NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THÈSES CANADIENNES

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.
NAME OF AUTHOR: NOM DE L'AUTEUR: Robert F. Hadley

TITLE OF THESIS: TITRE DE LA THÈSE: A Natural Language Query System for a Prolog Database

UNIVERSITY: UNIVERSITÉ: Simon Fraser University

DEGREE FOR WHICH THESIS WAS PRESENTED/ GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE: Master of Science

YEAR THIS DEGREE CONFERRED: ANNÉE D'OBTENTION DE CE GRADÉ: 1984

NAME OF SUPERVISOR/ NOM DU DIRECTEUR DE THÈSE: Nick Cercone

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

DATED: April 6, 1984

PERMANENT ADDRESS: RÉSIDENCE FIXE: 2476 Trinity St.

Vancouver, Canada V5K 1E1
A NATURAL LANGUAGE QUERY SYSTEM FOR A PROLOG DATABASE

by

Robert Francis Hadley

Ph.D. University of British Columbia 1973
B.A.(Hons) University of Virginia 1967

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Computing Science

Robert Francis Hadley 1983

SIMON FRASER UNIVERSITY

December 1983

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.
Name: Robert Francis Hadley
Degree: Master of Science
Title of thesis: A Natural Language Query System for a Prolog Database

Examining Committee:

Chairperson: Arthur L. Liestman

______________________________
Nick Cercone
Senior Supervisor

______________________________
Veronica Dahl

______________________________
Brian V. Punt

______________________________
Gordon McCalla
External Examiner
Associate Professor
Department of Computational Science
University of Saskatchewan

Date Approved: 15 December 1983
I hereby grant to Simon Fraser University the right to lend
my thesis, project or extended essay (the title of which is shown below)
to users of the Simon Fraser University Library, and to make partial or
single copies only for such users or in response to a request from the
library of any other university, or other educational institution, on
its own behalf or for one of its users. I further agree that permission
for multiple copying of this work for scholarly purposes may be granted
by me or the Dean of Graduate Studies. It is understood that copying
or publication of this work for financial gain shall not be allowed
without my written permission.

Title of Thesis/Project/Extended Essay

A NATURAL LANGUAGE QUERY SYSTEM FOR

A PROLOG DATABASE

/ 

Author: _____________________________

(Signature)

Robert F. Hadley

(name)

April 6, 1984

(date)
The problem of interpreting a natural language, such as English, for "comprehension" by machine is a notoriously difficult one which has many unsolved aspects. These include (a) the problem of resolving ambiguities of syntax, semantics, and reference which occur in natural discourse, by appealing to the general background knowledge which speakers of the language can reasonably be expected to have, and to special knowledge which speakers in a given situation are likely to have; (b) the problem of finding all reasonable interpretations for a given statement or question, when it is taken out of a particular discourse or context, and asking a speaker to choose the intended interpretation. The latter problem is primarily concerned with discovering and representing the syntax and semantics of natural language, and with representing the interrelations between these two in a fashion that is codifiable in computer programs.

The present thesis will be primarily concerned with a constrained version of the latter problem, although, to be sure, certain assumptions will be made about the intentions of speakers of the language. In particular, this thesis presents a system (SHADOW) which interprets English questions which are
posed to a Prolog database. The system not only interprets
questions by translating them into Prolog procedures which may
be conceived as formal queries, but answers questions by
evaluating the formal queries for a given database.
**ACKNOWLEDGEMENTS**

I wish to express my deep appreciation to all members of my Thesis Committee—Nick Cercone, Veronica Dahl, and Brian Funt—for their encouragement, support, and valuable instruction. I have learned a great deal from each of them, which has helped me not only in writing this thesis, but in my general area of interest, Artificial Intelligence. I am also indebted to my supervisor, Nick Cercone, for his very helpful comments and suggestions on the preparation of the final draft of this thesis. I am deeply indebted to Veronica Dahl for teaching me many of the techniques and strategies which made the writing of this thesis possible. To Brian Funt I am especially grateful for personal encouragement in dealing with the trials of being a student. I also wish to thank Ed Bryant for his considerable help in solving technical system problems which arose in the writing of this thesis. Finally, I wish to thank my dear friend Susan Wendell for all the support and good advice which she has given me, not only during the writing of this thesis, but during the course of my graduate studies.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. SOME EXISTING SYSTEMS</td>
<td>8</td>
</tr>
<tr>
<td>Montague Grammars</td>
<td>8</td>
</tr>
<tr>
<td>Terry Winograd and SHRDLU</td>
<td>14</td>
</tr>
<tr>
<td>Veronica Dahl and L3</td>
<td>19</td>
</tr>
<tr>
<td>The Planes Project</td>
<td>23</td>
</tr>
<tr>
<td>III. A COMPARISON OF TWO GENERAL APPROACHES</td>
<td>28</td>
</tr>
<tr>
<td>IV. ALGORITHMS</td>
<td>41</td>
</tr>
<tr>
<td>General Strategy</td>
<td>41</td>
</tr>
<tr>
<td>More Specific Algorithms</td>
<td>49</td>
</tr>
<tr>
<td>V. SPECIFIC PROCEDURES</td>
<td>53</td>
</tr>
<tr>
<td>The Question Phrase</td>
<td>54</td>
</tr>
<tr>
<td>The Subject Noun Phrase</td>
<td>54</td>
</tr>
<tr>
<td>Noun Phrases With Prepositional Modifiers</td>
<td>64</td>
</tr>
<tr>
<td>Parsing &quot;Dummy&quot; Noun Phrases</td>
<td>69</td>
</tr>
<tr>
<td>Parsing Verb Groups</td>
<td>69</td>
</tr>
<tr>
<td>Analysing Adverbial Phrases</td>
<td>71</td>
</tr>
<tr>
<td>Composing a Top-level Query</td>
<td>77</td>
</tr>
<tr>
<td>Evaluating a Top-level Query</td>
<td>83</td>
</tr>
<tr>
<td>VI. CONCLUSION</td>
<td>94</td>
</tr>
<tr>
<td>Problem Areas</td>
<td>94</td>
</tr>
<tr>
<td>Summary</td>
<td>100</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Possibilities</td>
<td>102</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>106</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>160</td>
</tr>
<tr>
<td>APPENDIX C - PROGRAM LISTING</td>
<td>163</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>208</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

The problem of interpreting a natural language, such as English, for "comprehension" by machine is a notoriously difficult one which has many unsolved aspects. These include (a) the problem of resolving ambiguities of syntax, semantics, and reference which occur in natural discourse, by appealing to the general background knowledge which speakers of the language can reasonably be expected to have, and to special knowledge which speakers in a given situation are likely to have; (b) the problem of finding all reasonable interpretations for a given statement or question, when it is taken out of a particular discourse or context, and asking a speaker to choose the intended interpretation. The latter problem is primarily concerned with representing the syntax and semantics of natural language, and with representing the interrelations between these two in a fashion that is codifiable in computer programs.

The present thesis will be primarily concerned with a constrained version of the latter problem, although, to be sure, certain assumptions will be made about the intentions of speakers of the language. In particular, this thesis presents a system (SHADOW) which interprets English questions which are posed to a Prolog database. The system not only interprets questions by translating them into Prolog procedures which may be conceived as formal queries, but answers questions by
evaluating the formal queries for a given database.

The problem of question interpretation is being delimited here in several ways: a) It is being assumed that the questions to be interpreted are concerned with a particular database, namely one which contains information usually found in a university calendar. Thus, the system will be designed to answer questions about: which courses are being offered by what department, and when; which professors are teaching those courses; which professors belong to which departments, and what the prerequisites for particular courses are. As a consequence of this first constraint, the semantic dictionary of the system can be small. Only certain topics are made "known" to the system.

b) Another consequence of this restriction is that the syntax of questions which the system is likely to receive is also constrained. Only certain grammatical types of question are likely to be posed to a database question-answering system, and the system is not designed to recognise all syntactically well-formed questions.

In spite of the above mentioned constraints, the system described here is not an expert system, as typically conceived. For the system has been designed to be expandable both in syntax and in semantics, so that, in principle, a wide range of information which is representable in the predicate calculus could be retrieved using an expanded version of this system. Even without expanding the present system, it could be quickly
adapted for use on relational databases which are concerned with a different subject matter, primarily by altering the semantic dictionary which is attached to the system's program. This is because, to a large degree, the system has been designed to be general in its syntax and semantics, following, abstractly, principles of semantics which have been developed by Richard Montague [Montague(1970)], and more concretely, methods of interpretation which have been developed by Veronica Dahl [Dahl(1981)], and to a certain degree by Terry Winograd [Winograd(1973)].

The system here described (SHADOW) is implemented in the Prolog programming language. In many ways Prolog is ideally suited for the problem of interpreting natural language. Its most striking features are its sophisticated automatic backtracking mechanism, and its automatic deduction capabilities. The virtues of these features will be discussed more fully in chapter 3. At this point, however, it is worth noting that these features allow the user to enter multiple senses for nouns, adjectives, and verbs. The program will automatically, through backtracking, select those senses of these entries which yield a coherent reading of the question, if one exists. The program includes information regarding type restrictions (semantic markers) [Katz and Postal(1964)] on the adjectives, nouns, and verbs in the program's dictionary. These type restrictions are consulted prior to evaluating a query, so that a correct interpretation of the query can be found.
efficiently.

Once a coherent interpretation for a question has been found, and an attempt is made to evaluate the query, it may be found that the interpretation contains certain false presuppositions, given the actual data which the database contains. The automatic backtracking mechanism of Prolog permits all other possible interpretations of such a question to be tested. SHADOW has been designed so that it will report a false presupposition to the user only after all possible interpretations have been tried.

Although SHADOW accepts a limited range of natural syntax, its range is quite broad enough to be useful. Many natural sounding constructions are accepted by the program, such as,

What courses are being offered next semester?

Who is teaching computing 354 in the summer?

What is Frege teaching this year?

In the first question, the existence of an implied noun phrase (the deep subject, which is "doing the offering") is inferred and made explicit in the internal representation of the query. In the second question, the type of the noun implied by the pronoun 'who' is inferred from the context. In the third case, we have an inversion of the auxilliary verb with the deep
subject of the sentence. This case is easily handled by treating the sentence as equivalent to a normal passive sentence, since the program recognises passive sentences as a matter of course. Also, in the third sentence the implicit deep object, which is represented by the surface pronoun 'what', is inferred to be 'courses'.

The program can also recognise questions which contain relative clauses nested arbitrarily deeply. This permits the user to pose very complicated questions, involving multiple set intersections, to the system.

Several natural language quantifiers, including 'the', 'a', 'some', 'every', and 'no' are recognized by the system and correctly interpreted, even when many quantifiers are present. Classically ambiguous questions, such as, "Which professors who do not teach a course this spring are teaching some course this fall?" or "Does every classics professor teach some course?" are disambiguated by SHADOW by questioning the user at crucial points during the interpretation of the sentence. Several examples of this are discussed in Appendix A.

The system also recognises questions involving complicated adverbial phrases, such as 'in the fall of every year', or 'during every semester of 84' and even 'in the year of 83 in the spring'. Many more common forms are also recognised, such as 'last fall', 'in the coming semester', 'in the previous year', and so on. Such phrases may occur in embedded sentences as well as in the main clause.
Many prepositional phrases, nested arbitrarily deep, are also recognised by the system, such as 'a member of the department of philosophy', or 'who teaches a course in computing for the department of mathematics in the year of 83'. In this latter case the adverbial phrase 'in the year of 83' is recognised as pertaining to the verb 'teaches' and not to the preceding noun, 'Mathematics'.

The SHADOW system also recognises the most common forms of negation, as in, 'Which professors do not teach a course this semester?' or 'Does some professor teach no course?' In Appendix A many detailed examples of questions which the program can analyse are discussed. They have been chosen to illustrate that SHADOW has been designed both to answer practical questions which are likely to be posed to the system, and to answer complex and subtle questions which may be of interest more to cognitive scientists, philosophers and linguists than they are to the average user of the system.

In the following chapter, we shall examine four existing systems which perform natural language interpretation. These systems have been chosen both because they are recognised as major works in the field of computational linguistics and because of their special relevance to the SHADOW system. We have not attempted in this thesis to present an exhaustive discussion of existing natural language interpreters. Rather, for present purposes, it is more useful to discuss in some depth a small number of systems which illuminate the present work.
In chapter 3 we compare two different approaches to natural language interpretation, and present reasons for choosing between them. Chapter 4 presents a discussion of the major algorithms and heuristics which are employed by the SHADOW system, and in the final chapter we discuss problem areas and possible extensions of the system.
II. SOME EXISTING SYSTEMS

As was stated in the previous section, SHADOW performs both syntactic and semantic analysis in a single pass through the sentence. This is also true of two other systems we shall discuss, Terry Winograd's SHRDLU [1973] and Veronica Dahl's L3 [1981]. It will be useful in understanding all three of these systems if we briefly examine a general theory which holds that syntax and semantics can be integrated, and indeed can be made isomorphic. Such a theory has been developed by the philosopher-logician Richard Montague [1970]. This theory has been considerably expanded by other people, notably, David Lewis [1976] and Barbara Partee [1976].

We shall not explore the intricacies of Montague's methods here, for the systems we shall describe share only some rather abstract features with Montague's system. The interested reader is referred, however, to [Lewis (1976)] for an excellent exposition of Montague's general theory.

**Montague Grammars**

The principal idea behind Montague's system is that the meaning of a complex string of symbols in natural language is a strict function of the meaning of its simpler parts. This notion is commonplace among logicians and mathematicians, but it is
Montague who has most convincingly argued that this principle can be applied to natural language, in spite of its ambiguities and complexities. Montague's syntactic system uses a categorial grammar. Such grammars have been around for many years, but only since they have been expanded to include transformational reordering rules have they proved useful [Lewis(1976), Partee(1976)]. For purposes of illustration I will discuss a limited categorial context-free grammar, which is due to David Lewis [1976].

Categorial grammars consist of two kinds of categories, basic categories, which in our example shall be sentence (S), name (N), and common noun (C), and derived categories. Each word in the language recognized by the grammar is entered in a lexicon, where each entry in the lexicon is an ordered pair consisting of the word and its category in the grammar. For example, in the lexicon there may be an entry (Sherlock N), indicating that Sherlock is the name of an individual.

Derived categories in the grammar have the general form: \( c/ c_1 \ldots c_n \). This form is both a name for the category and a rule which should be interpreted as follows: in order to obtain a molecule of the category \( c \), first write down a string which belongs to the category named by the current rule, and then concatenate a list of atoms or molecules which belong to the categories \( c_1 \ldots c_n \), in order. For example, suppose we are given that the word 'outwits' belongs to the category \( c/c_1 \), and that the word 'Moriarty' belongs to the category \( c_1 \). Then the
category name c/cl may be taken as a rule which tells us that if we write down a word in the current category, 'outwits', and follow it by a word from category, c1, e.g., 'Moriarty', then we obtain the string 'outwits Moriarty', and so c may be thought of as the category, verb phrase. This category may be represented in a categorial grammar as S/N, indicating that anything in this category requires only an element from the category N to make a complete sentence. Thus, given that we have derived that 'outwits Moriarty' is in the category S/N, and given that 'Sherlock' belongs to the category N, we easily derive that 'outwits Moriarty Sherlock' belongs to the category S, and so is a sentence. Clearly the sentence is in prefix notation, but the addition of transformational reordering rules to the grammar can, in general, restore natural word order.

The following is a simple categorial grammar which can produce an infinity of sentences [Lewis (1976)]:

\[
\begin{align*}
\{ & \text{a (S/(S/N)) / C } \{ \text{sleuth C} \} \\
& \text{believes (S/N) / S } \{ \text{cleverly (S/N) / (S/N)} \} \\
& \text{every (S/(S/N)) / C } \{ \text{Sherlock N} \} \\
& \text{sleeps S/N } \{ \text{something S/(S/N)} \} \\
& \text{is (S/N) / N } \{ \text{the ( S/ (S/N)) / C} \} \\
& \text{outwits (S/N) / N } \{ \text{which (C/C) / (S/N)} \} \\
& \text{Moriarty N } \{ \text{tricky C/C} \} 
\end{align*}
\]
The above grammar has an equivalent BNF type context-free grammar, and in terms of the syntactic properties alone, there is not much, if anything, to recommend this grammar over its BNF equivalent. But the interesting thing is that this grammar sets forth the manner in which the semantic structures of the language are built.

In order to illustrate how the syntactic rules of a categorial grammar may also be seen as semantic rules, we should first recall the distinction between the intension and extension of a symbolic expression. The extension of a proper noun (a name) is the entity in the actual world which it does denote, and its POSSIBLE extensions are any entities in any imaginary world which the name would denote if the imaginary world were the actual world. For example, the name 'Sherlock' has an empty extension in the actual world, but it would denote a real individual if the world were different in certain respects. The actual extension of a common noun is the set of things in the actual world which the noun does denote or refer to. For example, the extension of the noun 'tree' is the set of all trees which now exist. Finally, the extension of a sentence is the truth value of that sentence.

The intensions of each of these three basic types are those entities which determine what the extension of the linguistic item is. For example, the intension of a common noun determines the set of entities in the world which that noun properly applies to, and the intension of a sentence determines what its
truth value in the actual world is. Thus, we may think of intensions as functions attached to symbolic expressions, which, given certain information about the state of the world, determine the extensions of those expressions, provided the expressions fall into one of the three basic categories, name, common noun, or sentence. Roughly speaking, intensions correspond to the meanings of expressions, since it is the meaning of an expression which determines what it refers to.

So far, we have been considering only the intensions of the basic categories. The intensions of all derived categories, such as C/C and S/W, can also be viewed as functions. To see this, consider how adjectives work. In our example grammar, adjectives are placed in the category C/C. The name of the category reflects the fact that when we write down a member of that category, e.g., 'green', followed by a member of the category C, e.g., 'grass', the result is something which belongs to the category C, namely 'green grass'. Thus, in this grammar, 'green grass' is considered to be a common noun. So, in this grammar, an adjective is something which is applied to a common noun to yield a new common noun, whose meaning is more complicated than the original's. Therefore, we may view the meaning of an adjective as a function which takes as its argument the intension of a common noun and returns as its value the intension of a composite common noun. Thus the intension of some common nouns is determined by the composition of functions.
The functions we are discussing may be functions of any kind, as long as they satisfy the formal set-theoretic definitions of functions. The functions in question, therefore, may be functions defined in any programming language. We shall develop this idea in more detail later, but the important point just now is that the intension of every derived category $c/c_1...c_n$ is a function whose arguments are the intensions of the categories $c_1...c_n$ and whose returned value is the intension of the category $c$. Thus, the intensions of all derived categories are functions from functions to functions.

Consider another example, the derived category $S/N$ which approximates our notion of a verb phrase. The intension of a verb phrase is a function which takes as argument the intension of a name, which is a function from possible world states to extensions, and returns the intension of a sentence, which is a function from possible world states to truth values. For example, we have previously derived that 'outwits Moriarty' belongs to the category '$S/N$', which we may think of as the category "verb phrase". The intension of 'outwits Moriarty' is thus a function which requires the intension of a name, such as Sherlock, and returns the intension of the sentence 'outwits Moriarty Sherlock', which, given information about the world, determines the truth of this sentence.

This completes our sketch of Montague grammars. Our principal reason for examining the subject has been to introduce the concept of an intension as a function which may take other
intensions as arguments, and which may return a function which represents the intension of an entire sentence. If we bear in mind that the relevant functions may be procedures which are defined in a programming language, and that the intensions of complex expressions can be built up through the composition of intensions of simpler expressions, we should find it easier to understand the discussions of Winograd's SHRDLU, Dahl's L3, and to a certain extent the discussion of the SHADOW system.

**Terry Winograd and SHRDLU**

One of the earliest, and certainly one of the most famous systems for interpreting and answering natural language questions is Terry Winograd's SHRDLU system [1973]. (SHRDLU performs other tasks as well, but it is the question-answering aspect that concerns us here.) The deductive reasoning component, and the semantic routines of Winograd's system were written in the Microplanner programming language [Sussman et al (1970)], which is, in important respects, analogous to Prolog. Among other things, Microplanner has an automatic backtracking mechanism, an automatic pattern matcher (which is not as general as Prolog's), and a facility for dynamically asserting and erasing data base assertions.

One way in which Winograd's SHRDLU is remarkable is that it integrates syntactic, semantic, and pragmatic analysis in a single pass (including backtracking). The system includes a
syntactic parser possessing a large English grammar, and a fairly large number of semantic routines. The parser is able to recognize, among other things, a variety of forms of conjunction, and the semantic routines correctly interpret an impressive number of natural language quantifiers.

The system also maintains a record of dialogues with its users in order to provide a context in which to resolve anaphoric and elliptic references. Winograd uses both heuristic knowledge about the structure of discourse and "world knowledge" (pragmatic information) to resolve pronoun references.

One way in which Winograd integrates the syntactic and semantic analysis of a sentence is that he uses semantic markers (type restrictions) to help guide the parser and to select the appropriate senses for nouns, verbs, and adjectives when more than one sense is entered in the dictionary. Semantic markers are a device which was first introduced by Katz and Postal [1964], and which have come into common use by computational linguists since that time [Woods (1972), Winograd (1973), Dahl (1981)].

To illustrate the use of semantic markers, let us consider how Winograd uses them to ensure semantic agreement in his analysis of a sentence. In SHRDLU's semantic dictionary the verb 'contain' is listed as expressing two separate relations. The first relation corresponds to the sense expressed in "The box contains the toy" and the second one corresponds to the sense in "The stack contains the cube." The semantic markers listed for
the first relation require that both arguments to the relation be physical objects, and those for the second relation require that the subject of the relation be a construct, such as 'stack', which is capable of having a physical object, such as a cube, as a part [Winograd(1973)].

During the parsing of the sentence, "Which stack contains a cube?", the first relation will be retrieved, and its markers will be checked against the markers which have been retrieved for the two related noun phrases. However, when it is discovered that the semantic markers of 'stack' include the marker 'construct', an incompatibility will be detected and the parser will back up to see whether another sense for the verb can be found. When the second definition for the verb is found, semantic agreement will be discovered and the parse will succeed. In a somewhat analogous fashion, semantic markers can be used to ensure agreement between adjectives and nouns.

