The second and third terms are bound by the predicate. **Prefix** is a prefix of the string,

<sup>20</sup> - In Prolog "strings" are implemented as a list of ascii codes



**String**. **Alternatives** is a list of the possible interpretations<sup>21</sup> of the string **Prefix**, and **Suffix** is the portion of **String** after **Prefix** is removed, ie - append( Prefix, Suffix, String ). The inverted index is used to implement this predicate.

The final two predicates are used to send SQL commands to ORACLE and receive the results of the SQL. These predicates are shown in Figure D.3.

```
oracleServer_send( Command ).
oracleServer_read( Results ).
```

**Figure D.3 - SQL Access Predicates**

The first predicate, **oracleServer\_send/1**, is used to send an SQL command to ORACLE. Only three commands are currently supported, **login(userid/password)**, **logoff**, and **sql(SQL)**. The **login/1** and **logoff** are used to log on and off of a database. **sql/1** is used to send an SQL command directly to the DBMS. **SQL** is a Prolog atom that is an SQL command. The results of the SQL command can be obtained using the **oracleServer\_read/1** predicate. The first term of this predicate is a list (possibly empty) of tuples that are returned by the SQL command.

---

<sup>21</sup> - see the Interpretation Alternatives column in Figure 6.10 for examples

## E Test Runs

**smith works for ford.**

EMP0.ENAME	EMP1.ENAME
-----	-----
ALLEN	BLAKE
FORD	JONES
SCOTT	JONES
WARD	BLAKE

Do these examples seem reasonable?

| : yes.  
| : **smith works in research.**

EMP0.ENAME	DEPT0.DNAME
-----	-----
CLARK	ACCOUNTING
KING	ACCOUNTING
MILLER	ACCOUNTING
SMITH	RESEARCH

Do these examples seem reasonable?

| : yes.  
| : **smith is managed by ford.**

EMP0.ENAME	EMP1.ENAME
-----	-----
ALLEN	BLAKE
FORD	JONES
SCOTT	JONES
WARD	BLAKE

Do these examples seem reasonable?

| : yes.

To configure the system for a new database, the user (in **bold**) enters sentences that describe data in the database. By using these sentences and accessing the database the system can determine the meaning of nouns and verbs. (Users tend to know a lot about some of the data in the database).

**Examples are presented to provide feedback**

| : **smith's salary is 800.**

EMP0.ENAME	EMP0.SAL
-----	-----
ALLEN	1600
JONES	2975
SMITH	800
WARD	1250

Do these examples seem reasonable?

| : **yes.**

| : **the manager of smith is ford.**

EMP0.ENAME	EMP1.ENAME
-----	-----
ALLEN	BLAKE
FORD	JONES
SCOTT	JONES
WARD	BLAKE

Do these examples seem reasonable?

| : **yes.**

| : **the salary of ford is 3000.**

EMP0.ENAME	EMP0.SAL
-----	-----
ALLEN	1600
JONES	2975
SMITH	800
WARD	1250

Do these examples seem reasonable?

| : **yes.**

| : smith earns 800.

EMP0.ENAME	EMP0.SAL
-----	-----
ALLEN	1600
JONES	2975
SMITH	800
WARD	1250

Do these examples seem reasonable?

| : yes.

| : what is smith's salary.

EMP0.SAL
-----
800

| : which person's salary is 800.

EMP0.ENAME
-----
SMITH

| : the salary of which person is 800.

EMP0.ENAME
-----
SMITH

| : what is the salary of the person who manages the person who manages smith.

EMP0.SAL
-----
2975

| : what is the salary of the person who manages the person managing smith.

EMP0.SAL
-----
2975

That's it. System configuration is complete (for now). Questions can now be asked about the database. Note that questions can be asked as soon as the user has defined the words that will appear in the question. After asking some questions, more words can be defined if desired.

First an easy question to get warmed up.

| : which person's manager's salary is 2975.

Possessives and comparison.

EMP1.ENAME

-----

SCOTT

FORD

| : who does the person managing scott manage.

EMP1.ENAME

-----

SCOTT

FORD

| : which person's salary is over ford's salary.

EMP1.ENAME

-----

KING

| : who earns over what smith earns.

EMP1.ENAME      EMP0.SAL

-----

ALLEN	800
WARD	800
JONES	800
MARTIN	800
BLAKE	800
CLARK	800
SCOTT	800
KING	800
TURNER	800
ADAMS	800
JAMES	800
FORD	800
MILLER	800

| : who earns over smith's salary.

EMP1.ENAME

-----

ALLEN  
WARD  
JONES  
MARTIN  
BLAKE  
CLARK  
SCOTT  
KING  
TURNER  
ADAMS  
JAMES  
FORD  
MILLER

| : who earns over the money ford earns.

EMP1.ENAME

-----

KING

| : who is the person managing scott.

EMP1.ENAME

-----

JONES

| : who earns over an employee's average salary.