We have just seen one way in which Winograd uses semantic information to help disambiguate a sentence before the syntactic parsing is completed. He also uses pragmatic knowledge when disambiguating syntactically ambiguous sentences. This is done by having the program consult its database when certain structures are found. For example, when analysing a sentence such as "Put the blue pyramid on the block in the box", the system will first assume that 'on the block' is a prepositional modifier of 'the blue pyramid'. A Microplanner procedure is then constructed which is capable of verifying whether the system
knows of any blue pyramid on a block, and the verification is attempted. If no such object can be found, the system backs up to try the remaining syntactic path. It will then test whether there is a block in the box which the pyramid can be placed upon. By consulting the state of the world during the actual parsing process, certain paths can be ruled out early in the process.

Another interesting technique which Winograd employs is that of allowing the dictionary definitions to serve a dual purpose. Not only do the definitions include semantic markers, but they also contain the names of procedures which build Microplanner programs, using parts of the sentence as data. Winograd expresses it well: "The definition of every word is a program which is called at an appropriate point in the analysis, and which can do arbitrary computations involving the sentence and the present situation." [1973]

For example, the verb 'supports' has in its definition the name of a procedure which is capable of verifying whether the database either contains or implies the assertion that \( A \) supports \( B \), where \( A \) and \( B \) are the objects related by 'supports'. The procedure is also capable of asserting that \( A \) supports \( B \) if that assertion is not found in the database. The meanings of more complex phrases are represented as Microplanner procedures which are built by the procedures attached to individual words. For example, the meaning of the phrase "a red cube which supports a pyramid" is represented as the goal of satisfying the
following series of subgoals:

1. find an x which is a block.
2. verify that x is red.
3. verify that x is cubical.
4. find a y which is a pyramid.
5. verify that x supports y.

It is Winograd's view that procedures such as this not only enable computers to answer questions which are put to them, but that procedures something like this actually constitute the meanings of questions and commands for human listeners. As Winograd says, "We can think of any utterance as a program - one that indirectly causes a set of operations to be carried out within the hearer's cognitive system." [1973] Some of the operations which Winograd has in mind are such things as, consulting one's memory to verify a statement or retrieve an answer, or adding an assertion to one's memory bank. Winograd clearly implies [1973] that SHRDLU is intended to provide a cognitive model for human language understanding. There is considerable room for debate concerning the degree to which Winograd's model succeeds, but this topic will not be pursued here.
Veronica Dahl and L3

Let us turn now to a system which is in some respects modelled upon the methods of Richard Montague [1970], and which is in some ways analogous to Winograd's approach, but which incorporates a number of novel methods. This system was developed by Veronica Dahl, and is described in [Dahl(1981)]. The entire system (called L3), including a syntactic/semantic dictionary and a database, is written in Prolog, and the system uses the built-in backtracking and pattern matching facilities of Prolog in very interesting ways.

For example, like Winograd and others [Winograd(1973), Woods(1972)], Dahl employs semantic markers to aid in the disambiguation of sentences. However, in L3 this is accomplished by creating typed variables which can only match variables of certain types (this is guaranteed by Prolog's built-in unification algorithm), and by allowing Prolog's built-in backtracker to select exactly those dictionary entries for the words in a sentence which do have compatibly typed variables. In Dahl's system, variables of compatible types do not need to be variables of identical types, for her system includes a hierarchy of types. Thus, if the verb 'eats' requires an animate subject, and a person, P, is typed as human, the system will find the types to be compatible, since 'human' is below 'animal' in the hierarchy of types.
Like Winograd's system, L3 integrates both the syntactic and semantic analysis in a single pass. However, L3 employs Prolog to perform backtracking, rather than Microplanner. In chapter 3 we offer reasons for preferring Prolog to Microplanner, but one advantage which Prolog offers, and which is exploited in L3, is its close relation to Metamorphosis Grammars (MG). (MGs are now a built-in feature of some versions of Prolog.) The name 'Metamorphosis Grammar' may be somewhat misleading, since the name refers both to a formalism for writing particular grammars, and to an interpreter which is capable of translating these particular grammars into Prolog programs. An MG interpreter allows the designer of a language interpreter to write a grammar for a language in a form very similar to conventional phrase structure rules (with certain restrictions upon the form of context sensitive rules) while also allowing both terminals and non-terminals of the grammar to have argument lists which include input and output variables. Furthermore, the phrase structure rules may also include calls to Prolog procedures. A more complete description of MGs can be found in [Dahl(1981)].

The particular grammar which L3 uses is especially interesting in the way it integrates aspects of Montague grammars with traditional phrase structure rules. For, while the non-terminals of the system do not require syntactic categories as arguments, and do not produce syntactic categories as output, (as is common in categorial grammars) they do, in several cases,
require the intensions of certain categories as input and return the intension of a certain category as output. For example, in this system the rules for determiners have the form [1981]:

$$\text{Determiner}(k, s_1, s_2, \text{out1}) \rightarrow \text{det}$$

(where 'det' is a terminal, such as 'the', 'a', 'every', etc.)

In this example, assume that the input variable, $s_1$, is bound to an intensional representation of a noun phrase, that $s_2$ is bound to the intension of a verb phrase, and that out1 is used to return an output intension, which is a function of the determiner, the variable $k$, $s_1$, and $s_2$. Suppose, for simplicity, that the determiner is 'a', that $s_1$ is "student($k$)" and that $s_2$ is "laughs($k$)". Then the value which is returned for out1 could be roughly translated as "for those $k$ which satisfy \(\text{and}(\text{student}(k), \text{laughs}(k))\)\), the cardinality of $k$ is > 0 \). The cardinality of the relevant set of $k$'s is tested at a later stage when the query is being evaluated. At this stage only an intensional description of the set of $k$'s exists.

Other natural language quantifiers are handled in a somewhat analogous fashion; by introducing a variable which represents a set of objects which satisfy conditions which are imposed both by the quantifier itself, and by the intensional representations of a noun phrase and a verb phrase. This method makes it easy to detect certain false presuppositions. For example, L3 tests whether the presupposition implied by the
definite article 'the' is satisfied in 'the professor lives in London' by finding the set of x's which are professors, and testing whether the cardinality of the set is 1. If the test fails, the truth value, 'pointless', is assigned to the statement and no attempt is made to see whether the verb phrase 'lives in London' is satisfied.

When dealing with sentences which contain more than one quantifier, such as "Every boy in the class loves a girl", Dahl employs 3 heuristics to determine how the quantifiers should be ordered in the query which is finally evaluated. While these heuristics are successful in most cases, they do have exceptions, as Dahl notes. Also, the heuristics do not allow for the possibility that quantifier ambiguities may exist in the sentence, due to the fact that different orderings are possible for the quantifiers.

An especially nice feature of Dahl's method is her treatment of distributive and collective relations. For example, her system is able to distinguish the different uses of conjunction in "Ann and John speak Spanish and French", and "A and B are parallel." In the first case the 'speak' relationship is found to be distributive, and it is inferred that the truth of the sentence requires that Ann speaks Spanish, Ann speaks French, John speaks Spanish, and John speaks French. In the second case it is found that the predicate 'are parallel' must apply to the collective set of both A and B, for neither can be parallel by itself. Dahl also notes the special problems created
by respective plural relationships, as in, "Michel and Henri play banjo and guitar respectively."

Before leaving this discussion, we should note another important feature of Dahl's system, namely, that it is easily transferred from one database to another. The system was designed so that all information which is peculiar to a given database is isolated in the lexicon definitions of nouns, verbs, and adjectives, and in a few other places, such as the hierarchy of types.

There are several other significant aspects to this system, which cannot be discussed fully here. However, as will become apparent in later chapters, much of the SHADOW system which is presented in this thesis derives from Dahl's methods.

The Planes Project

Let us turn now to consider a question-answering system which is in many ways quite different from the systems we have so far discussed. This is the PLANES system, designed by David Waltz [1978]. In its overt behaviour, the PLANES system is very impressive. It accepts a wide variety of syntactic and semantic phrasings, including compound sentences, relative clauses, and comparative phrases. It can resolve several forms of pronoun reference and ellipsis, and accept a considerable range of non-grammatical input. Furthermore, it can perform spelling corrections and permits the user to introduce abbreviations and
expand the dictionary. PLANES also possesses the ability to return a paraphrase of a user's query, enabling the user to decide whether the query has been correctly interpreted. Other features include allowing the user to specify whether a graphical or list-type output is desired.

There are two principal factors which help to explain the breadth of PLANES' performance. First, it was a large project, involving a group of people over a few years. Secondly, the project dealt with a restricted domain, which enabled its designers to avoid certain general problems of human language comprehension. Although the systems we have so far examined have also dealt with restricted domains, both Winograd and Dahl were more concerned with the general problem of language comprehension than Waltz. For example, both Winograd and Dahl attempt to deal with lexical and syntactic ambiguity in a systematic and general way, and both integrate the syntactic and semantic meaning of a sentence in a way that seems plausible and satisfying. Waltz, on the other hand, explicitly asserts that the PLANES system is not designed to model human language understanding in a meaningful way. Nevertheless, PLANES employs some interesting techniques, which we will now summarize.

PLANES processes a query in five different stages: syntactic and semantic parsing, query generation, query optimization, query evaluation, and response generation. The parsing is performed by a variant of the kind of ATN parser described by Woods in [Woods(1970)] and used in LUNAR [Woods, et
The ATN parser is not purely syntactic, in that sub-networks in the ATN can be taken, in most cases, only when certain semantic conditions as well as syntactic conditions are satisfied. The ATN includes a different sub-network for each different kind of entity which the database contains. Consequently, a sub-network which could parse noun phrases such as 'defective aircraft' would be different from one which parses a noun phrase like 'broken propellor'. Waltz admits that this strategy is made possible only by the very limited domain of the PLANES project. An exception to this "domain dependent" parsing is the way in which relative clauses are parsed in PLANES.

In addition to semantically restricted subnets, the PLANES system also uses "concept case frames" to guide the semantic interpretation of a query. These case frames differ from case frames as usually conceived, in that they do not contain information about agents, objects, and instruments, etc. Rather, they seem to resemble lists of type restrictions which are placed upon primitive acts or relations which the system recognizes. For example, the primitive act "require" might be associated with the case frame "(require, planetype, maintainancetype)". Waltz also explains that these case frames "enumerate the patterns of questions which can be understood by the system" [1978]. When new question patterns are to be added to the system, new concept case frames must also be added.

The concept case frames help constrain the semantic interpretation in two ways: (a) they assist in the resolution of
pronoun reference in those cases where the reference cannot be determined solely on the basis of the "context registers" which the system maintains, (b) they are used to help determine the types of missing phrases, in the cases of elipsis. Context registers can be searched for phrases having types which match the types required by the missing elements in the case frame.

A curious feature of the PLANES system is that the output of the syntactic and semantic interpretation stages is an unordered set of semantic elements which have been put into canonical form. Except in the case of relative clauses, and comparative relations (such as 'greater than'), the system discards syntactic information, and attempts to reconstruct the query from this unordered set on the basis of (1) knowing what canonical types are required by the relations present in the database, (2) knowing the canonical types of the semantic elements present in the unordered set. The reconstruction of the intension of the original query is made possible only by the fact that each relation in the database does not contain more than one field of a given semantic type within itself. For example, if the relation 'loves' were included in a database, and the values of its fields were both required to be human, then the system would be forced to take syntax into account, as it does for comparative relations, which hold between objects of the same type (numbers).

Consequently, it appears that the PLANES system is unduly domain dependent, insofar as it discards information which is,
in general, necessary for the interpretation of a sentence, and replaces it with information which is, in general, only available in special circumstances. Although Waltz explicitly states that PLANES does not attempt to model human language comprehension, it seems imprudent to construct a system which may have to be modified in a basic respect if certain new relations are added to the database. The practice of ignoring syntax may be necessary in cases of ungrammatical input, but where correct syntax is discovered, it seems wasteful to discard it.
III. A COMPARISON OF TWO GENERAL APPROACHES

In this chapter two quite different approaches to the automated interpretation of database queries are discussed, and reasons are presented for deciding between them. The first approach is loosely modelled upon the methods used by Woods, Kaplan and Wash-Webber, in their now famous LUNAR project [1972], and the second approach is a synthesis of methods which we have seen in our discussion of Dahl's L3 [1981], and Winograd's SHRDLU [1973]. Let us begin by briefly outlining the general strategy adopted in the LUNAR system.

LUNAR divides the process of interpreting queries into two separate stages. The first stage uses an ATN parser to perform a completely syntactic parsing of the query. The output of the first stage is a syntactic parse tree of the query, written in Lisp. The second stage involves semantically interpreting this parse tree, by both semantically interpreting its terminal nodes (through consulting the lexicon) and by considering the syntax of the parse tree itself. The output of this semantic interpretation stage is a procedure which is capable of retrieving the requested information from the database. We may think of this procedure as a query posed in a formal query language whose meaning is "understood" by a data base management system.
In the LUNAR project the process of semantically interpreting the parse tree involves matching "semantic templates" to portions of the tree, and taking structure building actions when a match is found. The structure building actions eventually result in the construction of a formal database query. The semantic templates which are selected for matching against portions of the syntax tree are themselves tree structures which impose semantic restrictions on the terminal nodes. These semantic restrictions are quite specific. For example, the restrictions on the terminal nodes may require that the lexical entry on a subject node have the semantic feature "human" while the entry on the verb node may be required to have the semantic feature 'write' [Woods, et al (1972)]. This strategy clearly can lead to a large proliferation of semantic templates. However, other semantic strategies are possible. Therefore, in the following discussion, we will be concerned not so much with the specific manner in which LUNAR performs semantic interpretation, as with the higher level strategy of dividing the query interpretation process into 2 distinct phases, the construction of a syntax tree and the semantic interpretation of this syntax tree. We will refer to this strategy as the two-stage approach.

The second approach, which is the one adopted in the SHADOW system, involves integrating the process of syntactic and semantic analysis into a single phase or set of phases, so that semantic information is consulted before the syntactic parsing
is completed. Following Dahl, this is performed here by writing a syntactical analyser in Prolog, and interweaving calls to Prolog procedures which build Prolog predicates to represent the information which is semantically implied by the words in the input query. These Prolog predicates are assembled into a formal Prolog procedure invocation which answers the original query by consulting a Prolog database.

The above is a somewhat sketchy description of two possible approaches to interpreting natural language, but more details will emerge as we consider reasons for preferring one to the other. I will begin by discussing my reasons for preferring the integrated Prolog approach, and then go on to discuss whether the two-stage approach might offer some countervailing advantages.

Perhaps the greatest advantages which the integrated-Prolog approach has to offer are (1) the automatic backtracking features of Prolog make it ideal for solving the problem of interpreting natural language (NL), since the problem, by its nature, involves some backtracking; (2) integrating the syntactic and semantic stages of interpretation is more efficient than the two-stage approach since it usually requires only a single pass, and since it eliminates the need to create and interpret parse trees. Let us consider these reasons in turn.¹

¹ Microplanner could also be used to achieve these two goals, but Prolog offers features which are unavailable in Microplanner. For example, Prolog’s pattern matcher is based upon the Unification algorithm, which is the most general
It seems clear that both the problem of syntactically parsing a sentence and the problem of semantically interpreting a sentence involve exploring various possibilities and backtracking when certain possibilities fail to "pan out". For example, consider the sentence, "This can of beans is edible". In attempting to interpret this sentence, as it is spoken to us, we would naturally begin by supposing that the thematic subject of the sentence is 'can'. If the word 'heavy' had appeared instead of the word 'edible' (in the predicate), this assumption would be correct and no backtracking would be necessary. However, when the word edible is encountered we immediately realize that 'edible' is modifying 'beans' and not 'can', because the semantic features of 'edible' are not compatible with those of 'can'. This may be more a question of world knowledge than of semantics, but this need not concern us here since our "semantics" need not be "pure semantics". Once 'can' is rejected as the theme of the sentence, we backtrack to the last point of choice and try to find another candidate for the subject, which is obviously 'beans'. This backtracking seems unavoidable unless we carry along all the other possibilities as we progress through the sentence. But carrying along all possibilities does not, in general, involve exploring any fewer

\[\text{1}(\text{cont'd}) \text{ pattern matcher known. This pattern matcher allows the assignment of values to variables to be postponed in a very convenient way. Furthermore, Prolog includes certain built-in features, such as the "cut" operator, which allows the programmer to control backtracking in ways not possible with more limited Microplanner features, such as THUSE.}\]
paths than backtracking, and it does involve storing more information than is usually necessary. It would seem that backtracking is the most reasonable course to follow here, and utilising a language which automatically performs backtracking enables the programmer to focus attention upon higher level concerns, such as dealing with more complicated forms of syntax.

(2) Systems which employ the two-stage approach to query interpretation also employ backtracking, but in different ways. We need to distinguish two different forms of the two-stage approach; let us call them form A, and form B. In form A, all possible parse trees for a given query are constructed before any semantic interpretation is performed at all. Then an attempt is made to semantically interpret the first parse tree. If the attempt fails, we back up and try the second parse tree, and so on. This is clearly not an efficient method, since, in most cases, the ATN can be constructed so that the first parse tree generated will be successfully interpreted by the semantic routines [Woods, et al (1972)]. In this case the construction of additional parse trees is a waste.

In form B of the two-stage approach, (which is adopted in LUNAR) additional parse trees are constructed only when an attempt to semantically interpret an earlier parse tree fails. When such failure occurs, the system backs up to the last point of choice in the ATN analysis, which has been saved on stacks, and a new path through the transition network is sought. A new parse tree is constructed if an alternative path is found. This
new tree is then passed to the semantic routines.

Now, while form B of the two-stage approach avoids creating many superfluous parse trees, it does not entirely avoid our second criticism, which is that, in the great majority of cases, there is no need to construct any parse tree, and no need to "parse" the parse trees by means of semantic templates, or other means. For we have seen existing systems which completely analyse and answer a wide range of queries without the construction of parse trees.

Let us turn now to a different kind of efficiency which is offered by systems written in Prolog - the efficiency which is provided by using an inferential data base. One very nice aspect of inferential data bases is that they can often remove the necessity for storing redundant information [Dahl(1982)].

For example, suppose we have stored information stating which department in a university each professor belongs to by using assertions of the form: "member (prof-name, dept-name)." And suppose that all but two professors actually are members of some department. Then for nearly all professors we can establish that they are professors simply by finding out whether they belong to some department. For the remaining two "free floating" professors we need to explicitly store the information that they are professors, say, by using statements like, "Prof(Oddball)" and "Prof(Wierdo)."

Our database may now contain, in addition to these two assertions, a whole list of assertions of the form: Member(x,y).
It will also contain the assertion, "If member(x,anydept) then prof(x)". By using this last assertion as a premise in inferences, we avoid a large number of additional assertions. A traditional database, however, would normally explicitly store information stating who each and every professor is.

It is worth noting that this is not a contrived example, for this situation exists in our own university. Of course, avoiding redundancy is only one advantage offered by inferential databases. Their greatest advantage is that they permit the deduction of information which is simply not retrievable by means of union and intersection operations, or any other DBMS operations.

Related to the fact that Prolog offers all the power of an inferential database is the fact that the assertions in a Prolog database, together with the if-then assertions in the database, may be seen as constituting a semantic network, or nearly so, in the sense described by Schubert, Cercone, and Goebels, in [1979]. The semantic networks described in that paper are isomorphic to clause-form assertions in the Predicate Calculus, just as the assertions in a Prolog database are. The order of predicates and operators differ in the two systems, but that has no real import. The networks developed by Schubert, Cercone, and Goebels are linked together in topic hierarchies, and in other ways, in order to facilitate inference and information retrieval, but to some degree this can be simulated in Prolog. It is easy to create topic hierarchies in Prolog by
using rules such as, "x is a bird if x is a canary", and "x is an animal if x is a bird". The use of back-chaining in Prolog makes it easy to trace connections between related predicates. Of course, even back-chaining can run into exponential explosion, and some of the problems of associative networks are not solved by Prolog.

Another linkage mechanism in Prolog is that assertions which begin with the same predicate name, e.g., 'Bird', in 'Bird(x)', are linked together automatically by the Prolog interpreter. Assertions in Prolog are located by hashing, and axioms which share the same hash address are linked together to resolve collisions. Thus, at least some of the reasons for using semantic networks are satisfied by using Prolog databases.

Let us now turn to the question whether the two-stage approach to query interpretation might offer advantages over the single pass approach which SHADOW adopts. One possible advantage of the two-stage approach is that, since the construction of a formal query occurs at a stage which is clearly distinct from the syntactic analysis, and since there is no need to suppose that the query being constructed is framed in the same language as the query analyser, it may be easier to construct a "language-neutral" output query, which is passed to another program which performs query optimization, before any attempt is made to answer the query. Also, it may be easier to generate paraphrases of the original query given a "language-neutral" query as the output of the query analyser.
To respond to these points we need to realize that the SHADOW system does, in fact, produce a formal query which can be subjected to an optimization process before it is evaluated. Furthermore, as we shall see in chapter 4, Shadow performs some optimization during the analysis of noun phrases, and further optimization is still possible. The problem of generating paraphrases from a Prolog representation of a NL query has not been tackled in this system, but there appears to be no intrinsic difficulty in generating paraphrases from Prolog queries. It may in fact turn out to be easier than generating English from formal database queries. Clearly, it is not difficult to move from the type of expressions which the system now assembles, such as (find-all, x, prof(x), teaches(x,y), course(y)), to something like "find every x which satisfies: x is a professor, x teaches y, and y is a course."

This paraphrase is not quite English, but it could be converted to English without much trouble.

Another advantage which the two-stage approach might have is that the one-pass, integrated approach may very well not be capable of analysing certain complex types of constructions - compound sentences, for example. It may be necessary (and indeed the experience of Dahl and McCord supports this [forthcoming]) to perform a purely syntactic analysis of certain sentences before performing any semantic analysis. If this is so, is it not a serious problem for the integrated approach?
While it must be conceded that the integrated approach may not work for all queries which can be posed in English, the approach does work for the subset of English which is most commonly used when posing queries, and that is approximately the subset of English processed by the present system.

Furthermore, the integrated approach could be supplemented by a fully developed Definite Clause Grammar parser (DCG), which has been shown to be as powerful as any ATN parser [Pereira and Warren (1980)]. It is feasible, therefore, to use the integrated approach for the vast majority of sentences, and to switch to a two-stage DCG analysis when the integrated approach fails. So, let us consider in some detail, what DCGs are, and what their merits are.

Briefly, DCGs are normalized grammars (only one non-terminal may appear on the left-hand side of any rewrite rule), which, by virtue of being normalized, are always representable in Horn clause form (Horn clauses are conditionals having a single "literal" as the consequent). For example, the simple context-free rule,

\[ S \rightarrow NP + VP \]

can be represented in Horn clause form by

\[ S(p1, \text{nil}) \text{ if } NP(p1, p2) \land VP(p2, \text{nil}). \]
which we may translate roughly as follows: There is a sentence in the input list p1 if we can parse an NP in the list p1, from its starting point up to the sublist p2, and we can parse a VP in the sublist p2 up to the sublist whose only member is nil. When we have parsed a complete sentence in P1 the sublist, nil, will be all that is left.

Context sensitive rules are also representable in Horn clause form, by using constants as some of the elements in the input and output lists of the predicates (or procedure names). Furthermore, the procedure names, NP, VP, etc., may take arguments which act as flags of various kinds. In fact, in addition to parsing a portion of an input string the procedures may have any number of other input and output variables. Also, these procedures may test any number of conditions and take any number of actions, (such as setting registers and building syntax trees or semantic structures). For this reason it can be seen that definite clause grammars are equivalent in power to Horn clause logic, which is the same as the power of Prolog itself. And it has been shown [Andreka, et al (1976)] that Prolog has the power of a Turing machine.

Furthermore, F. Pereira and D. Warren have described straightforward methods for translating any given ATN into a definite clause grammar. I will not describe their methods here, but the interested reader is referred to (10). Pereira and Warren also argue that DCGs are at least as efficient as ATNs while being more perspicuous and more powerful in practice. The
argument they present for efficiency is mainly theoretical (rather than experimental). The two strongest theoretical considerations which they present are (a) that DCGs are in fact Prolog programs and consequently do not need to be translated into programs as ATNs must be translated into programs by an ATN compiler. (b) Prolog builds structures such as parse trees in a more efficient fashion than Lisp, since, in essence, it builds complex structures by sharing the lower level structures (through resetting pointers) rather than by copying simpler structures. This last point seems valid, but the first point is surely mitigated by the fact that an ATN compiler need only be invoked once to create a program capable of parsing an infinity of sentences.

The claim by Pereira and Warren that DCGs are more perspicuous than ATNs is also somewhat doubtful. It is true that one way of representing DCGs is very perspicuous, and this involves providing a set of rules very similar to a BNF grammar which is translated into a Prolog program by a DCG interpreter. But when DCGs are represented in this fashion they no longer have the advantage described above, namely, that they are already programs.

Furthermore, perspicuity is a rather subjective matter, which depends to a large extent upon one's familiarity with a particular approach to problem solving. One who has been trained to think in Lisp does not always find Prolog a perspicuous language, and vice-versa. Pereira and Warren may have a stronger
case when they argue that DCGs are more powerful than ATNs in practice. Their reasoning is that, since ATNs use Lisp functions, POP arcs can only return single values, whereas procedures in DCGs can return several values at once, and some of their output variables may be left unbound. This last fact is very useful, for it allows us to postpone the assignment of certain values to certain variables until a later point in the parsing, where more information is available. In effect, this eliminates the need for certain registers, and it permits more to be discovered in a single pass, without any backing up, than one would imagine possible.

On balance, it seems clear that ATNs do not possess any intrinsic advantage over DCGs, even when one is dealing with context sensitive languages. If there are sentences which require a purely syntactic parse before any semantic parse is performed, this can be done by DCGs as well as by ATNs. As noted, however, the system being presented in this thesis is restricted to a subclass of English which can be analysed in a single pass.
IV. ALGORITHMS

General Strategy

In the preceding chapters, certain elements of the general strategy adopted in the SHADOW system have been discussed. In this section we illustrate in greater detail how these strategies are implemented.

Perhaps the most important aspect of the SHADOW system is the way in which semantic information is extracted from a question during the syntactic analysis, and how this information is integrated into a Prolog query by the same procedures which perform the syntactic analysis. In our discussion of Dahl's L3 [1981], we saw how it is possible to construct procedures which parse determiners (natural language quantifiers) in such a way that the procedures not only "consume" a determiner in the input string, but require, as arguments, predications which represent the information contained in the noun phrase and verb phrase of a sentence. These predications involve an explicitly named variable, \( k \), as well as other embedded variables. The determiner rules take different actions, depending upon what determiner is found, but they always return a Prolog procedure which involves the variable \( k \) and those predications which were passed as arguments. In L3, the procedures which are returned by
the determiner procedures always test the cardinality of the set of all \( k \) which satisfies a truth-functional compound involving the argument predications.

A somewhat different technique is employed here. In SHADOW, generally speaking, each syntactic procedure which consumes a portion of the input returns intensional information only about the portion of the input which is consumed by that procedure. For example, the rules which consume determiners in the input string will return just a value for the determiner which is found, or an unbound variable, in those cases where the determiner's value must be ascertained after the "number" of the noun which follows it is known. Likewise, the procedures which parse noun phrases in SHADOW only return information which is extracted in the noun phrase which is consumed. However, those procedures which consume entire sentences invoke several subprocedures, which return semantic information about the portion of the input they consume. Then, after all this information is available, the Sentence procedure invokes procedures which perform compatibility checks on the semantic markers of the top-level nouns and verb, and which construct predications involving all the information which has been extracted so far. Compatibility checks are also performed by the relative clause procedures whenever an embedded sentence is discovered.

Sentences which are recognized by the system fall into the following two patterns:
S -> Question-phrase + Noun-phrase + Fancy-verb + Noun-phrase +
    Adverbial-Phrase.

S -> Question-phrase + Noun-phrase + Auxiliary-verbs +
    Noun-phrase + Fancy-verb + Adverbial-phrase.

The term 'Fancy-verb' should be understood as referring to
a verb group, which may include auxiliaries, the negative
operator 'not', and certain particles such as 'by' and 'to'. The
definition of 'noun phrase' is such that a noun phrase may be
missing altogether, or be represented by an interrogative pronoun
('who' or 'what') in the subject position. However, at least one
noun phrase must be explicitly present in the top-level of a
sentence. The adverbial-phrase procedure has been designed so
that it recognises both "real" adverbial phrases, and empty
strings (in cases where no adverbial phrase is present).
Adverbial phrases may be embedded in relative clauses as well as
in the main clause.

In SHADOW's program there are two Prolog conditionals which
correspond to the phrase structure rules shown above. The manner
in which these rules are represented in Prolog is described in
more detail later. At this point we need only note that invoking
the Sentence procedure, with a question as an argument, causes
each of the sentence rules to be tried in turn, until a
successful parsing is found. When a given rule fails to
recognise the input argument, Prolog automatically selects the
next sentence rule. (SHADOW's complete grammar is displayed in Appendix B.)

Before we consider more details of how the system works, it will be helpful if we describe very abstractly how the system extracts semantic information from verb groups and noun phrases.

One very useful fact about verbs which appear in main and relative clauses is that they nearly always express a relation which may hold between the different noun groups which surround the verb. Following Winograd and Dahl we have chosen the relation names which appear in our Prolog database so that they closely correspond to verbs which are likely to occur in users' queries. The main exception to this rule is that certain forms of the verb 'be' are not represented as a relation in the database, but are interpreted as expressing the identity relation. At present, SHADOW does not recognise non-auxiliary forms of 'be' which do not express identity or set membership. Forms of 'be' which express set membership are assimilated to the identity relationship. For example, "Is it true that X is a professor?" is interpreted as "Is it true that X is identical to some professor or other?" When verbs other than 'be' are successfully parsed in the program, an internal relation name, corresponding to the verb, is retrieved (along with other information) and this relation name is used when constructing the formal query which is eventually evaluated.

The retrieved relation name is used when constructing a complete relational predication, involving variables which
represent the nouns which are related by a verb group. These variables are introduced by the procedures which parse noun phrases and adverbial phrases. When a noun phrase is parsed, a variable name is arbitrarily chosen to represent the main noun, and the remaining information in the noun phrase is transformed into a list of predicates which contain the chosen variable. This is true even when the noun phrase includes relative clauses.

Relative clauses can be transformed into relational predicates, one of whose arguments is the variable representing the single main noun in the noun phrase. Another useful fact is that the determiner in a noun phrase, if there is one, can be transformed (by taking some contextual factors into account) into a quantifier of the variable which represents the main entity described by the noun phrase. A noun phrase which immediately follows a question phrase may lack a determiner, but in this case the question phrase, such as 'how many', or 'which', or 'what are the', may include or imply a determiner which can be transformed into an "operator" which can be applied to a variable, and a set of predications about that variable, to yield an answer to the question.

For example, a question of the form "Which x's ...?" will impose a set of restrictions upon x, which can be translated into Prolog predicates, and the phrase "Which" can be transformed into something like 'Find-all' which can be used to select a set of Prolog clauses which can print out all the values for x which
satisfy the set of restrictions on \( x \).

We now have a very sketchy idea of how SHADOW analyses a question. Let us turn now, to a somewhat more detailed description of this process, before we become immersed in explanations of particular procedures. We first consider the actions taken by the first of the two sentence rules (conditionals) which the program possesses. The actions taken by the remaining rule are basically a reordered set of those taken in the first rule. First, the Question phrase is identified. Usually, it indicates what the determiner or operator for the entire sentence should be. Sometimes, as is the case with 'Is it true that', the operator is indicated by the determiner which is explicitly present in the subject's noun phrase. Next, the subject noun phrase is analysed, and a variable name representing the main noun, and an "intensional description" (a list of predicates involving the variable name) is obtained for the noun. An exception to this is the case where the noun present is a proper noun. In general, whenever a noun phrase consists of a proper noun, the variable name which is returned to represent the noun phrase will be bound to the proper noun itself, and the intension returned for the noun phrase is the predicate 'true', which is always satisfied. Next, the verb group will be analysed, and an internal relation name for the verb is found. This relation name, together with the verb's voice and tense, and a flag indicating whether the 'not' operator was present, is returned to the top level of the
Subsequently, the object noun phrase is analysed to obtain a variable and an intensional description of the principal noun in the object phrase. Then, an attempt is made to discover an adverbial phrase in the sentence. If one is found, a variable and intension are returned to represent the information which was analysed there. Otherwise, nil is returned both as the value of the variable and the value of the intension. When an adverbial phrase is present, the variable which is returned is not exactly like those returned for noun phrases, but we defer the discussion of these details until later.

Along with the intensional information which is returned for each noun phrase and adverbial phrase which is parsed, a semantic type is also returned. Once the Sentence procedure has completed all syntactic parsing, an ordered list of all the semantic types which were retrieved is compared to a list of types required by the main verb in the sentence, to determine whether they are compatible. If not, other possible senses for the verb and nouns may be tried until a compatible reading is found. If none is found, the parsing of the sentence fails. Whenever a compatible reading is found, however, an ordered list of the known variables is constructed, which takes into account the original order in which the variables were discovered and the voice of the verb group which was parsed. This ordered list of variables is passed, along with the main verb's relation name, to a procedure which returns a predication composed of the analysis.
relation name and the list of variables.

Once the principal relation has been composed, it is passed, along with the operators, intensions, and variables of the subject, object, and adverbial phrases, to a procedure which composes a formal query, which is finally evaluated. This procedure takes different actions, depending upon whether the operators (quantifiers) for the subject and object phrases reveal that a different order of quantifiers should be used from that presented in the surface structure of the sentence.

As a general rule, however, the following occurs: a predicate is composed using the principal relation and the quantifiers, variables, and intensions of the object and adverbial phrases. The variable associated with the subject phrase also occurs in this predicate, but unbound by a quantifier. The remaining variables associated with the object and adverbial phrases are quantified in this predicate. Thus, we may think of the predicate as being a predicate "about" the subject's variable, since this variable's quantifier will appear outside this predicate. Finally, the quantifier (or operator, such as 'find-all') which belongs to the subject phrase is applied to the variable representing the surface subject, the intension of the surface subject, and the predicate which was just constructed.

For example, if our query was "How many math courses are taught by Euclid this semester", we would apply the operator "Count-all" to the arguments, variable, intension, (which
represents the information that whatever is bound to variable1 must both be offered by math and be a course) and to a predication which is constructed both from the relation 'teaches(variable2, variable1, variable3)' and from the intensions and operators which constrain variable2 and variable3. In this case, variable2 is already bound to 'Euclid' and variable3 is already bound to 'Fall83', as we shall see later.

More Specific Algorithms

We turn now to some of the mechanics of how the SHADOW reads a user's query, and how this query is processed by the Sentence procedures. The highest level procedure in the program, 'Start', is used to print instructions to the user, to fetch the user's questions, and to pass the questions to a high level procedure called 'Ans' which analyses and answers questions. Start calls upon a procedure, Get-ques, which reads a single question and converts the question to a list of atoms. In the process, phrases such as 'computing 354' are converted to atoms such as 'cmpt-354'. Once a question has been put into list notation, the Ans procedure passes it to the Sentence procedure to be analysed and answered. If the question cannot be properly analysed, or if every possible interpretation of the question contains a false presupposition, the Ans procedure prints messages to the user, explaining why the question cannot be
answered in a normal fashion. In the case of false
presuppositions, the messages are created and saved during the
process of evaluating the query. Only after it is known that all
interpretations for the question contain a false presupposition
is any message printed.

When the ans procedure invokes the Sentence procedure, it
poses the high level goal "Sentence(input-list, nil)", which may
be read as follows: "Parse and analyse a sentence beginning with
the input-list, and when you finish, there should be nil left
over, i.e., all the input should be consumed." This high level
goal generates a series of subgoals, which may be rendered into
pseudo-English somewhat as follows:

1. Find a Question Code

Beginning at the start of the input list, find a
question phrase between the start and point $x_1$. Return a
code for the question type in the process.

2. Parse the Subject Noun Phrase

Starting at point $x_1$, find a noun phrase in the
remaining list between points $x_1$ and $x_2$, and in the
process return a variable to represent the main noun, an
operator (determined on the basis of the question type
and the number of the main noun), an intension to
represent the meaning of the noun phrase, and a type for
the main noun.

3. Parse a Verb Group

Starting at point $x_2$, find a verb group (active or
passive) between $x_2$ and $x_3$, returning in the process a
name for the relation implied by the verb, its voice and
tense, and a flag indicating whether 'not' was present.

4. Parse an Object Phrase

Beginning at point $x_3$, find another noun phrase between
$x_3$ and $x_4$. Return for this noun phrase values for:
operator2, variable2, intension2, and type2.
5. Parse an Adverbial Phrase

Between point \( X \) and the end of the list, which is marked by nil, parse an adverbial phrase (which may be absent). For this phrase return a type, e.g., 'semester', a variable which is bound to an infix predication of the form, \( S \& Y \), an intension for the adverbial phrase, and a variable which occurs both in that intension and in the infix expression. This variable is used in cases where the intension requires us to quantify over \( S \) or \( Y \), where \( S \) stands for a season and \( Y \) stands for a year.

6. Check the Compatibility of Types

Check whether the types of the noun phrases and adverbial phrase are compatible with those required by the verb's relation name. Take order and voice into account, and return an ordered list of variables which should be used with that relation name. Also, return the name of the appropriate type of the subject noun phrase in case that type was previously unknown, due to the subject's being represented by a pronoun.

7. Compose a Formal Query

Using the information so far obtained, compose a query in which the outermost quantifier or operator is the one belonging to the subject's phrase, the second-highest order quantifier is the one belonging to the adverbial phrase (if the quantifier is not nil), and the innermost quantifier is the one belonging to the object noun phrase. However, if the subject's quantifier is 'some', (but not 'some-particular') then reverse the roles of the subject and object phrase.

8. Evaluate the Formal Query

Evaluate the query which was constructed in the previous step.

The eight steps just outlined are used to analyse sentences where the overall pattern is:

question phrase \( + \) noun phrase \( + \) verb group \( + \) noun phrase \( + \)
(possible) adverbial phrase.

The other general sentence pattern recognised by the program is:
question phrase + noun phrase + auxiliary group + noun phrase + verb group + (possible) adverbial phrase, e.g.,

"Which courses will Beauvoir be teaching in the spring?"

The same procedures are invoked by this second Sentence rule as are invoked by the first Sentence rule, except that the second rule invokes the Aux procedure at the top level, whereas the first rule invokes the Aux procedure indirectly, when it analyses the verb group. Also, the second sentence pattern is always treated as a passive sentence, since the deep subject of such sentences appears as the second noun group. Both the first and second Sentence rules compose a final query in the same fashion, however, by invoking the Make-query procedure after all parsing has been completed.
V. SPECIFIC PROCEDURES

We will now consider in detail how these various procedures are implemented in Prolog. The logic of the more important Prolog procedures will be described, but the minor procedures will not be explained since most of them are described in the program listing in Appendix C. In this chapter we subdivide our discussion to reflect the flow of control in SHADOW. The following topics will be discussed:

1. The Question Phrase
2. Analysing Noun Phrases
   2.1 Analysing Relative Clauses
      2.1.1 Checking Type Compatibility
      2.1.2 Composing a Sub-Query
   2.2 Checking Quantifier Ambiguity
3. Noun Phrases with Prepositional Modifiers
4. Finding "Dummy" Noun Phrases
5. Parsing Verb Groups
6. Analysing Adverbial Phrases
7. Composing A Top-level Query
8. Evaluating A Top-level Query
The Question Phrase

We begin with the question phrase. A set of Prolog statements named Quest-phrase inspects the first several words in the input list to see if certain recognised combinations occur. If not, the parsing fails at this stage. Otherwise, the question phrase is translated into a question code, which is usually just a hyphenated version of the phrase. This code is returned via an output parameter of the Quest-phrase predicate.

The Subject Noun Phrase

Next the NP (noun phrase) rules take over. The first NP rule tries to parse a noun phrase involving a common noun. If that fails, a second NP rule is tried which parses proper nouns. Let us consider the proper noun rule first, since it is short. This rule checks to see if the word beginning its input list has a synonym. For example, 'mathematics' can be a proper noun representing a department name, and its internal synonym is 'math'. If a proper noun has no synonym, e.g., Gauss, then the noun is treated as its own synonym. The synonym is then checked to see if it has a basic type in the database. Gauss, for example, has the type 'prof' since Gauss appears as a member of some department. This is determined by the Get-Type predicate. If a noun is found to have a basic type in the database by the Get-type predicate, then it is always a proper noun. In this
case the NP predicate returns the type of the noun, and 'the' is returned as the operator, with "true/nill" as the intension, and the proper noun itself (or its synonym) as the value which must satisfy the intension, which it does trivially.

If the noun phrase cannot be parsed by the proper noun version of NP, then it may be recognisable by the common noun version of the rule. This rule begins by trying to parse a determiner in its input list. The NP predicate is passed a code for the question type to help in determining the determiner. If the Qtype (question type) is nil, then we are not looking for a surface subject and a genuine determiner must be present. It may be either 'the', 'a', 'some', 'every', or, when preceded by a verb group, 'no'. The determiners 'some' and 'a' will be translated into the operator 'some'. 'Every' and 'no' remain themselves, and 'the' returns an unbound variable as an operator, since the kind of operator will depend upon whether the main noun is singular or plural. If singular, the operator will be set to 'the'; otherwise it is set to 'some' or 'every', depending upon whether the determiner follows a form of 'to be'.

If the Qtype is not nil, then we are parsing a subject noun phrase and the Qtype will determine what the operator should be (with the exception of 'which', which is similar to the way we handled 'the'). For example, the Qtype 'Who-are-the' tells the Determiner rules to return the operator 'Find-all'.

Once the determiner or operator has been found, the NP predicate looks for a series of zero or more adjectives. Each of
these adjectives is checked to see if it is valid and if it has a synonym, and all valid adjectives are returned in a list. As soon as a non-adjective is discovered, control returns to the NP level, where the Noun predicate begins to look for a valid common noun. Once again, synonyms are checked. A noun is located in the dictionary under an entry of the form

Get-noun( nounword, number, type(X), type, X),

which allows us to tell immediately whether the particular noun is entered in the dictionary, what its number is, what intensional predicate should be used to represent it, and what variable and type represent the noun. This information is made available to the NP procedure, which next takes the type of the noun it has discovered, together with the list of adjectives which have been discovered, and passes both of these to the Select-Adjs predicate, which returns a list of the form

Bind(type, adj1)/Bind(type, adj2) ... /Bind(type, adj-n).

This list becomes part of the intension which will be returned for the noun phrase as a whole. Later, each member of the list is made into a predicate of the form Adj(adj-n, type, X), and when an attempt is being made to find a value for X which satisfies all predicates which appear in a given intension (this is done by the Satisfies rules) the dictionary will be searched for an entry which unifies with this predicate. For example, in the dictionary we find a rule of the form

\[ \text{Adj(adj, course, X) if OFFERS(adj, X, Y)} \]

which means that an adjective which requires the type 'course'
can be true of I if we can show that the database contains an assertion (or if we can derive the assertion) that I is offered by the department whose name is the same as that adjective. It happens that in our database all adjectives so far entered can modify only courses, departments, or professors. Each of these adjectives is a department name. But the method of entering adjectives in the dictionary, as illustrated above, is absolutely general and can be used even if an adjective has no type restrictions, since we can always use an unbound variable where the type value is normally placed.

Analysing a Relative Clause

Let us return now to the NP procedure, which has just turned a (possibly empty) list of adjectives into a list of predicates. This list is saved for a while, while the relative clause predicate is called upon to process any relative clauses that may be present. The Relative-clause procedure begins by looking for a relative pronoun. If none is present, control returns to the NP procedure and no input is consumed. Otherwise, a verb form is searched for in the fashion described earlier. If a verb is found, its relation name, voice, tense, and negation flag are temporarily saved. Then a noun phrase is searched for by calling upon the NP procedure once again. This call introduces the possibility of a more deeply embedded relative clause being discovered, but that is not our concern here. The
call to the MP procedure, if it succeeds, will return an intension, operator, a type, and a variable for that noun phrase. Next, an attempt is made to parse an adverbial phrase, and if one is found, relevant information is returned to the Relative-clause procedure. At this point, the type of the adverbial phrase just parsed (which may be nil), together with the type of the preceding noun phrase and the type of the noun phrase at the higher level (which has been passed downwards when the call to Relative-clause was made), are all passed to the Compat procedure, to see whether the types are compatible with the verb relation which was found. The operation of the Compat procedure is a bit subtle, so we digress briefly to discuss that procedure.