EMP0.ENAME

-----

JONES  
BLAKE  
CLARK  
SCOTT  
KING  
FORD

Functions such as average,  
total, highest, and lowest are  
provided.

| : **who earns over an employee working in research's average salary.**

```
EMP0.ENAME
-----
JONES
BLAKE
CLARK
SCOTT
KING
FORD
```

| : **who earns over the average salary of an employee working in research.**

```
EMP0.ENAME
-----
JONES
BLAKE
CLARK
SCOTT
KING
FORD
```

| : **what is the average salary of an employee working in research.**

```
AVG(EMP0.SAL)
-----
2175
```

| : **what is the lowest salary of the people who earn over the average salary of an employee.**

```
MIN(EMP0.SAL)
-----
2450
```

Okay I admit that the response generation is very crude, but response generation is another thesis in itself.

| : **list the name and salary of people earning over the average salary of an employee.**

r3s3: No Parses not implemented yet

| : list the name and salary of the people earning over the average salary of an employee.

EMP0.ENAME	EMP0.SAL
-----	-----
JONES	2975
BLAKE	2850
CLARK	2450
SCOTT	3000
KING	5000
FORD	3000

| : how many people earn over the average salary of an employee.

```
COUNT(EMP0.ENAME)
-----
6
```

| : can you list the salary and manager of employees working in research.

r3s3: No Parses not implemented yet

| : can you list the salary and manager of the employees working in research.

EMP0.SAL	EMP1.ENAME
-----	-----
3000	JONES
3000	JONES
800	FORD
2975	KING
1100	SCOTT

| : can you list the name salary and manager of the employees working in research.

EMP0.ENAME	EMP0.SAL	EMP1.ENAME
-----	-----	-----
SCOTT	3000	JONES
FORD	3000	JONES
SMITH	800	FORD
JONES	2975	KING
ADAMS	1100	SCOTT

| : print the average salary of the employees working in research.

```
AVG(EMP0.SAL)
-----
2175
```



| : list the name and salary of the employees earning over 800.

EMP0.ENAME	EMP0.SAL
-----	-----
ALLEN	1600
WARD	1250
JONES	2975
MARTIN	1250
BLAKE	2850
CLARK	2450
SCOTT	3000
KING	5000
TURNER	1500
ADAMS	1100
JAMES	950
FORD	3000
MILLER	1300

| : can you list the name manager and salary of the employees working in research.

EMP0.ENAME	EMP0.SAL	EMP1.ENAME
-----	-----	-----
SCOTT	3000	JONES
FORD	3000	JONES
SMITH	800	FORD
JONES	2975	KING
ADAMS	1100	SCOTT

| : who works in research.

EMP0.ENAME
-----
SMITH
ADAMS
FORD
SCOTT
JONES

| : **tkb sport shop paid 58 for ace tennis net.**

2 Interpretations. They Are:

Option #1

"58" refers to ACTUALPRICE in the ITEM table

Option #2

"58" refers to ITEMTOT in the ITEM table

You may select an option as a correct interpretation, ask for more options, or decide to stop this definition

| : **i like number 1.**

[pickAnOption(1)]

| : **what is the average price a customer has paid for ace tennis net.**

AVG (ITEM0.ACTUALPRICE)

-----

56.6666666666666666666666666667

| : **who paid under 58 for ace tennis net.**

CUSTOMER0.NAME

-----

JOCKSPORTS

| : **who has paid less than 58 for ace tennis net.**

CUSTOMER0.NAME

-----

JOCKSPORTS

| : **what did jocksports pay for ace tennis net.**

ITEM0.ACTUALPRICE

-----

50

| : **what is the lowest highest and average salary of employee working in research.**

r6s4: No interpretation not implemented

Let's teach the system a word with a complicated meaning. This one has ambiguity and requires 4 table joins. The user is presented with a choice of two meanings.

Let's try some questions.

| : what is the lowest highest and average salary of the employees working in research.

MIN(EMP0.SAL) , MAX(EMP0.SAL) , AVG(EMP0.SAL)

```
-----
```

800	3000	2175
-----	------	------

I added another database about **movies**. Let's try using it for a while.

| : steven spielberg directed jurassic park.

MVDIR0.DIRNAME	MVMOVIE0.MVNAME
-----	-----
PENNY MARSHALL	LEAGUE OF THEIR OWN
SPIKE LEE	MALCOLM X
STEVEN SPIELBERG	JURASSIC PARK

Do these examples seem reasonable?

| : yes.

| : who directed malcolm x.

MVDIR0.DIRNAME

```
-----
```

SPIKE LEE

| : league of their own starred tom hanks.

MVMOVIE0.MVNAME	MVSTAR0.STNAME
-----	-----
JURASSIC PARK	JEFF GOLDBLOOM
LEAGUE OF THEIR OWN	GEENA DAVIS
LEAGUE OF THEIR OWN	MADONNA
LEAGUE OF THEIR OWN	TOM HANKS

Do these examples seem reasonable?

| : yes.

| : who did jurassic park star.