Checking Compatibility of Types

The Compat procedure is given a verb relation, an ordered list of types, an ordered list of corresponding variables, and the voice of the verb. It retrieves the argument types required by the verb relation, and it performs a subtle pattern match with these types and the ordered list of types it has been passed. To begin with, it takes the voice into account before attempting to match the types. Also, it will allow the types 'dummy' and 'nil' to match any constant. This is necessary to accommodate dummy noun phrases which are introduced to take the position of implicit noun phrases, and to accommodate optional
phrases, such as adverbial phrases. If the pattern match is successful, Compat returns a list of the original variable names, reordered to take voice into account. Compat also returns the actual type of the surface subject of the relation in those cases where the given type was 'dummy'. The subject type is needed in cases where negation is present, for reasons which we explain later.

Returning now to the Relative-clause procedure, once the compatibility test has been passed, a relation is composed of the relation name and the ordered list of variables which was returned by Compat. This parallels what occurs at the top level in the Sentence rules. What occurs next is also a process that occurs within the top-level procedure, Make-gry. That is, the information which has been derived so far is integrated into a complex predication involving the constructed relation, the quantifiers and intensions of the adverbial phrase, and the noun phrase which precedes it. This complex predication is composed in two stages, by the Partial-intension procedure and the Make-pred procedure.

Composing a Sub-Query

The Partial-intension procedure has the job of embedding the previously constructed relation, which we will call Rel, in a "quantified" expression involving the quantifier and intension of the object phrase (which we represent by Var2).
Partial-intension also checks a negation flag to see whether it should negate the quantified predicate which it returns. If no negation is present, Partial-intension returns a predicate of the form

"apply-oper(Var1, Var2, Rel, Intension2, Operator2)"

which may be roughly translated as, "Var1 satisfies the relation Rel for every value of Var2 which is required to satisfy Intension2 by Operator2." For example, if Operator2 is 'the', and Var1 is bound to 'Frege', then the predication may be read as, "Frege satisfies Rel for the value of Var2 which satisfies Intension2."

If a negation flag is set, however, or if Operator2 is 'no', then a different action is taken. When Operator2 is 'no' (or should be construed as 'no' because of the context), then a predicate such as the one shown above is returned, but with a special negation predicate prefixed. If the negation flag is set, but Operator2 is not 'no', then a check is made to see if Operator2 is 'some' (as in "some X or other"). If the check succeeds, then Operator2 is replaced by 'every' and a negation operator is prefixed to

"apply-oper(Var1, Var2, Rel, Intension2, every)"

because the intended interpretation is that the relation in question does not hold for every Var2 that satisfies Intension2. However, if the negation flag is set, and Operator2 is not 'some', then the negation is performed at a later stage.
Once a Predication has been returned by the Partial-intension procedure, it is passed to the Make-pred procedure, along with the variable (Var1) representing the higher level noun phrase which the relative clause is modifying, and along with the intension, variable, and operator of whatever adverbial phrase may be present.

Make-pred will first test whether the adverbial intension (Intension3) is nil. If so, it will either return the Predication it has just been passed, or its negation, in the case where negation is present and no previous negation action has been taken. Otherwise, this Predication is embedded in a more complex Predication, of the form

"apply-oper(Var1, Var3, Predication, Intension3, Oper3)"

which may be read as, "Var1 satisfies Predication for those values of Var3 which are required to satisfy the adverbial intension by the adverbial operator." Thus, when an adverbial phrase is present, a second order predication is constructed. For example, given the question "Does Mary teach a course in every semester of 83", the following second order predication would be composed:

apply-oper(Mary, S, P, 5083/season(S)/nil, every),

where P is bound to

apply-oper(Mary, X, teaches(Mary, X, S), course(X)/nil, some).

The occurrences of S, at every level, have the same meaning.

Any second orderPredication which is constructed is returned as the value of the Make-pred procedure, unless an
unprocessed negation flag is present. If the negation flag is set and no previous action has been taken on the negation, then a negation operator is prefixed before returning the second order predication. In any case, whatever is returned by the Hare-pred procedure becomes the intension of the entire relative clause and the Relative-clause procedure terminates.

At this point we are back in the NP procedure, and whatever intension has been returned from the relative clause procedure is prefixed to the list of restrictions which were obtained from processing whatever adjectives were present in the noun phrase. If either no adjectives were present or no relative clause was found, then nil is prefixed, and no harm is done. The list which is obtained from appending the relative-clause restrictions to the adjective restrictions is next prefixed to the predicate which represents the intension of the main noun (e.g., 'course(X)' or 'prof(X)').

The result of all this appending will be returned as the intension of the noun phrase as a whole. The order in which these predicates are appended together was chosen to optimize, in the vast majority of cases, the search for values which satisfy this intension, as we shall see in Chapter 6.

Checking Quantifier Ambiguity

Before the NP procedure returns a value for the quantifier or "operator" which has been found to quantify the main noun, it
calls upon a procedure which examines the quantifier to see whether ambiguities are present. If the quantifier is 'some', or has been found to be equivalent to 'some', then an ambiguity may exist. For example, in "Does every math professor teach some course", the occurrence of 'some course' is ambiguous. In such cases, the procedure, Case-some, questions the user about the intended interpretation. If no negation precedes the noun phrase, as in our example, then the user is asked whether the intended interpretation is 'some particular course' or just 'some course or other'. Depending upon the user's reply, the procedure will either change the quantifier to 'some-particular', or leave it as 'some'. If a negation did precede the noun phrase, however, then a third possible interpretation is posed to the user. For example, if the original question was "Is it true that every professor who teaches some course does not teach a course this spring", the phrase 'a course' which follows 'not teach' could mean 'a particular course', 'some course or other', or 'no course at all'. The user is asked to choose between these interpretations. If the third interpretation is chosen, the original quantifier, which was interpreted as 'some', is reinterpreted as 'no'. It may frequently happen that the context of a question will rule out certain interpretations, and that the program will find ambiguities where none would occur to a human listener. The way in which context constrains a human's interpretation of quantified sentences is very complex, however, and we defer
discussion of this problem until chapter 6.

Once the NP procedure has attempted to remove ambiguities in the determiner of the main noun of a noun phrase, it has completed its work, and control returns to a higher level. Depending upon the determiner which was finally chosen for a noun phrase, SHADOW will take different actions when the final query is evaluated. We postpone discussion of these actions until later, however.

Noun Phrases with Prepositional Modifiers

We have just seen how SHADOW processes noun phrases of the form

"determiner + adjs* + noun + relative-clause*".

The program also recognises noun phrases which contain prepositional modifiers, e.g., noun phrases of the form

"determiner + adjs* + noun + preposition + noun-phrase*".

Prepositional noun phrases are processed by the program in two different ways. The first approach is to see whether the main noun, which is modified by a prepositional phrase, implies a relation which would connect the main noun with the noun phrase which follows the preposition. For example, in "the prerequisites for math 303" the main noun, 'prerequisites', implies something like "courses which are required by", and in "every member of computing", 'member' implies something like "professor who belongs to." The second approach is to try to
discover what relation holds between the nouns related by the preposition on the basis of the semantic types of those nouns. The second approach is adopted only if the first approach fails. We will briefly consider the two approaches in order.

On the first approach, when a preposition is found to follow an expression of the form, "determiner + adjs* + noun", the dictionary is searched to see whether the main noun phrase implies a phrase which can be used to construct a relative clause. Examples of such phrases are given in the previous paragraph. If such a phrase is implied, the implied phrase is prefixed to the noun phrase which follows the preposition, and the resulting expression is passed to a routine called Special-rel-clause, which performs the same operations as the relative clause procedure which we have already seen. In addition, Special-rel-clause calls upon a procedure, Set-oper, which inspects both the quantifier of the noun which the relative clause is modifying, and the quantifier of the noun in the embedded clause. When certain combinations of quantifiers are discovered, one or both of the quantifiers may be changed, in order to capture the true sense of the expression. For example, if the expression "every member of every department" were being processed, the pair of quantifiers (every every) would be examined, and the pair (every some) would be returned. This is because, if the program were to try to find every professor who is a member of every department, it would find that there are no such professors. But this is not what the user
is trying to find out. Generally speaking, the intended interpretation of "every member of every department" is "every person who belongs to some department or other", and this is the interpretation which is constructed. On the other hand, the Set-oper procedure leaves (the some) unchanged, so that "the member of some department" will be interpreted as "the professor who belongs to some department."

The Set-oper procedure relies upon heuristics to a greater extent than most other parts of the program, because generalizations about how combinations of quantifiers are to be interpreted are subject to exceptions. For example, in a certain context "every teacher of every student" might mean "those teachers who have taught every student" (in a primary school, say).

There is one combination of quantifiers which the Set-oper procedure identifies as being too complex for the present state of the system, namely, "some x of every y", as in, "Is some prerequisite of every computing course a math course?" When this kind of construction is discovered in a question, the user is informed that the system cannot presently answer such questions. The reasons for this are discussed in the next chapter.

Once Set-oper has returned an appropriate pair of quantifiers, (in many cases the same pair it was given), the Special-rel-clause procedure proceeds in a fashion analogous to the normal relative clause procedure, and it returns an
intension to the NP procedure which was processing a prepositional phrase. This NP procedure then constructs an intension for the entire noun phrase just as if it had processed an ordinary relative clause.

If the first approach to prepositional phrases fails, because the principal noun in the phrase does not imply a relative-clause type of phrase, then the second approach is adopted, and the system will try to find an appropriate interpretation for the preposition present by inspecting the types of the nouns involved. For example, suppose our phrase is "every course in mathematics". During the parsing of this phrase by an NP procedure, the preposition 'in' is discovered, and an attempt is made to see whether 'course' implies a missing phrase. When this attempt fails, another NP rule will be tried which, when it discovers the preposition, will pass the types of the two surrounding noun phrases to a procedure, Find-Rel, which attempts to find a relation involving the two types ('course' and 'dept' in our example). If an appropriate relation can be found, then the relation name, together with the list of all types required by the relation, are passed to a procedure (Transform) which inspects the variables which correspond to the two types ('course' and 'dept'), and which embeds these variables in a list of variables which should be used when constructing a relational predicate involving the relation name which has been found. For example, suppose that the variable X represents the noun 'course' and that Y represents 'mathematics
(department). When it is discovered that the relation 'offers' involves the types (dept, course, semester), the Transform procedure will create the list \((Y, I, Z)\) as a list of variables to be used when constructing the relation, \("offers(Y, I, Z)\". The Transform procedure is capable of filling in missing variables, \("Z" in this case\), and of reordering variables, since prepositions do not imply what the voice of the relation should be. Once a viable relation has been found to relate the different noun phrases surrounding a preposition, the entire prepositional modifying phrase is handled as though it were a relative clause.

The entire procedure just described is clearly a heuristic one, and it is based upon the assumption that we only use prepositional modifiers in cases where the missing relation can be inferred either from one of the noun phrases present, as in "prerequisite of", or where the missing relation can be inferred from both noun phrases being connected by the preposition. This assumption is not universally true, \(^1\) but it has been remarkably successful in dealing with a wide variety of prepositional phrases. It should be emphasized that the second method of "inferring the relation" is only used in prepositional phrases, and only when the first method fails and the phrase cannot be

\(^1\)In some cases the relationship between the related noun phrases may be inferred from the preposition itself, e.g., 'above', 'below', 'from'. The SHADOW system does not presently contain relations where such prepositions are appropriate. However, the system could easily be expanded to accommodate such prepositions and to extract the implied relation directly from the preposition.
interpreted as an adverbial phrase.

**Parsing "Dummy" Noun Phrases**

There remains one other kind of noun phrase which the program recognises, and that is the "missing" noun phrase, whose existence is postulated in order to allow a sentence to be successfully parsed. For example, the program infers the existence of an agent in "What courses are offered this semester?"

When the program parses a "missing" noun phrase it consumes no input, and returns a 'dummy' semantic type, a variable, and a quantifier which is determined from the context. The trivial intension, "true", is also returned, but this intension is replaced by a non-trivial intension when the "dummy" noun phrase is the subject of a negated verb phrase. We will discuss how this is done in the section entitled "Composing a Top Level Query".

**Parsing Verb Groups**

We have now seen the various kinds of noun phrases which the SHADOW system can analyse. Let us briefly consider the way in which the system recognises verb groups. The program contains a procedure, Fancy-verb, which is capable of recognising both active and passive verb forms, and verbs such as 'belongs to'
and 'is not' (as in 'A is not B') which are followed by particles. The procedure also recognises verb formations in which the negative, 'not', appears where auxiliary verbs are likely to be found, as in, "will not be teaching". Particles which follow a verb are routinely consumed by the Fancy-verb procedure. Everything else which this procedure performs is accomplished by calling upon the Verb procedure, which has different Prolog conditionals for recognising active and passive forms. The Verb procedure calls upon the Aux procedure to consume auxiliary verbs and negative particles. Whenever an auxiliary verb is consumed its tense is noted, and whenever a negative particle is consumed a negation variable is set to 'not'. Thus, the Aux procedure returns information about tense and negation status. The Verb procedure also calls upon the Get-Verb procedure to check whether particular words are verbs, and to fetch their associated relation name and tense. This information is then passed upwards, through Fancy-verb, to the level of major or subordinate clause processing. We have seen how the relation name and negation flag which are returned by Fancy-verb are useful, but what use is made of the tense of the verb? To answer this we must examine how adverbial phrases are processed by the system.
The program recognises two different classes of adverbal phrases. The first class has the general pattern "prep + deter + orderword + noun", where 'prep' may be absent, or be 'in' or 'during'. Likewise, 'deter' may be absent, or may be 'the' or 'this'. The 'orderword' may also be absent, or be a word which determines an ordering, such as 'next', 'last', or 'this'. The noun present must be a temporal noun such as 'spring', 'year', or 'semester'. Examples of adverbal phrases which fit the first pattern are: "next fall", "this coming semester", "during the previous year", "in the spring", and the like. Adverbal phrases which fall into the second general class are those which involve quantifiers such as 'every', 'some', etc. But, before we consider the second pattern of adverbal phrases, let us see how examples of the first pattern are analysed.

First we should observe that all adverbal phrases which the program recognises are temporal phrases having to do with the seasons, semesters, and years. In this respect the program is domain specific, for it has not attempted to deal with a wide range of adverbials. Nevertheless, an attempt has been made to process the kinds of adverbials which the program does recognise in a general way.

The most important feature to note about the first class of adverbial phrases is that they either contain or imply a temporal ordering operator. Thus an attempt is always made to
discover a word such as 'coming', 'previous', or 'this'. If such a word cannot be found, then the determiner, 'the', must be present, as in 'during the spring'. This determiner, by itself, does not imply an ordering operator, but when the tense of the verb group is consulted, an order operator can be inferred. There is a procedure named 'Time-func' which extracts an order operator in cases like this. In such cases the operator returned will always be, 'last', 'this', or 'next'.

Once some kind of ordering operator has been discovered, and once a temporal noun has been parsed, a procedure, Det-time, is invoked which inspects the type of the temporal noun, to see whether nouns of that type are ordered sequentially (as years are) or are ordered in an indexed cycle. For example, the seasons are ordered in the cycle "spring, summer, fall", but are also indexed by years, e.g., "fall 83". In cases where the type of a noun indicates that it is ordered sequentially, a procedure is invoked which finds what the current value for nouns of that type is, e.g., what the current year is, and depending upon what the order operator is, either increments, decrements, or leaves the value unchanged. This procedure has been designed to be as general as possible, except that it returns a value of the form "season\&year", where the value of 'year' is a constant which was determined by the procedure, and the value of 'season' is left undetermined. The fact that a value of this form is returned by the procedure reflects the fact that relations in the database contain fields pertaining to seasons of the year, and that the
values of these fields have the form, S&Y.

In cases where the type of a noun indicates that it is ordered by cycles but indexed by a sequential parameter, a procedure, Indexed-cycle, is invoked which takes the noun, e.g., 'fall', and the order operator, e.g., 'next', and it looks up the current season and year. It then finds the position of the given noun (fall) in the standard cycle for nouns of that type, and decides on the basis of this position and the order operator (next), whether the sequential parameter (year, in our case) should be incremented, decremented, or left the same. For example, when the given noun is 'fall', and the order operator is 'next', and the current semester is 'fall', it is clear that 'next fall' must be in the next year and not this one. For this example, the Indexed-cycle procedure returns the value, 'fall1984'.

Once a value of the form 'S&Y' has been obtained, the processing of the first class of adverbial phrases is complete. The 'S&Y' value is bound to the variable which represents the denotation of the adverbial phrase, and 'nil' is returned as the intension of the phrase. This is because the intensional information has all been embedded in the 'S&Y' value which is eventually used when constructing relations such as "offers(math,math-303,S&Y)". When adverbial phrases which fall into the second class are analysed, however, a non-empty intension is returned. Let us turn to this second type of adverbial.
The class of adverbial phrases we now consider have the general form: prep + quantifier + noun + (possible repetitions of this pattern). Examples of phrases which have this form are: "during every semester", "in some fall", "during every semester of some year", "in the fall of 83", "in the year of 83 in the fall", and so on. In general, these phrases have a form which is recognizable as a normal noun phrase by the procedures we have already discussed, provided we first remove the leading preposition which is required to be 'during' or 'in'. Once again, the nouns which appear in these expressions are required to have temporal semantic markers.

Let us first consider how simple phrases, such as "during every semester" or "in some year" are processed. The leading preposition of such phrases is first removed, and the remainder is parsed as a simple noun phrase. The NP procedure will return a variable (call it Y) to represent the main noun, a quantifier, and an intension which restricts the range of the variable (such as year(Y)). A type, such as 'yr' is also returned. However, we want our 'Adverbial' procedure to return more than this. We also want to return a value of the form $SEY$ which can be used in the manner already described. To accomplish this, the Adverbial procedure constructs a predicate of the form $SEY$ by embedding the variable already obtained ('Y' in our case) in the position indicated by the semantic type which was returned (season or year). The remaining variable in the predication ($S\&Y$ is a syntactic variant of the predicate '$S(S,Y)$') is either left
unbound, or is quantified in an appropriate way. For example, the expression 'every semester' is interpreted to mean 'every semester of every year'.

The information which is returned by the Adverbial procedure will now be something like "(SEY, year(Y), yr, Y, every )", which would indicate that we are concerned with every Y such that Y is a year and Y satisfies the predication, SEY. In the database of the program there is a list of "viable years", i.e., the years that the database is concerned with. For Y to be a year it suffices for Y to be a viable year, and for Y to satisfy SEY it suffices for Y to be a year and for S to be a season.

When more complex adverbial phrases, such as "during every semester of some year", are analysed by the program, the phrase "every semester of some year" is analysed as a noun phrase which has a prepositional modifier. The relation between 'semester' and 'year' is discovered by looking at the possible relations into which their types enter. There is only one predicate in the database which relates the types 'seas' and 'yr', and that is 'S'. Consequently, in the process of analysing "every semester of some year", the NP procedure would automatically construct a predicate of the form SEY, and would return the information that we are looking for every S such that S satisfies SEY, and S is a season, and Y is a year. The predicate, SEY, is extracted from the complex intension in which it is embedded and is returned as the value which should be embedded in top level relations.
However, the remaining intensional information concerning the restrictions on the variable $S$ is also returned, since we need to quantify over $S$. The restrictions on $Y$ are embedded in this intensional information about $S$. The program contains a procedure named 'Gather' which performs some rather subtle analysis on the kinds of intensions we have been discussing, but we will pass over these complexities here.

Now that we have a better picture of the kinds of values which are returned from the analysis of adverbial phrases, we are in a better position to understand how this information is used when constructing a final query. If the main verb in a given clause corresponds to a two-place relation, and has no place for a temporal argument (such as "member(prof, dept)"), then, if the user's question is to be successfully interpreted, there must be no adverbial phrase present in the query, and invoking the Adverbial procedure merely returns a set of 'nils', which are ignored. If the main verb is a three-place relation, with a position for temporal information, it is still possible that no adverbial phrase is present in the query. In this case an unbound variable is supplied as the temporal argument to the relation, and this variable can be unified with any constant. In the case where the main verb implies a three-place relation, and temporal information is found by the Adverbial procedure, then an argument of the form, $S$,$G$,$Y$, is supplied to the relation.

There are two sub-cases for this last case. In the first sub-case all the information present in the adverbial phrase has
been represented in the S5Y expression (as in "next year") and a nil intension is returned for the adverbial phrase. In the second sub-case, both an expression of the form, S5Y, and an intension and quantifier are returned, as is the case with "every semester of 83".

**Composing a Top-level Query**

When we discussed the Relative-clause procedure, we saw how the information which is returned from both an adverbial phrase and an object phrase can be formed into a predication which is "about" the variable which represents the subject phrase. This is done by the Partial-intension and Make-predicate procedures. The result obtained by invoking these two procedures in succession is a predicate of the form:

"apply-oper (Var1, Var2, Relation, Intens2, Oper2)"

where Var1 represents the subject which the predication is about, and where Relation may itself be a predicate of the above form, in the case where complex adverbial information forced a second-order predication to be formed.

We now need to consider in somewhat greater detail how a top-level query is formed, using the "operator" and intension of the surface subject. We have briefly discussed the Make-query procedure, which performs this task, but a few aspects of this procedure need clarification. As a general rule, the behaviour of this procedure is very straightforward. It calls upon the two
procedures mentioned above to integrate the information from the object and adverbial phrase into a complex predication which is about the surface subject of the sentence. Let us call this predication "P(X)", where X represents the surface subject. Next a query is constructed having the form:
apply( Operator1, X, Intension1, Y, P(X) ).

Suppose that Operator1 is "find-all". Then this query can be roughly interpreted as: "apply the operator 'find-all' to those Xs which both satisfy Intension1 and P(X)". Later we will discuss how the 'apply' predicate is defined. The question we need to consider at this point is: what happens in those cases where Intension1 was introduced through parsing an implied noun phrase, as happens with "who is teaching this semester?". When an implied noun phrase is parsed, the trivially satisfied intension, 'true/nil', is returned. Since no variables occur in this expression, it cannot be used to bind constants to any variables, and thus, cannot be used to find Xs that satisfy Intension1. In cases where the surface verb phrase contains no negation, this causes no problems, since the predicate, P(X), can be used to instantiate X. However, in cases where a negation is present in the verb phrase, the predicate P(X) will contain an embedded negation operator, and cannot be used to find ground instances of X. This is because, in Prolog, it is not possible to find values for X which fail to have a property, Q, by evaluating a predicate of the form, not(Q(X)). What happens when we try to evaluate a predicate of this form is that Prolog
first tries to satisfy the goal, \( Q(x) \). If there are any individuals which have the property \( Q \), then the goal succeeds and the negated goal fails. On the other hand, if no individuals have the property \( Q \), then \( Q(x) \) fails, the negated goal succeeds, but \( x \) is left unbound. In any case, we have not found those \( x \) which fail to have property \( Q \), and the problem remains, how do we find those individuals? An excellent discussion of many aspects of this problem is presented in [Dahl (1980)].

One point which Dahl makes is that whenever we have both positive and negative predications which a variable \( x \) is required to satisfy, we can always rearrange the order in which the predications are evaluated, so that we evaluate positive predications involving \( x \) before evaluating the negative one(s). In this way the variable \( x \) is always grounded before the negative goal is posed. This technique is used whenever possible in the SHADOW program. In particular, it is always used when attempting to find a value for \( x \) which satisfies the intension of a noun phrase which contains a negative relative clause. For in every such noun phrase there is a positive noun which the relative clause is modifying. For example, in "professors who do not teach this semester", we have the positive information that \( x \) must be a professor. As a general rule, the program attempts to ground the intension of a relative clause before evaluating the intension of the noun which heads that clause, because, usually, the relative clause is more restrictive than the head noun. However, whenever a negation is embedded in the relative clause.
the opposite course is followed.

We have seen, however, that this tactic cannot easily be applied in those cases where negation occurs at the top level of a sentence, and the subject of the sentence is only implied. In order to deal with such cases, the Make-query procedure has been designed to construct appropriate positive intentions for those variables which represent "dummy" subjects. As a general rule, the main verb of a sentence will require its subject to be of a certain semantic type, and we have noted that this type is retrieved in cases where the surface subject of a sentence is a dummy. Thus, in the sentence "who is not teaching this semester" the type 'prof' is known to be appropriate for the subject of the sentence. It is also assumed that when questions such as this are asked, what the speaker wants to know is "Which of those individuals who are capable of teaching are not teaching...?" Consequently, the Make-query procedure constructs a predicate using the semantic type which the subject of the query is required to be. In the present example, the predicate 'prof(X)' is constructed and assigned as the intension of the subject phrase. Of course, this strategy requires the designer of the system's lexicon to provide rules for finding individuals which satisfy any semantic type known to the system. This is a valid requirement, however, since it is not plausible that we have semantic types attached to our concepts unless we know how to find individuals of those types.
what happens in cases where the main verb of a sentence does not impose type restrictions? For example, the 'is' of predication in, "What is not a department?", does not require the subject to be of a particular semantic type. In such cases the program tries to see whether the interrogative pronoun imposes any type restrictions. If the pronoun is 'who', then the type 'human' is assumed, and the intension of the surface subject is inferred to be 'human(X)'. It happens that the only humans listed in the present database are professors, but if students were also listed, then we would establish that X is human by showing that X is a student or a professor. If the interrogative pronoun is not 'who' but 'what', then the intension assigned to the surface subject is 'entity(X)', where X is an entity just in case X satisfies any semantic type of any noun in the lexicon. In this way we can guarantee that the variable associated with the subject of any query can always be grounded before a negative predicate is evaluated. The program always tries to construct the most restrictive intension which can be inferred from the context, but if no restriction can be inferred, then the universally satisfiable intension 'entity(X)' is always available.

Returning once again to the Make-query procedure, once this procedure has made sure that the surface subject has a positive intension, it proceeds to construct a query in the fashion we have already outlined. The query which is constructed will be evaluated by a procedure named 'Eval', but before the query is
evaluated a check is made to see whether there are any global flags indicating that some quantifier was interpreted as 'some particular'. The program takes care to instantiate all variables which are quantified by 'some-particular' before it evaluates any other predicates. This is to ensure that if some complicated relationship is supposed to hold for some particular $x$, that the particular value of $x$ does not change during the process of evaluating that relationship. For example, if we are evaluating the query "Is it true that every professor who teaches some particular course is a math professor", we want to make sure that the particular course does not vary with the particular professor. This could be assured by restructuring the query as "Verify that for some course, every professor who teaches that course is a math professor". However, restructuring the query is awkward, and can become very complicated if the quantifiers we need to reorder are separated by more than one level of nesting. The SHADOW system adopts the strategy of reordering quantifiers only when 'some' applies to the surface subject of the query. In all other cases a different strategy is adopted. The program contains a procedure (Instantiate) which has the task of grounding every variable, at every level of nesting, which is quantified as "some particular". With the exception of when "some particular" applies to the surface subject, this quantifier or operator always appears in an expression of the form:

Global flags are created by adding special assertions to the database which are removed when appropriate.
apply-oper(\(X, Y, \text{Relation}, \text{Y-intension}, \text{some-particular}\))

which can be read as, "\(X\) satisfies \text{Relation for some particular} \(Y\) which satisfies \text{Y-intension.}" The \text{Instantiate} procedure performs a depth first search on each query for which \(X\) is invoked, and every time it encounters an expression of the form shown, it instantiates \(Y\) to a particular value, the most deeply embedded \(Y\)s being the first instantiated. Once the entire query has been instantiated in this fashion, the query is evaluated. If the query is successfully evaluated, an answer is printed, a new question is fetched, and the interpretation process begins again. If the query fails, however, either because it cannot be proved or because a false presupposition is discovered, then Prolog backtracks to the Instantiate procedure and if an alternative method of instantiating a particular \(Y\) exists, then that path will be tried. Since this backtracking occurs in a depth first fashion, every possible way of selecting particular values for the various \(Y\)s will be tried if need be.

\textit{Evaluating a Top-level Query}

We are nearly in a position to examine how a particular formal query is evaluated. Before doing so, however, it will be useful to understand, abstractly, how the "Truth-cond" procedure works, which is capable of finding ground values for a variable which occurs in a list of predicates which we have been loosely calling intensions. Suppose our list of predicates is
"P(X)/\neg(X)/\text{nil}". (The list \((a \ b \ c)\) is written \(a/b/c/\text{nil}\).) The Truth-cond procedure will try to find an \(X\) which satisfies this list of predicates in roughly the following way. The list will first be reordered so that any predicates which contain embedded negations are moved to the end of the list. Next, a check is made to see whether this particular list of predicates has been evaluated before, and if so, whether all possible values which satisfy the list have already been found. If all possible values have NOT been found, then the "Satisfies" procedure is called upon to find a value for \(X\) which satisfies each predicate in the list. If a value is found, then an assertion is added to the database saying that that particular value satisfies the list, and the value is returned as a value for \(X\). The Truth-cond procedure can be forced to find all such values for \(X\) by forced failure and backtracking. If it ever again becomes necessary to find all values for \(X\) which satisfy a given intension, it will not be necessary to derive these results a second time. Once all such values have been found, a global flag is asserted which records this fact, and in the future the Truth-cond procedure finds values which satisfy that intension by consulting the database directly. This tactic of saving derived information can greatly improve the speed for evaluating queries in which universal quantifiers appear. Derived information is not stored from one user query to the next, but the program could easily be modified to enable this if it seems desirable.
Before leaving this discussion of the Truth-cond procedure, we will very briefly consider how the Satisfies procedure works. This procedure is invoked by posing the goal, Satisfies(\(X, \text{Intensional-list}\)). The procedure simply works its way through the top level of the intensional list, making sure that the first predicate is satisfied before recursively invoking itself with the remainder of the list. As soon as the first predicate in the list is satisfied (through being posed as a goal), the value of \(X\) is bound, and all remaining predicates in the list must hold for this \(X\). The only complication arises when predications of the form, \(\text{Bind(type, adj)}\), are found in the list. Such predications are converted to predicates of the form \(\text{Adj(adj, type, } X)\), so that they will apply to the variable \(X\). These predicates can then be evaluated in the manner described in the discussion of adjective parsing.

We are now in a position to understand how any formal query which the system has constructed is evaluated. We can gain this understanding by examining how the 'Apply-oper' and 'Apply' predicates are evaluated, since these are the primary predicates involved in evaluating a query. Let us first consider the behaviour of the Apply-oper predicate (which is also a procedure).

The Apply-Oper Procedure
The Apply-oper predicate has the general form,

"Apply-oper(X, Y, Relation, Y-intension, Y-operator)"

where the Y-operator may be 'some', 'some particular', 'the', or 'every'. We will consider the behaviour of this procedure for each of these operators (quantifiers) in turn. When Apply-oper is invoked with the Y-operator equal to 'some', the procedure will try to verify that the value of X satisfies the value of Relation for some Y which satisfies Y-intension. If X is already bound, then Relation is first posed as a goal, to find a Y which has the proper relation to X. A test is then made to verify that this Y also satisfies Y-intension. If failure results, Prolog will back up to see if another value for Y can be found which satisfies the Relation, and the cycle repeats. If the variable X is originally unbound, then evaluation occurs in the reverse order. First an attempt is made to find a Y which satisfies Y-intension, (using the Truth-cond procedure) and if one can be found, the Relation is next posed as a goal. Again, backtracking occurs if necessary. If both goals succeed, X will be returned with a bound value. It is important to remember that if the Relation which is being tested contains a negation at any level of nesting which involves X, then X will be bound at the time Apply-oper is invoked. Both the Apply and Truth-cond procedures have been designed to ensure this.

The case where Apply-oper is invoked with the Y-operator equal to 'some-particular' is much simpler. For in this case the Y variable will already have been given a value when the
Instantiate procedure was invoked. All that remains is to test that \( x \) does satisfy the given relation for the particular \( Y \) which is present. If the test is satisfied, then the Apply-oper procedure succeeds. Otherwise, everything backs up to the Instantiate procedure to see whether another particular \( Y \) can be found which satisfies the \( Y \)-intension.

The case is more complicated when Apply-oper is invoked with \((X, Y, \text{Relation, } Y\text{-intension, the})\) as arguments, for in this case the procedure must verify that \( x \) satisfies the given Relation for the unique \( Y \) which satisfies the \( Y \)-intension. To assist this verification, a procedure, Unique, is invoked to insure that one and only one \( Y \) satisfies the \( Y \)-intension. The Unique procedure is somewhat complicated, because, in order to test whether more than one value satisfies the \( Y \)-intension, it must first find one such value, save that value by adding an assertion to the database, and then force a failure to cause another value for \( Y \) to be searched for. This forced failure means that the first conditional defining Unique must fail. Consequently, flags which are tagged with "level numbers" must be asserted (and later removed) to enable the remaining conditionals which define Unique to tell what happened in the first conditional. The "level numbers" are required, since one call to Unique may be nested inside another call to Unique. It is not necessary for us to unravel all the subtleties of how these flags operate. The important point here is just that the flags can be used to tell whether zero, one, or more than one
value was found which satisfies the \( Y \)-intension.

If no such value was found, a message is stored to the effect that there is a false presupposition concerning the existence of a value which satisfies the phrase from which the given intension was derived. The program has saved this phrase and can retrieve it when needed. If, on the other hand, exactly one value was found, then all is well and the call to Unique succeeds. If more than one value was found, however, then a different false presupposition message is stored. The Unique procedure is designed so that it never tries to find more than two values satisfying the relevant intension, since, as soon as two values are known, uniqueness is disproved. When two values are found, they are stored, along with an explanatory message, and a 'deviant' flag is asserted. All messages which report false presuppositions are saved until all possible interpretations of the original query are tested, for it may happen that the false presupposition existed only on a given interpretation. When new interpretations are found, all 'deviant' flags are removed, but all deviant "messages" remain stored on a push down stack. If all interpretations fail, then the most recent message on the message stack is printed out, the deviant phrase in the original query is identified, and then all messages are erased.

Returning now to the Apply-oper procedure, once a successful call to Unique has been completed, all that remains is to test that the unique \( Y \) which has been found does satisfy
the given relation for the given value of X. If X was previously
unbound, then X will be bound by posing the Relation as a goal.

The case where Apply-oper is invoked with (X, Y, Relation,
Y-intension, every) as arguments is complicated in ways which
are similar to the complexities of the Unique procedure. If X is
bound to a constant at the time of invocation, then a procedure,
Test-every, is invoked which tests that X satisfies Relation for
every Y which satisfies the Y-intension. The Test-every
procedure works roughly as follows: first an attempt is made to
find a Y which satisfies Y-intension. If one is found a flag is
asserted and the given Relation is tested to show that both X
and Y satisfy the Relation. (The Relation may of course contain
other variables). If this test is passed, the original flag is
deleted, and failure is forced. The forced failure causes
backtracking to the point where another value for Y is sought,
and the whole process repeats. The procedure is designed so that
backtracking does not cause the given Relation to be retested
until a new Y is found. Also, if a value for Y is found which
fails to satisfy the Relation, failure results and no more Ys
are sought.

After a series of values for Y have been found, the
Test-every procedure examines the state of its flags (which are
tagged with level numbers to make them local), and if any flag
is left in place, then we know that some Y was found which did
not satisfy the Relation, and Test-every fails. If no values for
Y were ever found, then a "false presupposition" message is
Returning now to the case where Apply-oper was invoked with 
(I, Y, Relation, Y-intension, every), we have just examined what 
happens when X is already bound. If X is initially unbound 
(which doesn't happen when the Relation contains a negation), 
then an attempt is made to ground X before invoking the 
Test-every procedure. This can only be done by grounding the 
given relation, but efficiency requires that we first ground Y, 
so that an X can be found which has a chance of being related to 
all Ys. We do not want to ground the Y-intension, however, since 
we must eventually find all values of Y which satisfy this 
intension. Therefore, a copy is made of the Y-intension in which 
the Y variable is renamed. Likewise, a similar copy is made of 
the original Relation. This makes it possible to obtain ground 
values for Y without grounding variables that we don't want to 
ground. Also, a copy of the original Relation is made in which 
every variable but X is renamed. This makes it possible to 
ground X without grounding any other relevant variables. The 
result is that we end up with a ground value of X and copies of 
Y and Y-intension which can be passed to Test-every to verify 
that X satisfies the original Relation for every Y1 which 
satisfies Y1-intension. This completes our discussion of the 
Apply-oper procedure.
The Apply Procedure

We turn now to the Apply procedure, which has the general form

Apply(X-oper, X, X-intension, Y, Predication),

where X represents the subject of the query and Predication is an Apply-oper predicate involving X, Y, and possibly other embedded variables. The Apply predicate is the top level predicate which is formed to represent the meaning of an entire query. Consequently, the dominant operator, X-oper, will either be one of 'find-the', 'find-all', or 'count-all', or in case the question-phrase was equivalent to "is it true that", the X-oper will either be 'the', 'some', or 'every'. In the latter case the Apply procedure behaves in a fashion very similar to, and indeed calls upon, the Apply-oper procedures. In addition, every form of the Apply procedure is responsible for printing out an answer, unless a false presupposition is uncovered, in which case a message is printed by the Ans procedure.

Let us consider the case where Apply is invoked with the arguments (find-the, X, X-intension, Y, Predication).

Essentially what happens in this case is that the X-intension is prefixed to the Predication, the result is called Hold, and then Apply-oper(Y, X, true, Hold, the) is invoked. This will test whether a unique X satisfies Hold, and whether that X satisfies 'true', which it does, of course. If a unique X is found its value is printed. Otherwise, a deviant message has been stored.
In cases where the $X$-operator in "Apply($X$-operator, $X$, $X$-intension, $Y$, Predication)" is 'the' or 'some', the corresponding Apply-oper procedures are also invoked, except that 'Predication' plays the role of 'Relation', and $X$ and $Y$ exchange roles. For example, if we want to Apply the operator 'some', we use Apply-oper to verify that $Y$ satisfies the Predication for some $X$ which satisfies $X$-intension. $Y$ is unbound at this point, of course, so $X$ will be grounded before the Predication is invoked, which in turn grounds $Y$. If everything succeeds, a "Yes" message will be printed. Otherwise, a "no" message is stored for later printing in just the case where no successful interpretation of the query can be found.

The case is very similar when the $X$-operator is 'every', except that we do not call upon the Apply-oper procedure. Rather we bypass that procedure and directly call upon the Test-every procedure to verify that every $X$ which satisfies $X$-intension also satisfies the Predication.

When the Apply procedure is invoked with 'find-all' or 'count-all' as the $X$-operator, the Apply-oper procedure cannot be directly used to answer the query, since these operators do not occur except in the surface subject. The operators correspond to question phrases like "Who are the" and "How many".

When the Apply procedure is told to "find-all" it repeatedly finds individuals which satisfy both the subject intension and the main Predication. These individuals are
non-redundantly stacked up, and then printed out while the stack is being destroyed. An appropriate message is printed if no individuals could be found to satisfy the given conditions.

A similar procedure is followed when 'count-all' is the dominant operator, except that the individuals in the non-redundant stack are counted as the stack is destroyed and the total is printed out. The individuals are stacked before counting to prevent duplicate values from being counted (they are not stacked).

This completes our discussion of the major algorithms and heuristics employed in the SHADOW system. Of necessity, many of the subtleties of the system have been passed over. It is hoped, however, that many of the subtleties have been exposed without obscuring the high level strategies. More details on SHADOW's behaviour can be found in Appendix C.
VI. CONCLUSION

In this chapter we discuss certain difficulties which are introduced by the SWADON system, but not resolved. We also discuss ways in which the system could be expanded and improved. Finally, we summarize the most salient features of the system.

Problem Areas

In this thesis a serious effort has been made to disambiguate queries which are ambiguous by virtue of the order of quantifiers present in the query. Although the system does succeed in disambiguating many classically ambiguous queries which fall into this class, it does not succeed for all such queries. For example, consider the queries,

(A) Who does not teach a course in the fall of every year?
(B) Who teaches no courses in every semester of 84?

When query (A) is posed to the system the user will be asked to disambiguate the phrase "a course in the fall of every year". Let us suppose that the user chooses the interpretation "no courses at all in the fall of every year?" Even after the user has chosen among three possible interpretations for the phrase 'a course', there still remain at least two possible interpretations of the query. The user could be asking, "Who fails to teach in any fall semester at all?" or "Who fails to
teach in some fall semester?” The ambiguity arises from the fact that the negation operator could either apply to 'teaching' or to 'every'. Likewise, question (B) has a similar ambiguity.

At present, the program will find only one interpretation for each of these questions, once the phrase 'a course' has been disambiguated. If 'a course' is taken to mean 'no course at all' (in the context), then the system will interpret question (A) to mean, "Who teaches no courses in any fall semester", and question (B) is interpreted to mean "Who does not teach in any semester of 84?". In both cases, the interpretation which the system finds seems to agree with the interpretation which is most commonly chosen, although some people find other interpretations. Still other people have trouble understanding these questions at all, and perhaps we should not expect a computer to find every interpretation for a question which many people find confusing. Nevertheless, if a question is genuinely ambiguous, and it is posed to the system by a user, then it is dangerous to have the system find only a single interpretation, even if that interpretation is the most common one. For presumably the user does have some interpretation in mind, which may not be the interpretation the system finds. Of course, humans do sometimes misinterpret each other’s questions and fail to discover the mistake, but we should not program a computer to do this if we can avoid it.

Ideally, the system should be able to ask the user to disambiguate queries whenever the context alone is not
sufficient to completely constrain the interpretation. In situations where the context can very reliably constrain the interpretation, the ideal solution may be to have the computer choose the only "reasonable" interpretation. At present, the SHADOW system never consults the context (in the sense of consulting "world knowledge") to decide what a reasonable interpretation would be. In some respects this is a weakness. For example, given the question "Which computing courses do not have a prerequisite?", the system will ask the user to choose between three possible interpretations where a reasonable person would only find one.

On the other hand, the system never makes the mistake of assuming that the reasonable interpretation is intended when, because of ignorance of the domain, a user's actual intended meaning is not a reasonable one in the context. For example, someone who is very new to a university might not know how unlikely it is that every instructor in a given department would teach the same course. If such a person were to ask "Does every math professor teach some course?" the person might actually mean "some particular course", when the "reasonable" interpretation, given some familiarity with universities, would be "some course or other". A great deal of "world knowledge" and heuristic programming would be required to enable a computer to discriminate between cases where a given interpretation is "reasonable", and cases where the interpretation is correct beyond any reasonable doubt. The general problem of using...
pragmatic information to constrain the interpretation of ambiguous sentences is clearly a very complex and interesting one, but it is beyond the scope of this thesis.

Another difficulty which remains for the SHADOW system is the treatment of such phrases as "some X of every Y", and "the X of every Y", as in "Is some prerequisite of every computing course a math course?" and "What are the prerequisites of every course?". For the two combinations of quantifiers, (some every) and (the every), which occur in prepositional phrases such as these, the system responds with the message "This query is too complex for us at present". The reasons are different in each case. Questions of the latter form are especially complicated because, when someone asks, "What are the prerequisites of every course", the person usually wants each course to be identified and listed with its prerequisites before the next course is listed with its prerequisites, and so on. In such cases we actually need to execute two separate kinds of "finds".

One approach to solving this problem is to treat the '(the every)' combination as a special case, and to replace these determiners with special "find" operators which can "communicate" with each other, through the use of flags, say. If such an approach is adopted, however, one must take care to distinguish cases where such treatment is needed from cases such as, "What are the names of every student?", which can be easily handled as "Find all Xs such that X is some name of some student or other". This interpretation can easily be found by the
present system, by using the 'Set-oper' procedure which was discussed in the last chapter.