MVSTAR0.STNAME

```
-----
```

JEFF GOLDBLOOM

| : **steven spielberg directed jeff goldbloom.**

MVDIR0.DIRNAME	MVSTAR0.STNAME
-----	-----
PENNY MARSHALL	GEENA DAVIS
PENNY MARSHALL	MADONNA
PENNY MARSHALL	TOM HANKS
STEVEN SPIELBERG	JEFF GOLDBLOOM

Do these examples seem reasonable?

| : **yes.**

| : **who was tom hanks directed by.**

MVDIR0.DIRNAME
-----
PENNY MARSHALL

| : **denzel washington is the star of malcolm x.**

MVMOVIE0.MVNAME	MVSTAR0.STNAME
-----	-----
JURASSIC PARK	JEFF GOLDBLOOM
LEAGUE OF THEIR OWN	GEENA DAVIS
LEAGUE OF THEIR OWN	MADONNA
LEAGUE OF THEIR OWN	TOM HANKS

Do these examples seem reasonable?

| : **yes.**

| : **who is the star of league of their own.**

MVSTAR0.STNAME
-----
MADONNA
TOM HANKS
GEENA DAVIS

| : **steven spielberg directed jurassic park.**

MVDIR0.DIRNAME	MVMOVIE0.MVNAME
-----	-----
PENNY MARSHALL	LEAGUE OF THEIR OWN
SPIKE LEE	MALCOLM X
STEVEN SPIELBERG	JURASSIC PARK

Do these examples seem reasonable?

| : **yes.**

| : **steven spielberg directed jeff goldbloom.**

MVDIR0.DIRNAME	MVSTAR0.STNAME
-----	-----
PENNY MARSHALL	GEENA DAVIS
PENNY MARSHALL	MADONNA
PENNY MARSHALL	TOM HANKS
STEVEN SPIELBERG	JEFF GOLDBLOOM

Do these examples seem reasonable?

| : **yes.**

| : **who directed jeff goldbloom.**

MVDIR0.DIRNAME
-----
STEVEN SPIELBERG

| : **who directed jurassic park.**

MVDIR0.DIRNAME
-----
STEVEN SPIELBERG

| : **who directed malcolm x.**

MVDIR0.DIRNAME
-----
SPIKE LEE

| : **who directed tom hanks.**

MVDIR0.DIRNAME
-----
PENNY MARSHALL

Now we'll try defining two different senses for the verb direct. Let's start with director directing movie. Notice that we defined this earlier. The system can handle such mistakes.

Now we'll define the other sense of the verb direct - director directing star.

Now a couple of question to check things out.

| : **smith works for ford.**

EMP0.ENAME	EMP1.ENAME
-----	-----
ALLEN	BLAKE
FORD	JONES
SCOTT	JONES
WARD	BLAKE

Let's try two different senses of the verb works. One will relate to the employee database and one will relate to the movie database. We'll start with employee works for manager.

Do these examples seem reasonable?

| : **yes.**

| : **tom hanks works for penny marshall.**

MVSTAR0.STNAME	MVDIR0.DIRNAME
-----	-----
GEENA DAVIS	PENNY MARSHALL
JEFF GOLDBLOOM	STEVEN SPIELBERG
MADONNA	PENNY MARSHALL
TOM HANKS	PENNY MARSHALL

Now we'll define the 'movie star works for director' sense.

Do these examples seem reasonable?

| : **yes.**

| : **who works for ford.**

A couple of test questions.

EMP0.ENAME
-----
SMITH

| : **who works for king.**

EMP0.ENAME
-----
JONES
CLARK
BLAKE

| : **who works for steven spielberg.**

MVSTAR0.STNAME
-----
JEFF GOLDBLOOM

| : smith works in research.

EMP0.ENAME	DEPT0.DNAME
CLARK	ACCOUNTING
KING	ACCOUNTING
MILLER	ACCOUNTING
SMITH	RESEARCH

Do these examples seem reasonable?

| : yes.

| : geena davis works in league of their own.

MVSTAR0.STNAME	MVMOVIE0.MVNAME
GEENA DAVIS	LEAGUE OF THEIR OWN
JEFF GOLDBLOOM	JURASSIC PARK
MADONNA	LEAGUE OF THEIR OWN
TOM HANKS	LEAGUE OF THEIR OWN

Do these examples seem reasonable?

| : yes.

| : who works in accounting.

EMP0.ENAME
CLARK
KING
MILLER

| : who works in jurassic park.

MVSTAR0.STNAME
JEFF GOLDBLOOM

The purpose of this program is to demonstrate that the claims made in my thesis are computationally effective. The thesis makes claims only about the problem of portability of a natural language interfaces and so only minimal effort was directed towards creating a general purpose natural language interface.