A related problem exists for questions such as "Is some prerequisite of every computing course a math course?" The natural interpretation of this question is "Is it true that every computing course has a math prerequisite?" However, given the question, "Is some friend of every person in this room a football player?" we can see that another type of interpretation exists, namely, "Is someone who is a friend of everybody in this room a football player?". The system could easily be made to find this last interpretation, but the remaining interpretation, which is, "Does everybody in this room have some friend or other who is a football player?" is much more complicated. The system could be made to find this interpretation by reordering the quantifiers in the original question, so that the highest order quantifier becomes 'every' and the lower order quantifier becomes 'some'. Thus, "Is some I of every Y a Z" would become "Is is true that for every Y, some I of that Y is a Z". This approach is adopted by Dahl in [1981]. It is not adopted here because the process of reordering embedded quantifiers can become very complicated when the quantifiers are separated by more than one level of nesting.

Consider, for example, the question:

"Is it true that that some prerequisite of every course which is offered by some department from every faculty is offered next year?"
In this question the highest order quantifier is the 'every' which quantifies 'faculty'. The lowest order quantifier is 'some' which quantifies 'prerequisite'. Clearly, for this example, the process of reordering the quantifiers would be quite complicated. Rather than attempting to deal with this class of problems in the present work, it was judged desirable to treat this kind of problem in a separate work, where alternative solutions could be explored. It must be admitted that for most practical sentences the reordering strategy would be quite adequate, aside from the ambiguity problem, which exists in any case.

Another important area in which the existing system needs further development is the area of query optimization. In the previous chapter it was remarked that the structures which represent the intensions of noun phrases are constructed in such a way as to optimize query evaluation in a large majority of cases. In other places attempts have also been made to minimize the search space required for evaluation. Nevertheless, more could be done to optimize the minority of cases which are not presently optimized by the system. For example, whenever a noun phrase has a relative clause, the system currently evaluates the restrictions imposed by the relative clause before attempting to find an individual which satisfies the noun which heads that clause. This is because, in most cases, the relative clause is more restrictive than the head noun. This is not always the case, however.
Consider the example, "Which millionaires who live in Brazil are travelling in Europe this week?" Clearly, the class of millionaires is much smaller than the class of people living in Brazil. If this query were properly optimized, the predicate 'millionaire(X)' would be evaluated before the predicate 'lives-in(Brazil,X)'. The optimization process would require that the sizes of the relevant classes either be known beforehand or be calculated as needed. We cannot always assume that because the extension of a given noun, say 'millionaire' is small, that it will be smaller than the extension picked out by the relative clause. Consider the phrase, "millionaires who live on Mount Rainier". If we are to optimize all queries in a systematic fashion, we may have to build in a large amount of world knowledge, so that the sizes of the relevant classes can be calculated and compared. Although this may not be a problem for the average database, we must weigh the cost of optimizing every query against the cost of optimizing a large majority of queries with no computational overhead.

**Summary**

In this thesis a system has been presented which is capable of interpreting and answering what we believe to be a useful subset of English questions, which can be posed to a Prolog database. In Appendix A an extended series of example questions

---

I am indebted to Veronica Dahl for this example.
is displayed with the system's responses and with commentary. We believe that these examples demonstrate both that the system is capable of answering many informally phrased questions, which are likely to be posed to the system, and many complex questions which are interesting from a logical point of view. In every case the system gives a correct answer to the question, on a reasonable interpretation of the question, or gives a reason why the question cannot be answered. The system is capable of identifying three types of logical presupposition, and it also identifies those phrases in a user's question which contain false presuppositions.

The program which implements the SHADOW system has been designed in a highly structured way, with abundant commentary, so that the program can be expanded to include a broader range of syntax. Whenever possible the system strives for generality, and avoids ad hoc solutions, so that obstacles are not built into the system which preclude a much broader range of syntax and vocabulary. No claim is made that the system is indefinitely expandable, but we believe that a wide range of English questions can be accommodated by the present system. Because the system has been designed to be general, it may be applied to other databases, provided suitable changes are made to the program's semantic dictionary.

Probably the most unusual, and perhaps most interesting, aspect of the SHADOW system is the way in which it attempts to disambiguate queries dynamically, by questioning the user when
ambiguities are discovered in quantified statements. Although
the system does not resolve all ambiguities in very complex
quantified questions, it does so for many question types which
are recognised as problematic.

Possibilities

We turn now from the discussion of problem areas to
consider other ways in which the SHADOW system could be
enhanced. There are certain practical respects in which the
program could be improved. For example, the routines which fetch
a user's question could be expanded to include a spelling
correction module which would question the user about misspelled
words and offer a choice of corrections. Such a module is
included in the Plane's project [13]. The module should also
identify words which are completely unknown to the system.
Furthermore, more diagnostic procedures could be built into the
parsing process to help identify the portion of a user's query
which the system was unable to parse.

Also, the syntactic range of the system could be broadened
considerably. For example, the Relative-clause procedure could
easily be extended to recognise clauses which lack a relative
pronoun, as in, "Which students taking a course this semester
are enrolled in..." This could be achieved by permitting a
"null" relative pronoun to be consumed only when the verb-form
is a present participle. Another form of relative clause could
also be recognised, namely, those in which the order of nouns and verb-form are inverted. An example is, "Is the cat which the dog chased now in a tree?" This form of relative clause would require a greater modification of the Relative-clause procedure, but it presents no serious problem.

SHADOW's syntax could also be expanded to include indirect objects, as in "The department assigned the course to Frege" or "The department assigned Frege the course." This extension could probably be achieved with a small amount of modification to the two sentence rules which the program now recognises. (SHADOW's grammar can be found in Appendix B.) For example, in addition to attempting to parse direct objects, the system could attempt to parse an additional noun phrase in cases where the verb present indicates that an indirect object is present. The system could be flexible about whether an indirect object is present. In cases where the verb does not clearly require an indirect object (e.g., 'take'), an attempt could be made to parse one anyway. If the attempt fails, a "nil" intension and quantifier could be returned, as happens when no adverbial phrase is present. These nil values would be discarded at the point where the verb relation is composed. Also, in cases where two noun phrases are found to follow a verb, and the verb permits an indirect object, various checks would have to be made to distinguish the direct from the indirect object. These checks would include determining whether the particle 'to' is present, and if so, where. When 'to' is not present, order and type restrictions might be used
to decide the question, as in, "Mary gave the boy a book."

Another way in which SHADOW's range could be extended would be to expand the types of adverbial phrases recognised. This could be easily achieved for some types of adverbials, including those which indicate where an event is to take place. For example, in "Gauss is teaching a class in the classroom complex" or "John saw Mary at the dance", the adverbial phrases could be handled in a manner similar to the way in which SHADOW now parses "simple" temporal adverbs. Other types of adverbial phrases may prove difficult to accommodate, however, especially those which indicate the manner in which an action is performed. For example, in "Mary handed the glass to John carefully", the adverb could have been placed before the verb or just after the direct object. Also, the meaning of manner adverbs may be difficult to represent in a formal system, unless we treat such adverbs as primitives which occur as values in a relation's fields. Fortunately, because they tend to express subjective qualities, such adverbs are seldom used in data base queries.

An especially interesting extension which could be made to SHADOW's syntax would be to enable it to parse certain compound noun phrases and verb phrases. Examples of compound noun phrases occur in "How many math or computing students are taking an Arts course this semester?", and "Which professors teach math courses and computing courses?" Examples of compound verb phrases which the system could eventually recognise occur in "Which professors belong to computing and teach a math course?" and "Which
students took math 232 and failed math 300?" To accommodate compounds in noun and verb phrases the system would need procedures which are intermediate in level between Sentence procedures and lower level procedures such as the NP and Fancy-verb procedures. In some cases these intermediate procedures would have to be capable of taking "processed" noun phrases and verb relations as arguments, and of distributing the verb's relation over the noun intensions. Other cases would require a different treatment. For example, recall Dahl's treatment of the collective relation in "A and B are parallel." [Dahl (1981)]

Both Winograd and Dahl have successfully incorporated several forms of conjunction in a single-pass system. Nevertheless, conjunctions pose an especially challenging problem to the computational linguist. In chapter 4 we discussed the fact that there are certain context sensitive compound sentences which do not appear to be analysable in a single-pass system. There is very good reason to believe, therefore, that the SHADOW system cannot be extended indefinitely. Nevertheless, with suitable modification, this system can be made to recognise quite a useful range of English syntax.
APPENDIX A

In this appendix we demonstrate the behaviour of the SHADOW system on a wide range of questions. Commentary is included for most of the questions shown, to explain what capacity of the system the particular question is demonstrating. In the series of questions which immediately follow, we demonstrate some natural sounding questions, many of which begin with interrogative pronouns. In such cases the existence of a "dummy" noun phrase is inferred where the pronoun occurs.

who is teaching computing 405 next spring?

peters

that is all

seconds used = 1.70001

[In the question just above, a dummy noun phrase is inferred where the pronoun 'who' occurs. No restrictions are placed upon the intension or type of the noun, except those required by the verb 'teaches'. This restriction is left implicit, however. Anything which is found to be 'teaching computing 405 next spring' is returned as the answer.]
What computing courses are offered in the summer of 83?

Nothing satisfies the conditions you specify.

seconds used = 17.8

[In the question above the "deep subject" of the question, which is what is "doing the offering", is inferred and made explicit in the formal query which is evaluated. In this case, the program took a long time to answer the question. This is because the program never reports a negative answer without backtracking and looking to see whether other interpretations exist for the question.]
who is not teaching a course this semester?

In the following phrase, which concludes your question:

a course this semester

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.
Or do you mean no course at all?
In this last case just type the numeral 3

euler
fermat
ullman
codd
caplin
hadley
biagioni
peters
hell
liestman
battacharya
funt
dahl
cerceone

that is all

seconds used = 3.91669

[In the above example, the type of the implied noun phrase, represented by 'who', is inferred to be 'prof' by inspecting the argument types required by the verb, 'teaches'. Also, note that the presence of the negation in the question causes a request for disambiguation of the determiner 'a'. In this context the disambiguation is probably unnecessary, but there are contexts where the disambiguation is needed. The program does not have the ability to distinguish between contexts on pragmatic grounds.]
who belongs to the department of mathematics?

gauss
euler
fermat

that is all

seconds used = 2.26666

[In the above example, we have the verb 'belongs' which requires the particle, 'to'. The particle is consumed in the process of parsing a "Fancy-verb".]

who is a member of mathematics?

gauss
euler
fermat

that is all

seconds used = 2.36667

[In the question above we have the noun 'member' appearing as the head of a prepositional phrase. The noun is examined to see whether it has an implication. It does imply the phrase, "professor who belongs to" which is used to replace the phrase 'member of'. In this way the program makes sense of 'member of mathematics'.]
which professors who are teaching this fall do not teach next spring?

minsky
liu

that is all

seconds used = 3.60004

[In the example above there are two unstated noun phrases which the program must discover. In both cases the noun phrases concern "what is taught".]

is a computing course offered in every semester of 84?
yes it is true

seconds used = 4.20001

[In the above case we have a sentence which begins with 'is'. The sentence is treated as equivalent to "Is it true that a computing course is offered in every semester of 84?" Note, in this case the "deep subject" is inferred once again.]
how many math courses are offered this year?

The numerical answer is 4

seconds used = 2.28333

[Once again, in the above example we have the "deep subject" being inferred. In this case the question type is "how many".]

who is teaching every course?

your sentence contains words or syntax which are unknown to this system please check your spelling or rephrase your question

seconds used = 3.60335

[In the above example we see that the program does not yet have the ability to report the individual word which is mispelled. The program can be easily modified to include this ability.]
who is teaching math 234?

this course is not offered by any department

math-234

seconds used = 3.15002

[In the above example, the unknown course is identified and reported, and no attempt is made to evaluate the query.]

who is teaching math 808?

misplaced question mark

please reenter the query

[In the above case we demonstrate that the program does some error checking when reading the user's input.]
who is teaching math 808?
gauss
that is all
seconds used = 1.76666

who is teaching philosophy 101?
Only blanks may follow a question mark.
Please reenter your question.

[In the case above, we have another example of error checking. This is done in case the user accidentally types a question mark in the middle of a query.]
who is teaching philosophy 101?

beauvoir
occam
spinoza

that is all

seconds used = 1.85004

[Above, we show the question correctly entered and answered.]

what prerequisites does computing 405 have?

computing 205
computing 201
mathematics 152

that is all

seconds used = 5.66669

[In the above example, we see that the main verb, 'have', which sometimes functions as an auxiliary, is recognized here as having the meaning 'possess'.]
which computing course has a math prerequisite?

the answer is

computing 405

seconds used = 1.66666

[In the above example we show that the same reasoning is applied to the verb 'has'. Also, we see here that only one computing course has a math prerequisite. This is relevant to the next example.]

which computing courses have no math prerequisites?

computing 103
computing 118
computing 205
computing 410
computing 354
computing 105
computing 201

that is all

seconds used = 8.54999
which course is not a course?

In the following phrase, which concludes your question:

a course

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2. Or do you mean no course at all? In this last case just type the numeral 3.

There is a false presupposition concerning the existence of a value which satisfies the description of the subject of your question.

perhaps you would care to rephrase your question

seconds used = 4.78333

[In the above example, we have a case where the program correctly answers an odd question, by reporting a false presupposition.]
which department does beauvoir belong to?

the answer is

philosophy

seconds used = 1.51666

[In the above question, we have an inversion between the auxiliary verb and the deep subject of the sentence. The sentence is treated as a passive sentence.]
is it true that every professor who teaches every course is a professor?

There is a false presupposition in your question
Concerning the existence of a set of entities which
Satisfies a description in your question appearing
At the beginning of the phrase shown or described below.

ey every professor who teaches every course is a professor

seconds used = 24.6166

[Above we have a question which we would answer "true" if we were being pedantic (since it is not the case that something satisfies the subject without satisfying the object), but the program responds in a more sensible fashion by reporting the false presupposition.]

which courses offer philosophy?

There appears to be a semantic or physical impossibility implied in your question

perhaps you would care to rephrase your question

seconds used = 3.18335

[In the question above, the nonsensical nature of the question is detected by checking the types required by the verb. No attempt is made to evaluate the query.]
who teach computing 354?
codd
that is all
seconds used = 1.15002

[In the above example, we see that the program is able to handle some forms of ungrammatical input. The program does not check for agreement in number, except when distinguishing singular from plural determiners.]

which departments is offer a math course?
mathematics
that is all
seconds used = 1.90002

[Here we have another example of ungrammatical input being sensibly interpreted and answered.]
is it true that every professor teaches some course?

In the following phrase, which concludes your question:

some course

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.

1

No it is not true.

seconds used = 8.34999

[In this case, above, we have an ambiguity in the interpretation of the quantifier 'some'. Depending upon what answer the user supplies, different orders are chosen for the quantifiers 'every' and 'some' in the internal query.]
is it true that every professor teaches some course?

In the following phrase, which concludes your question:

some course

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.

2

yes it is true

seconds used = 2.68335

[In the example above, we see that it is true that every professor does teach some course or other, although not the same course.]
does every professor who teaches a philosophy course teach every philosophy course?

yes it is true

seconds used = 2.90002

[In the above example, we see that an unusual situation exists in the philosophy department of this small imaginary university. Also, we show that the program can correctly answer questions involving multiple quantifiers.]

does every professor who teaches a math course teach every math course?

No it is not true

seconds used = 7.71655

[Here we show that the analogous situation does not exist in the math department.]
is it true that every department which offers some course is a department which does not offer some course?

[The question just above is ambiguous. The system now queries the user about the intended interpretation. There are six different possibilities altogether. Two possible interpretations exist for the first occurrence of 'some', and three possibilities exist for the second occurrence. This makes a total of six possible combinations. Each combination is tested in the following series. In each case the question is correctly answered.]

In the following phrase, which concludes your question:

some course is a department which does not offer some course

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.

1

In the following phrase, which concludes your question:

some course

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.
Or do you mean no course at all?
In this last case just type the numeral 3

1

yes it is true

seconds used = 6.66666
is it true that every department which offers some course is a department which does not offer some course?

In the following phrase, which concludes your question:

some course is a department which does not offer some course

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2.

1

In the following phrase, which concludes your question:

some course

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2. Or do you mean no course at all? In this last case just type the numeral 3.

2

yes it is true

seconds used = 4.79993
is it true that every department which offers some course is a department which does not offer some course?

In the following phrase, which concludes your question:

some course is a department which does not offer some course

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.
1

In the following phrase, which concludes your question:

some course

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.
Or do you mean no course at all?
In this last case just type the numeral 3
3

No it is not true

seconds used = 28.0334
is it true that every department which offers some course is a department which does not offer some course?

In the following phrase, which concludes your question:

some course is a department which does not offer some course

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2.

In the following phrase, which concludes your question:

some course

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2. Or do you mean no course at all? In this last case just type the numeral 3.

No it is not true

seconds used = 38.6833
is it true that every department which offers some course is a department which does not offer some course?

In the following phrase, which concludes your question:

some course is a department which does not offer some course

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.

In the following phrase, which concludes your question:

some course

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.
Or do you mean no course at all?
In this last case just type the numeral 3

yes it is true

seconds used = 33.8667
is it true that every department which offers some course is a department which does not offer some course?

In the following phrase, which concludes your question:

some course is a department which does not offer some course

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2.

2

In the following phrase, which concludes your question:

some course

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2. Or do you mean no course at all? In this last case just type the numeral 3.

3

No it is not true

seconds used = 10.4667
who is not a member of the department which offers every course which is taught by funt?

In the following phrase, which concludes your question:

a member of the department which offers every course which is taught by funt

The words at the beginning are ambiguous. Do you mean some particular member? If so, just type the numeral 1. Or do you mean just some member or other? In this second case just type the numeral 2. Or do you mean no member at all? In this last case just type the numeral 3.

beauvoir
occam
spinoza
gauss
euler
fermat

that is all

seconds used = 10.5166

[In the above example, the type of the noun phrase represented by 'who' is inferred to be human.]
which computing professor teaches in every semester of some year?

In the following phrase, which concludes your question:

some year

The words at the beginning are ambiguous. Do you mean some particular year? If so, just type the numeral 1. Or do you mean just some year or other? In this second case just type the numeral 2.

the answer is

caplin

seconds used = 15.4667

[In the above example we see that there is a professor who teaches in every semester of a given year. When we answer '2', meaning 'some year or other', we are informed that there is a false presupposition in our question, since there is no such professor.]
which computing professor teaches in every semester of some year?

In the following phrase, which concludes your question:

some year

The words at the beginning are ambiguous.
Do you mean some particular year?
If so, just type the numeral 1
Or do you mean just some year or other?
In this second case just type the numeral 2.

2

There is a false presupposition concerning the existence of a value which satisfies the description of the subject of your question.

perhaps you would care to rephrase your question

seconds used = 25.9834

does every professor who teaches this year teach this fall?

No it is not true

seconds used = 4.34998
does every professor who teaches this fall teach this year?

yes it is true

seconds used = 1.81665

[In the two previous examples, we have missing noun phrases being inferred both in the relative clause and in the object phrase.]

is it true that every course which is offered by the department which offers every course which is taught by every professor who teaches every course which is offered by philosophy is a philosophy course?

yes it is true

seconds used = 35.4

[In the above example, we show that the program can handle very deeply nested relative clauses which involve quantifiers. The answer in this case was 'yes' only because in the philosophy department, every course is taught by every philosophy professor, and only by philosophy professors. Later, when we ask a similar question about math courses, we are informed of a false presupposition.]

132
which professors teach in the fall of every year?

Nothing satisfies the conditions you specify.

seconds used = 6.91669

what is offered by computing in every semester of 84?

computing 103
computing 118
computing 205
computing 405
computing 410

that is all

seconds used = 11.7833

[In the last two examples, we are simply demonstrating that the program can handle complicated adverbial phrases. We show other kinds of adverbial phrases which the program can handle in the next eight examples.]
who is teaching a computing course in the coming semester?

caplin
hadley
peters
funt

that is all

seconds used = 4.11667

which courses were offered by computing in the spring?

computing 354
computing 105
computing 201

that is all

seconds used = 4.05002

[In the above example, the phrase 'in the spring' is interpreted as referring to the previous spring, since the past tense is used. In the next example, 'in the spring' is interpreted as referring to the coming spring, since the future tense is used.]
which courses will be offered by computing in the spring?

computing 103
computing 118
computing 205
computing 405
computing 410

that is all

seconds used = 4.46667

what courses were offered by math during the previous year?

mathematics 101

that is all

seconds used = 3.96667
which professors who teach in the spring of 84 will be teaching during the summer?

caplin
hadley

that is all

seconds used = 3.95001

[In the above case, we have adverbial phrases in both an embedded clause and in the major clause.]
who does not teach in some semester of 84?

In the following phrase, which concludes your question:

some semester of 84

The words at the beginning are ambiguous. Do you mean some particular semester?
If so, just type the numeral 1
Or do you mean just some semester or other?
In this second case just type the numeral 2.

2

beauvoir
occam
spinoza
gauss
euler
fermat
ullman
minsky
codd
liu
hadley
biagioni
peters
hell
liestman
battacharya
funt
dahl
cercone

that is all

seconds used = 6.85004

[In the above example, we are given a list of all those people for whom there is a semester in 84 when they do not teach.]
who teaches a course in every semester of 84?
caplin
that is all
seconds used = 11.7333

who teaches no courses in every semester of 83?
caplin
hadley
biagioni
peters
hell
liestman
battacharya
funt
dahl
cercone
that is all
seconds used = 7.1167

[The above example is ambiguous. The system interprets the question to mean 'Find those people who do not teach in any semester of 83'. This seems to be the most common interpretation of the question. However, it could be interpreted to mean 'Find those professors such that it is not true that they teach during all semesters of 83'.]
how many math courses which were offered last spring were taught by a professor who belongs to philosophy?

There are none

seconds used = 6.45001

[In the next seven examples we show the ability of the program to interpret negation. In the first example we find the negation operator in the place where auxiliary verbs normally occur.]

which computing courses are not offered this year?

computing 103
computing 118
computing 405

that is all

seconds used = 6.91666
who is teaching no courses next year?

fermat
ullman
minsky
codd
liu

that is all

seconds used = 3.23334

(In the example above we have the negation being introduced by the negative determiner 'no'.)
which professors do not teach a course in the fall of every year?

In the following phrase, which concludes your question:

a course in the fall of every year

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2. Or do you mean no course at all? In this last case just type the numeral 3.

The question shown above is even more ambiguous than we might expect. Even after the intended interpretation of the existential quantifier is resolved, there is still a question about the order of quantifiers in the verb phrase. The most common interpretation of this question seems to be, "Which professors are such that they do not teach any courses in any fall semester?" This is the interpretation which SHADOW now answers, (when the user responds with '3'). However, one could interpret the question to mean, "Which professors are such that there is a fall semester when they do not teach?" The program could be easily changed to find this interpretation.
which professors do not teach a course in the fall of every year?

In the following phrase, which concludes your question:

course in the fall of every year

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2. Or do you mean no course at all? In this last case just type the numeral 3.

beauvoir
occam
spinoza
.
.(the correct list is printed)
.
funt
dahl
cercone

that is all

seconds used = 18.1167

[When the user chooses the second interpretation for the existential quantifier, the question is interpreted to mean, "Which professors are such that there is a course (which may vary with the given professor) which each professor does not teach in any fall semester." When this interpretation is answered, we see that every professor has this property.]
is it true that every member of a department which does not offer computing 405 is a professor who does not teach a computing course?

In the following phrase, which concludes your question:

a computing course

The words at the beginning are ambiguous. Do you mean some particular course? If so, just type the numeral 1. Or do you mean just some course or other? In this second case just type the numeral 2. Or do you mean no course at all? In this last case just type the numeral 3.

3.
yes it is true.

seconds used = 5.45001

[In the above example, we see that the program can handle negations in relative clauses as well as in the major clause. Any number of negations may appear in a well formed query, and the program will still correctly answer the question on a reasonable interpretation of the question.]
who is not teaching this year?
caplin
hadley
biagioni
peters
hell
liestman
battacharya
funt
dahl
cercone

that is all

seconds used = 3.54999
who teaches no courses this year?
caplin
hadley
biagioni
peters
hell
liestman
battacharya
funt
dahl
cercone

that is all

seconds used = 2.90002

[In the two previous examples, we see that synonymous negative questions are answered correctly by the program. In the first example, the missing noun phrase was inferred and inserted into the query.]
is it true that a professor who does not teach a course which is not offered by computing is a computing professor?

In the following phrase, which concludes your question:

a course which is not offered by computing is a computing professor

The words at the beginning are ambiguous.
Do you mean some particular course?
If so, just type the numeral 1
Or do you mean just some course or other?
In this second case just type the numeral 2.
Or do you mean no course at all?
In this last case just type the numeral 3

yes it is true

seconds used = 3.41669

[In the above example, we have a complicated question involving two negations at different levels of nesting. The question is correctly answered.]
what are the prerequisites for computing 405?

computing 201
computing 205
mathematics 152

that is all

seconds used = 4.76666

(The above example shows a case where a prepositional phrase is analysed by extracting implied information from the word 'prerequisite'. The sentence is interpreted to mean, "What are the courses required by computing 405?".)
is some prerequisite of every computing course a math course?

In the following phrase, which concludes your question:

some prerequisite of every computing course a math course

The words at the beginning are ambiguous. Do you mean some particular prerequisite? If so, just type the numeral 1. Or do you mean just some prerequisite or other? In this second case just type the numeral 2.

2

This query is too complex for us at present.

seconds used = 8.08334

[In the above example, we have a query we is currently beyond the capacities of the program. In this case the program reports its limitation.]
what are the prerequisites for the course which is taught by
ullman in the year of 82 in the fall?

computing 201
computing 205
mathematics 152

that is all

seconds used = 62.4833.

[In the above example, we have a question which contains
several prepositional phrases. In the case of those prepositions
which appear between words which do not, by themselves, imply
what the connection is that is being implied by the preposition,
the program reasons about what the relation could be, on the
basis of what relations the relevant types of nouns occur in.]
is it true that every professor who teaches a course for philosophy in the fall of some year is a professor who teaches a course which is offered by the department of philosophy in a semester of 83?

In the following phrase, which concludes your question:

some year is a professor who teaches a course which is offered by the department of philosophy in a semester of 83

The words at the beginning are ambiguous. Do you mean some particular year? If so, just type the numeral 1. Or do you mean just some year or other? In this second case just type the numeral 2.

2

yes it is true

seconds used = 16.6334

[In the previous example, a question containing many prepositional phrases is correctly answered. Note that if we change the year 83 to the year 82, as we do below, the answer changes. Both answers are correct.]
is it true that every professor who teaches a course for philosophy in the fall of some year is a professor who teaches a course which is offered by the department of philosophy in a semester of 82?

In the following phrase, which concludes your question:

some year is a professor who teaches a course which is offered by the department of philosophy in a semester of 82

The words at the beginning are ambiguous.
Do you mean some particular year?
If so, just type the numeral 1
Or do you mean just some year or other?
In this second case just type the numeral 2.

No it is not true

seconds used = 46.9833
how many professors are not a member of the department which offers a course in every semester of 84?

In the following phrase, which concludes your question:

a member of the department which offers a course in every semester of 84

The words at the beginning are ambiguous.
Do you mean some particular member?
If so, just type the numeral 1
Or do you mean just some member or other?
In this second case just type the numeral 2.
Or do you mean no member at all?
In this last case just type the numeral 3

The numerical answer is 6
seconds used = 24.0167
who is not a member of the department which offers a course in every semester of 84?

In the following phrase, which concludes your question:

a member of the department which offers a course in every semester of 84

The words at the beginning are ambiguous. Do you mean some particular member? If so, just type the numeral 1. Or do you mean just some member or other? In this second case just type the numeral 2. Or do you mean no member at all? In this last case just type the numeral 3.

beauvoir
occam
spinoza
gauss
euler
fermat

that is all

seconds used = 25.3167

[In the remaining series of questions, false presuppositions are detected and reported. In the first example, below, there is a presupposition that one and only one professor teaches computing 405. This presupposition turns out to be false.]
which professor teaches CMPT 405?

there is a false presupposition concerning the uniqueness of the values which satisfy the description which begins the portion of your question shown or described below:

The subject of your question is the intended phrase.

At least the two values below were found.

battacharya
peters

perhaps you would care to rephrase your question

seconds used = 3.38342

which professors teach computing 405?

ullman
peters
liestman
battacharya

that is all

seconds used = 3.54993
What courses are offered by the department which offers the course which is taught by every professor?

There is a false presupposition concerning the existence of a value which satisfies the description which appears at the beginning of the portion of your question shown below:

the course which is taught by every professor

Perhaps you would care to rephrase your question.

seconds used = 21.7001

[In the previous example, we have a question which contains more than one false presupposition. The most deeply nested error is the one reported, since when deeply nested errors are corrected, other false presuppositions often are corrected, as in the present case.]
what is the course which is offered by the department which offers a course which is taught by dahl?

there is a false presupposition concerning the uniqueness of the values which satisfy the description which begins the portion of your question shown or described below:

The subject of your question is the intended phrase.

At least the two values below were found.

cmpt-118

cmpt-103

perhaps you would care to rephrase your question

seconds used = 14.3999
is it true that every course which is offered by the department which offers every course which is taught by every professor who teaches every course which is offered by math is a math course?

There is a false presupposition concerning the existence of a value which satisfies the description which appears at the beginning of the portion of your question shown below:

the department which offers every course which is taught by every professor who teaches every course which is offered by math is a math course

perhaps you would care to rephrase your question

seconds used = 29.75

which departments offer every course which is taught by every professor?

There is a false presupposition in your question Concerning the existence of a set of entities which satisfies a description in your question appearing At the beginning of the phrase shown or described below.

every course which is taught by every professor

seconds used = 15.2168

[In the above example, a false presupposition introduced by the word 'every' is detected and reported.]
which department offers the course which is taught by occam?

there is a false presupposition concerning the uniqueness of the values which satisfy the description which begins the portion of your question shown or described below:

the course which is taught by occam

At least the two values below were found.

phil-808
phil-101

perhaps you would care to rephrase your question?

seconds used = 4.4834

which department offers a course which is taught by occam?

the answer is

philosophy

seconds used = 1.6499
which professor teaches every course?

There is a false presupposition concerning the existence of a value which satisfies the description of the subject of your question.

Perhaps you would care to rephrase your question.

seconds used = 22
APPENDIX B

In this appendix the grammar which the SHADOW system recognises is presented. The grammar is intended to be an approximate outline rather than a grammar which is accurate in every detail. This is because there are certain restrictions on a few of the rules. The restrictions are:

There may not be more than one "null" noun phrase in the top level of a sentence.

The determiner 'no' may determine only those noun groups which follow a verb group.

The rule 'QNP -> NP' may only be used when the question phrase produces the question code 'is it true that'.

S -> JUES-PHRASE + QNP + V-FORM + NP + ADVERBIAL-PH
S -> JUES-PHRASE + QNP + AUX + NP + VERB + ADVERBIAL-PH

JUES-PHRASE -> which | how many | what are the | who are the | what is the | who is the | is it true that | who | what | is

QNP -> PROPER-NOUN

QNP -> DET + ADJ = NOUN REL-CLAUSE

QNP -> SIMP-NP + PREP + NP

QNP -> empty string

SIMP-NP -> DET + ADJ = NOUN

DET -> the | every | a | some | no

QNP -> NP

QNP -> ADJ = NOUN + REL-CLAUSE

QNP -> ADJ = NOUN + PREP + NP
V-FORM -> ACTIVE-FORM | PASSIVE-FORM

ACTIVE-FORM -> VERB | AUX + VERB

PASSIVE-FORM -> AUX + PPART + by

AUX -> AUX-WORD + AUX= | empty string

AUX-WORD -> not

AUX-WORD -> is | are | was | were | have |
  has | have | been | being | do | does |
  did

VERB -> teach | require | offered | instructs
  (all tenses and participles are
  included in the list of accepted
  verbs)

PPART -> taught | required | offered | instructed
  | belonged to | prerequired | was | had

REL-CLAUSE -> REL-PRONOUN + V-FORM + NP | EMPTY STRING

REL-PRONOUN -> that | which | who

ADJ -> DEPT-NAME | empty string

ADJ -> ADJ=

DEPT-NAME -> mathematics | computing | philosophy

PROPER-NAME -> PROF-NAME | COURSE-NAME | DEPT-NAME

PROF-NAME -> some actual names will go here

COURSE-NAME -> actual course names will go here
WJN -> course | professor | prerequisite | people | member | instructor
(and others, including plurals of these).

PREP -> in | with | for | of | from

ADVERBIAL-PH -> SIMPLE-ADV | COMPLEX-ADV

SIMPLE-ADV -> T-PREP + DETERM + ORDERWORD + TIMENOUN

COMPLEX-ADV -> T-PREP + DET + TIMENOUN + COMPLEX-ADV

COMPLEX-ADV -> empty string

T-PREP -> in | during

DETERM -> the | empty string

ORDERWORD -> this | next | last | coming
    present | previous
    | future | past

TIMENOUN -> year | spring | summer | fall | semester
APPENDIX C - PROGRAM LISTING

NOTE: Due to restrictions imposed by the device which formatted this text, certain symbols differ here from those which appear in the original program code. In particular, the tilda is replaced by '6', and the backwards slash is represented here by the forward slash, '/'. Also, the program has been reformatted to fit within normal page size. In some cases this has caused character strings to be divided over two lines. This would cause syntax errors if the program were loaded in the form shown here.

{ First we define certain infix operators which are used by the program. The slash '/' is used as a list operator. Thus we create the list [ a, b, c] by writing a/b/c/nil, in my slightly amended version of prolog. Also, we define '6' as an operator which binds seasons to years. }

:- op(5, xfy, [ '/ ' ]).
:- op(8, xfy, [ '6' ]).

{ The first 300 lines of this file comprise the dictionary and database which the program uses. First the dictionary is entered. The database follows, together with some semantical rules of inference. }

get-adj( phil).
get-adj(math).
get-adj(comp).

{ Adjectives are defined by using patterns shown below. Adjective senses are selected on the basis of what type of entity they are supposed to be modifying. It happens that in our database, all adjectives are also the proper names of departments, and that an adjective can only modify a course, a professor, or a department. However, the schema shown below is absolutely general, since the type slot can always be left as a variable in the definition of a particular adjective. In this way we can also define adjectives which do not discriminate on the basis of types, if there are any such adjectives. }

adj(Adj, course, X) :- offers(Adj, X, 5).
adj(Adj, prof, X) :- member(X, Adj).
adj(Adj, dept, X) :- equal(Adj, X).
{ Since the proper nouns for professors and courses are non-redundantly listed in the member and offerings relations respectively, these proper nouns can be located by searching those lists. However, the proper nouns representing department names are only listed in a redundant fashion in the relations of the database. Therefore, we give a non-redundant listing of department names in the following fashion. }

depth(math).
depth(capt).
depth(phil).

{ Rather than explicitly asserting that each person who teaches is a professor, e.g., prof(occam), etc., we will use rules of the form,

\[ \text{prof}(x) \implies \text{member}(\text{dept}, x). \]

our entries for particular nouns, e.g., professor can now take the form shown below:

\[ \text{get-noun(professor,sg,prof}(X),\text{prof},X). \]

For example, this entry tells us that 'professor' is a noun, singular, and that to establish that \( x \) is a professor, establish that \( \text{prof}(X) \) (which we do by means of a rule), and further, it tells us that the type of a professor is prof and that the variable name identifying a professor in the is \( X \). }

\[ \text{get-noun(professor,sg,prof}(X),\text{prof},X). \]
\[ \text{get-noun(professors,pl,prof}(X),\text{prof},X). \]
\[ \text{get-noun(courses,pl,course}(X),\text{course},X). \]
\[ \text{get-noun(department,sg,dept}(X),\text{dept},X). \]
\[ \text{get-noun(departments,pl,dept}(X),\text{dept},X). \]
\[ \text{get-noun(fall, sg, equal}(X, \text{fall}\&\text{Year}), \text{season}, X). \]
\[ \text{get-noun(spring, sg, equal}(X, \text{spring}\&\text{Year}), \text{season}, X). \]
\[ \text{get-noun(summer, sg, equal}(X, \text{summer}\&\text{Year}), \text{season}, X). \]
\[ \text{get-noun(semester, sg, equal}(X, \text{Season}\&\text{Year}), \text{semester}, X). \]
\[ \text{get-noun(semester, pl, equal}(X, \text{Season}\&\text{Year}), \text{semester}, X). \]
\[ \text{get-noun(year, sg, equal}(X, \text{Season}\&\text{Year}), \text{year}, X). \]
\[ \text{get-noun(years, pl, equal}(X, \text{Season}\&\text{Year}), \text{year}, X). \]
\[ \text{get-noun(summer, sg, equal}(X, \text{summer}), \text{sea}, X). \]
\[ \text{get-noun(fall, sg, equal}(X, \text{fall}), \text{sea}, X). \]
\[ \text{get-noun(season, sg, season}(X), \text{sea}, X). \]
\[ \text{get-noun(seasons, pl, season}(X), \text{sea}, X). \]
\[ \text{get-noun(year, sg, year}(X), \text{yr}, X). \]
\[ \text{get-noun(years, pl, year}(X), \text{yr}, X). \]

\{ The rule below is used to find the internal synonym for a noun

164
which the system should recognize. )

get-noun(W,Num,Pred,Type,Var)
    :- syn(W,S) , get-noun(S,Num,Pred,Type,Var).

{ Below appear two semantical rules which tell us how to establish whether something is a professor or a course. }

prof(X) :- member(X, D).
course(X) :- offers(D, X, S).
semest(X) :- year(Y), season(S), equal(X, S&Y).

{ Below are two semantical rules which tell us how to establish whether something is a human or an entity. These must be updated whenever new types of nouns are introduced to the system. }

human(X) :- prof(X).
entity(X) :- course(X) ; dept(X) ; human(X) ; semest(X).

{ Now the verbs are defined. An entry of the form

get-verb( teach,present,teaches,both)

tells us that 'teach' is a verb of present tense, that its internal relation name is 'teaches', and that it can appear with both singular and plural nouns. }

get-verb( teach,present,teaches,both).
get-verb( teaches,present,teaches,sg).
get-verb( taught,past,teaches,both).
get-verb( teaching,prespart,teaches,both).

get-verb( require,present,prereq,both).
get-verb( requires,present,prereq,sg).
get-verb( required,past,prereq,both).
get-verb( requiring,prespart,prereq,both).

get-verb( offer,present,offers,both).
get-verb( offers,present,offers,sg).
get-verb( offered,past,offers,both).
get-verb( offering,prespart,offers,both).

get-verb( belong, present, member, pl).
get-verb( belongs, present, member, sg).
get-verb( belonged, past, member, both ).

get-verb( V,Vtense,Rel,Num)
    :- syn(V, S), get-verb(S,Vtense,Rel,Num).

{ The verbs below function as main verbs only when they are not followed by a verb. }
get-verb(is,present,equal,sg).
get-verb(are,present,equal,pl).
get-verb(was,past,equal,sg).
get-verb(were,past,equal,pl).

{ Below are listed the auxiliary verbs which the program
recognizes. }

gat-aux(is, present, Neg).
gat-aux(are, present, Neg).
gat-aux(was, past, Neg).
gat-aux(were, past, Neg).
gat-aux(not, Tense, not).
gat-aux(have, Var, Neg).
gat-aux(has, Var, Neg).
gat-aux(will, future, Neg).
gat-aux(be, future, Neg).
gat-aux(being, Var, Neg).
gat-aux(hasen, past, Neg).
gat-aux(do, Var, Neg).
gat-aux(does, present, Neg).
gat-aux(did, past, Neg).

{ Now rules for recognizing prepositions are introduced.
Currently the prepositions, 'for', 'of', 'from', and 'in' are
recognised. }

prep(for/Rest, Rest, for) :- !.
prep(of/Rest, Rest, of) :- !.
prep(from/Rest, Rest, from) :- !.
prep(in/Rest, Rest, in ) :- !.

vague-prep( P/Rest, Rest)     :- or( equal(P, for), equal(P, of)).

{ Below are a few semantical rules which tell us what phrases
are implied by words like 'prerequisite' and 'from', as in 'the
man from Tasmania'. }

phraseimpl( prerequisite, which/is/required/by/nil, Prep)    :- or( equal(Prep, for), equal(Prep, of)), !.

phraseimpl( prerequisites, which/are/required/by/nil, Prep)    :- or( equal(Prep, for), equal(Prep, of ) ), !.

phraseimpl( Any, who/comes-from/nil, from) :- !.

phraseimpl( member, who/belongs/to/nil, of ) :- !,
phraseimpl( members, who/belong/to/nil, of ) .

{ In this section we list expressions which are recognized to
have internal synonyms. In a pair of the form \( \text{syn}(a, b) \), \( b \) is the internal synonym for \( a \). 

\[
\begin{align*}
\text{syn} & (\text{mathematics, math}). \\
\text{syn} & (\text{computing, cmpt}). \\
\text{syn} & (\text{philosophy, phil}). \\
\text{syn} & (\text{member, professor}). \\
\text{syn} & (\text{members, professors}). \\
\text{syn} & (\text{prerequisites, courses}). \\
\text{syn} & (\text{prerequisite, require}). \\
\text{syn} & (\text{prequires, requires}). \\
\text{syn} & (\text{prerequired, required}). \\
\text{syn} & (\text{prequiring, requiring}). \\
\text{syn} & (\text{have, require}). \\
\text{syn} & (\text{has, requires}). \\
\text{syn} & (\text{prerequisite, course}). \\
\text{syn} & (\text{instruct, teach}). \\
\text{syn} & (\text{instructs, teaches}). \\
\text{syn} & (\text{instructed, taught}). \\
\text{syn} & (\text{instructing, teaching}). \\
\text{syn} & (\text{offered, offered}). \\
\text{syn} & (\text{offering, offering}). \\
\text{syn} & (\text{previous, last}). \\
\text{syn} & (\text{current, this}). \\
\text{syn} & (\text{future, next}). \\
\text{syn} & (\text{coming, next}). \\
\text{syn} & (\text{present, this}). \\
\text{syn} & (\text{past, last}). \\
\text{syn} & (\text{prespalt, this}).
\end{align*}
\]

{ Now we enter some values for the four relations in the database }

\[
\begin{align*}
\text{member} & (\text{beauvoir, phil}). \\
\text{member} & (\text{occam, phil}). \\
\text{member} & (\text{spinoza, phil}). \\
\text{member} & (\text{gauss, math}). \\
\text{member} & (\text{euler, math}). \\
\text{member} & (\text{fermat, math}). \\
\text{member} & (\text{ullman, cmpt}). \\
\text{member} & (\text{minsky, cmpt}). \\
\text{member} & (\text{codd, cmpt}). \\
\text{member} & (\text{liu, cmpt}). \\
\text{member} & (\text{caplan, cmpt}). \\
\text{member} & (\text{hadley, cmpt}).
\end{align*}
\]
member (biagioni, cmpt).
member (peters, cmpt).
member (hell, cmpt).
member (liestman, cmpt).
member (battacharyya, cmpt).
member (funt, cmpt).
member (dahl, cmpt).
member (cercone, cmpt).
prereq (math-303, math-202).
prereq (math-303, math-101).
prereq (cmpt-118, cmpt-103 ).
prereq (cmpt-205, cmpt-105 ).
prereq (cmpt-205, cmpt-118 ).
prereq (cmpt-405, cmpt-201 ).
prereq (cmpt-405, cmpt-205).
prereq (cmpt-405, math-152).
prereq (cmpt-410, cmpt-201).
prereq (cmpt-410, cmpt-205).

offers (cmpt, cmpt-103, spring 84).
offers (cmpt, cmpt-103, summer 84).
offers (cmpt, cmpt-103, fall 84).
offers (cmpt, cmpt-118, spring 84).
offers (cmpt, cmpt-118, summer 84).
offers (cmpt, cmpt-118, fall 84).
offers (cmpt, cmpt-205, spring 84).
offers (cmpt, cmpt-205, summer 84).
offers (cmpt, cmpt-205, fall 84).
offers (cmpt, cmpt-405, spring 84).
offers (cmpt, cmpt-405, summer 84).
offers (cmpt, cmpt-405, fall 84).
offers (cmpt, cmpt-410, spring 84).
offers (cmpt, cmpt-410, summer 84).
offers (cmpt, cmpt-410, fall 84).
offers (cmpt, cmpt-205, fall 83).
offers (cmpt, cmpt-354, spring 83).
offers (cmpt, cmpt-405, fall 82).
offers (cmpt, cmpt-410, fall 83).
offers (cmpt, cmpt-105, spring 83).
offers (cmpt, cmpt-201, spring 83).

offers (math, math-152, spring 83).
offers (math, math-101, fall 82).
offers (math, math-202, spring83).
offers (math, math-303, fall83).
offers (math, math-606, fall84).
offers (math, math-707, summer83).
offers (math, math-808, spring84).

offers (phil, phil-101, fall83).
offers (phil, phil-808, spring84).

teaches (caplin, cmpt-103, spring84).
teaches (caplin, cmpt-103, summer84).
teaches (caplin, cmpt-103, fall84).

teaches (hadley, cmpt-118, spring84).
teaches (hadley, cmpt-118, summer84).
teaches (hadley, cmpt-118, fall84).

teaches (peters, cmpt-205, spring84).
teaches (hell, cmpt-205, summer84).
teaches (liestman, cmpt-205, fall84).

teaches (peters, cmpt-405, spring84).
teaches (peters, cmpt-405, summer84).
teaches (liestman, cmpt-405, fall84).

teaches (tunt, cmpt-410, spring84).
teaches (dahl, cmpt-410, summer84).
teaches (dahl, cmpt-410, fall84).

teaches (ullman, cmpt-405, fall882).
teaches (minsky, cmpt-410, fall883).
teaches (codd, cmpt-354, spring83).
teaches (liu, cmpt-205, fall883).
teaches (liu, cmpt-105, spring83).
teaches (ullman, cmpt-201, summer883).

teaches (beauvoir, phil-101, fall883).
teaches (beauvoir, phil-808, spring884).
teaches (occam, phil-101, fall883).
teaches (occam, phil-808, spring884).
teaches (spinoza, phil-101, fall883).
teaches (spinoza, phil-808, spring884).

teaches (euler, math-101, fall882).
teaches (euler, math-152, spring883).
teaches (gauss, math-303, fall883).
teaches (fermat, math-202, spring883).
teaches (euler, math-606, fall884).
teaches (fermat, math-707, summer883).
teaches (gauss, math-308, spring884).

The assertions below state the current year and semester.  
Also, the range of years which the database knows about is
stored under the predicate viableyears (viable years). This information must be updated each semester, since it cannot be easily computed. Prolog has no built-in date function.

current(semester, fall) :- !.
current(year, 83).
viableyears(82/93/84/nil).

{Here is some other information which the program uses when analysing temporal phrases. This information does not have to be updated.}

season(spring).
season(summer).
season(fall).

year(Y) :- viableyears(L), anymemb(X,L).
cycle(season, spring/summer/fall/nil).

ordertype(semester, indexed-cycle).
ordertype(year, sequential).
ordertype(season, indexed-cycle).

{The tilda operator is an infix operator which is used to bind together seasons and years. The reason for this is that time entries in the database are in terms of season (semester) - year pairs.}

S?Y :- season(S), year(Y).

{The procedure below is a high level one which answers a user's question. Once a user's question has been fetched by GET-QUEST and put in the form of a list, the list can be passed to the ANS procedure to obtain an answer to the question. For example, when the question "Which professor teaches cmpt-405", has been read, by get-ques, a list is returned having the form "which/professor/teaches/cmpt-405/nil". This list is then passed to the ANS procedure.}

ans(In) :- clean-slate,
sentence(In, nil), !,
testonce(parsed), abolish(parsed, 0), nl.

{The second and third definition of ans, shown below, is used when parsing of the sentence succeeds, but the answer could not be found because of false presuppositions. The fourth and fifth rules below cover the case where the parsing of the sentence also failed.}

ans(In) :- message(Type,Mess,Action), nl, mywrite(Mess), nl,
nl, abolish(message, 3), Action, !, abolish(parsed, 0).

ans(In) := or(testonce(parsed),wierdone),
        abolish(parsed, 0) , ! , nl,
        deviant, abolish(deviant, 0),
        abolish(wierdone, 0),
        mywrite('perhaps you would care to rephrase your question'),
        nl.

ans(In) := unknown, abolish(unknown, 0), !.

ans(In) :- nl, mywrite(
        'your sentence contains words or syntax which are unknown.'),
        mywrite('please check your spelling or rephrase.'),
        nl.

{ The CLEAN-SLATE predicate is used to initialize counts, and remove leftover flags from the processing of the previous sentence. }

clean-slate :- set-all-counts, abolish(global,2),
        abolish(message, 3),
        abolish(parsed, 0), abolish('/', 2), abolish(triedall, 1),
        abolish('deviant, 0 ), abolish(stack, 2), !.

{ The predicate below is the master predicate which is used to parse and analyse a sentence. The first rule shown below handles sentences where the verb is not split by a noun phrase. The second rule handles the case where it is split. }

sentence(Input, X5) :- ques-phrase(Input, X1, Qtype)
        , np(X1, X2, Oper1, Var1, Intens1, Qtype, Type1, Neg)
        , fancy-verb(X2, X3, Vb-rel, Voice, Vtense, Auxtense, Neg)
        , np(X3, X4, Oper2, Var2, Intens2, nil, Type2, Neg)
        , adverbial(X4, X5, Auxtense, Vtense, Var3, Intens3, Type3,
                Var-inside, Oper3)
        , compat(Type1/Type2/Type3/nil, Var1/Var2/Var3/nil,
               Voice, Vb-rel, Useargs, First-type)
        , make-relation(Vb-rel, Useargs, Relation)
        , assert(parsed)
        , make-qry(Oper1, Var1, Intens1, First-type, Oper2,
                  Var2, Intens2, Oper3, Var-inside,
                  Intens3, Neg, Relation, Query)
        , eval(Query).

{ The sentence form recognized by the predicate below is a context sensitive form, namely, sentences of the form
The procedure below constructs a relation using the relation name, and a list of appropriate variables which has been returned from the Compat procedure. The relation name and list of variables are composed into a Relation. The general form for the procedure is make-relation( Vb-rel, Useargs, Relation )

make-relation( Vb-rel, Useargs, Relation ) :- mylist( Useargs, Sqlist ) ,
  cons( [ Vb-ref | Sqlist ], Relation ) .

The prototype for the MAKE-QRY PROCEDURE is make-qry(Oper1, Var1, Intens1, First-type, Oper2, Var2, Intens2, Oper3, Var-inside, Intens3, Neg, Relation, Query). This procedure is used to make a formal query which must still be interpreted by other prolog procedures in this program.

make-qry( Oper1, Var1, Intens1, First-type, Oper2, Var2, Intens2, Oper3, Var-inside, Intens3, Neg, Relation, Query ) :-
  (var(First-type)
  -> fail

  (or(nonvar(Neg), equal(Oper2, no))
  -> (and(equal(First-type, entity), global(implied, human)))

  -> cons([human, Var1], Hold)
  ; cons([First-type, Var1], Hold)), equal(Onceonly, yes).

make-qry( Oper1, Var1, Hold/nil, Unbound, Oper2, Var2, Intens2, Oper3, Var-inside, Intens3, Neg, Relation, Query )
; fail ) ), nonvar (Onceonly), ! .

make-qry( some, Var1, Intens1, First-type, Oper2, Var2, Intens2, Oper3, Var-inside, Intens3, Neg, Relation, Query ) :-
  var (Neg), not (equal (Oper2, no)),
equal (Temp, apply-oper ( Var2, Var1, Relation, Intens1, Oper2)),
!, make-pred (Var1, Temp, Oper3, Var-inside, Intens3, Neg, Predication, yes, Var2, Intens2, Oper2),
equal (Query, apply (Oper2, Var2, Intens2, Var1, Predication)).

make-qry( Oper1, Var1, Intens1, First-type, Oper2, Var2, Intens2, Oper3, Var-inside, Intens3, Neg, Relation, Query ) :-
  partial-intens (Var1, Var2, Relation, Intens2, Oper2, Neg, Done, Temp, Var-inside, Intens3, Oper3 ),
!, make-pred (Var1, Temp, Oper3, Var-inside, Intens3, Neg, Predication, Done, Var2, Intens2, Oper2),
equal (Query, apply (Oper1, Var1, Intens1, Var2, Predication)).

{ The EVAL procedure is used to see whether a query should be instantiated or not before it is invoked. A flag is checked to see whether any 'some-particular' operators are present in the query. }

eval (Query) :-
  retract ( global (sub, deviant) ), fail.

eval (Query) :-
  not ( global ( partic, X) ), !, Query .

eval (Query) :-
  global ( partic, X), !,
instantiate ( Query, Q), f.

{ The Instantiate procedure, shown below, is used to deal with queries which contain, at some level of nesting, the operator "some-particular". Sub-procedure invocations in the query which contain this word are instantiated in the sense that if the sub procedure is asking us to find some particular x which has a certain property, a value for that x is found at this stage, by the present procedure, and substituted for x. This is done at every level of nesting. The procedure is written so that backing up is possible at every level of nesting so that the procedure can supply other 'particular x's ' should the need arise. }

instantiate ( apply (Op1, V1, Int1, V2, Predication), Newquery ) :-
  not ( equal (Op1, some-particular ) ),
instantiate ( Int1, Newint ),
instantiate ( Predication, Newpred ),
equal ( Newquery, apply ( Op1, V1, Newint, V2, Newpred)).
instantiate( apply(some-particular, V1, Int1, V2, Predication),
    apply( some, V1, Newint, V2, Newpred ) )
  :-  instantiate(Int1, Newint),
      instantiate( Predication, Newpred ).

instantiate( apply-oper(Var, Var2, Relation,
    Intens2, some-particular), Result)
  :-  instantiate( Intens2, Newint),
      truth-cond( Var2, Newint),
      instantiate( Relation, Newrel),
      equal( Result, apply-oper(Var, Var2, Newrel, true, some-particular )).

instantiate( apply-oper( Var, Var2, Relation, Intens2, Oper2 ),
    Result )
  :-  not( equal(Oper2, some-particular )),
      instantiate( Relation, Newrel),
      instantiate( Intens2, Newint ),
      equal( Result, apply-oper(Var, Var2, Newrel,
        Newint, Oper2 )).

instantiate( nil, nil ) :- !.

instantiate( mynot(P), mynot(Q) ) :- instantiate( P, Q ).

instantiate( F/Rest, F1/Rest1 )
  :-  instantiate( F, F1 ),
      instantiate( Rest, Rest1 ) .

instantiate( F, F)
  :-  not( equal( F, A/B ) ),
      not( equal( F, mynot( Any ))),
      not( equal( F, apply( A1, A2, A3, A4, A5))),
      not( equal( F, apply-oper( B1, B2, B3, B4, B5 ))).

{ The predicate below has the general form ques-phrase(Inlist,
  Outlist, Qtype) and it examines the beginning of a sentence to
determine what the question type is. The question-type is output
via the third parameter of the pred. }

 ques-phrase( does/Rest, Rest, true-that) :- !.
 ques-phrase( which/Rest, Rest, which) :- !.
 ques-phrase( how/many/Rest, Rest, how-many) :- !.
 ques-phrase( what/are/the/Rest, Rest, what-are-the).
 ques-phrase( who/are/the/Rest, Rest, who-are-the).
 ques-phrase( what/is/the/Rest, Rest, what-is-the).
 ques-phrase( who/is/the/Rest, Rest, who-is-the).
 ques-phrase( is/it/true/that/Rest, Rest, true-that) :- !.
 ques-phrase( what/Rest, Rest, which) :- not(parsed).
 ques-phrase( what/Rest, Rest, what-are-the ) :- not(parsed), !.
 ques-phrase( is/Rest, Rest, true-that ) :- !.
 ques-phrase( who/Rest, Rest, who-are-the )
  :- not(parsed),

174
assert( global( implied, human)), !.

{ The predicate 'get-type' shown below, will find the type of a proper noun only. The predicate has the general form "get-type( noun, type)". }

get-type(X, prof) :- member(X, D).
get-type(X, dept) :- dept(X).
get-type(X, course) :- offers(D, X, S), !.
get-type(X, yr) :- year(X).

{ The get-type predicate below is used only to recognize course names which do not appear in any relation in the data base. The name is dissected to see if it has the general form of a course name. If so, an error message is printed since the course is unknown. }

get-type(X, course) :- not(unknown),
    name(X, [ A, B, C, D, Dash, One, Two, Three ])
    , name(Name, [ A, B, C, D ])
    , name(Num, [ One, Two, Three ])
    , dept(Name)
    , nl
    , integer(Num)
    , mywrite('this course is not offered by any department')
    , mywrite(X), nl, !, assert(unknown), fail.

{ The np predicate below is used to process a complex np which may contain determiners, adjectives, certainly a noun and possibly one or more relative clauses. The relative clauses and adjectives are turned into restrictions. The relative clause restriction takes the form of a single predication, whereas the adjectives are turned into a list of restrictions. the internal predicate representing the main noun is also found. All these predicates are appended into a single list of the form pred1/pred2...../predn which is returned as the intension of the noun phrase. The np predicate also returns a value for its operator, which is a function of its determiner and the question type. Also the type of the main noun is returned and a variable name which appears in the returned intension is also returned. }

np(Ix, Ox, Operator, Var, Intens, Qtype, Type, Meg)
    :- jet(Ix, X1, Oper, Qtype)
        , adj2(X1, X2, Adj-list), first(X2, Mword)
        , noun(X2, X3, Main-form, Var, Oper, Qtype, Type)
        , select-adjs(Type, Adj-list, Tc-list)
        , rel-cl(X3, Ox, Var, Intens1, Type)
        , append( Intens1, Tc-list, Hold)
The np predicate immed. below will check whether the first element of its input list is a proper noun by checking whether it is listed as having a type under the pred. "get-type".

np(N/R, R, the, S, true/nil, QType, Type, Nword) :-
    get-syn(N, S), get-type(S, Type),
    assert(global(N/R, done)).

The np predicate shown below is used to process noun phrases which contain prepositional phrases of a certain type. Namely, the prep-phrase must either be preceded by a noun which implies some deleted phrases, such as 'prerequisite' implies "which is required by", or the preposition itself must imply a phrase, such as 'from' may sometimes imply "which comes from".

np(Ix, Ox, Operator, Var, Intens, QType, Type, Nword) :-
    simple-np(Ix, X1, Op, Var, Subintens, QType, Type, Nword),
    prep(X1, X2, Prep),
    phraseimpl(Nword, Phrase, Prep), !,
    append(Phrase, X2, X3),
    special-rel-cl(X3, Ox, Var, Intens1, Type, Op, Oper),
    append(Intens1, Subintens, Intens),
    case-some(Oper, Intens, Ix, Nword, Var, Operator, Nword),
    assert(global(Ix, Intens)).

The NP rule shown below will process prepositional phrases which contain prepositions which are vague, in that they do not clearly indicate what the relation between the bordering nouns should be. Also the nouns do not by themselves indicate what the relation should be, as is the case with 'prerequisite'. Therefore, the rule below examines the types of both nouns which border the preposition, in order to find a relation in the database which involves both of the nouns. If such a relation can be found a predicate involving the relation name and the variables representing the nouns is constructed. If the wrong relation was found, others can be found through backtracking.

np(Ix, Ox, Operator, Var, Intens, QType, Type, Nword) :-
    simple-np(Ix, X1, Op, Var, Subintens, QType, Type, Nword),
    prep(X1, X2, Prep),
    np(X2, Ox, Op2, Var2, Intens2, nil, Type2, Nword),
    find-rel(Type/Type2/nil, Relation),
    transform(Type/Var/Type2/Var2/nil, Argtypes),
    set-oper(Op, Op2, Oper, Oper2),
    mylist(Argtypes, Args),
    cons([Relation | Args], Relation),
    equal(Intens3, apply-oper(Var, Var2),

append(Hold, MAIN-form/nil, Intens)
    case-some(Oper, Intens, Ix, Nword, Var, Operator, Nword)
    assert(global(Ix, Intens)).
append( Intens3/nil, Subintens, Intens ),
case-some( Oper, Intens, Ix, Nword, Var, Operator, Neg ),
assert( global(Ix, Intens )).

{ The NP rule below will be tried if all other NP rules fail. 
This rule consumes no input, but creates a dummy noun phrase for 
those cases where the NP is implicit, as in "what computing 
courses are being offered ? " }

np( Ix, Ix, Op, V, true/nil, Qtype, dummy, Neg )
  :- not(parsed),
      not( global( Ix, Any ) ),
      ( var(Qtype) -> equal(Op, some) )
      ;
      ( or( equal(Qtype, who-are-the),
           equal( Qtype, what-are-the ) )
        -> equal( Op, find-all )
        ; (equal(Qtype, nil)
           -> equal(Op, some)
           ; fail )).

{ The SIMPLE-NP predicate shown below is used to parse simple 
noun phrases which have the form, det adj* noun This predicate 
returns all the type of information returned by the full NP 
predicate, except that prepositional phrases and relative 
clauses are omitted. Also, the actual word for the principal 
noun in the phrase is returned in Nword. }

simple-np( Ix, Out, Oper, Var, Intens, Qtype, Type, Nword )
  :- det( Ix, X1, Oper, Qtype),
     adj2( X1, X2, Adj-list ), first(X2, Nword),
     noun( X2, Out, Main-form, Var, Oper, Qtype, Type),
     select-adj( Type, Adj-list, Tc-list ),
     append( Tc-list, Main-form/nil, Intens ).

{ The next version of SIMPLE-NP is used to parse those NPs which 
are simply proper nouns . }

simple-np( N/R, R, the, S, true/nil, Qtype, Type, S )
  :- get-syn( N, S ), get-type( S, Type ).

{ The FIND-REL procedure, below, will take a list of noun types, 
and find a relation name which involves those types (it is 
assumed that only two types are being passed, although the 
relation may itself be n-ary ). The relation name is returned 
via the second parameter, and a list of the arguments required 
by the relation is returned via the third parameter. The list 
which is returned will contain distinct variables where the 
required types were not found among the input list of types. For 
example, if the input list was course/prof-nil and the types 
required by the relation found is course/prof/semester-nil, then
the list returned will look like course/prof/VAR,nil.

find-rel(Typlist, Rename, Argtypes) :-
  argtype(Rename, Arglist),
  reorders(Typlist, Ordering),
  checkorder(Ordering, Arglist),
  fill-positions(Ordering, Arglist, Argtypes),!.

find-rel(X/X/nill, equal, X/X/nill).

{ The REORDERS predicate, below, can find two different

orderings for a list of two elements. }

reorders(Typlist, Typlist).
reorders(F/S/Rest, S/F/Rest).

{ The FILL-POSITION procedure, below, does a pattern match on

the top-level elements in the lists given as its first and

second arguments. It will fill positions in its first argument

with 'Vjunk' where the first list does not match the second.

"Vjunk" will match anything at a later stage, but here it is

used to expand the input Ordering, which contains a subset of

the elements in Arglist, until both lists are of equal length

and every element in Ordering matches the type in the

corresponding position in Arglist. The padded result is

returned. The prototype is "fill-positions(Ordering, Arglist,

Paddedresult)". }

fill-positions(nil, nil, nil) :- !.

fill-positions(nil, F/Rest, Vjunk/Result) :- !, fill-positions(nil, Rest, Result).

fill-positions(F1/Rest1, F1/Rest2, F1/Newrest) :-
  !, fill-positions(Rest1, Rest2, Newrest).

fill-positions(dummy/Rest1, F2/Rest2, dummy/Newrest)
  :- !, fill-positions(Rest1, Rest2, Newrest).

fill-positions(F1/Rest1, F2/Rest2, Vjunk/Newrest)
  :- fill-positions(F1/Rest1, Rest2, Newrest).

{ The procedure below will replace types in a list of argument

types by their corresponding variables. The correspondence

between types and variables is provided by the first argument

which is a list of the form Type1/Var1/.... TypeN/VarN/nill. }

transform(nil, Argtypes, Argtypes) :- !.

transform(T1/V1/Rest, Argtypes, Resultlist) :-
  replace1(T1, V1, Argtypes, Rest),
  transform(Rest, Rest, Resultlist).
The REPLACE1 rule, below, will replace the FIRST occurrence of its first argument, in the list which is provided as its third argument, by its second argument. The result is returned in Result. No variables in the original list will be altered.

replace1( Ty, Var, pil, nil ) :- !.
replace1( Ty, Var, C/Rest, Var/Rest ) :- nonvar(C), equal(C, Ty), !.
replace1(Ty, Var, X/Rest, X/Newrest ) :- replace1( Ty, Var, Rest, Newrest ).

The predicate rule given below will take a list of internally recognized adjectives, along with a noun type, and return a list of predications of the form, bind( type, adj), where each adjective in the input list gets placed in one such pair.

select-adjs(Ty, nil, nil) :- !.
select-adjs(Ty, Ad/Rest, We/Rest) :- cons([ bind,Ty,Ad ], New ), select-adjs(Ty, Rest, Newrest ).

The series of det(determiner) axioms below are used to find the proper determiner of a noun. When the noun follows immediately after a ques-phrase the noun may not have a determiner. In this case the question phrase may be examined to tell what the determiner should be. In some cases the decision must be postponed until the noun itself is found. Then the decision is made by examining the noun and the question type together. The determiner is returned under the identification of Oper, since in this program the determiner is treated as an operator.

det(Ix, Ox, Oper, Qtype) :- or( equal(Qtype, true-that), equal(Qtype, nil) ), find-det(Ix, Ox, Oper).
det(Ix, Ix, count-all, how-many) :- !.
det(Ix, Ix, find-all, what-are-the) :- !.
det(Ix, Ix, find-all, who-are-the) :- !.
det(Ix, Ix, Oper, which) :- !.
det(Ix, Ix, find-the, who-is-the) :- !.
det(Ix, Ix, find-the, what-is-the) :- !.

The find-det predicate examines the determiner which begins
its input list and it decides what the operator should be. If the determiner is 'some' or 'a', then the operator returned is 'some'. 'every' returns 'every' and 'the' returns a variable, since the meaning of 'the' depends upon whether it precedes a singular or plural noun.

\[
\text{find-det(some/R, R, some) : - } !.
\]

\[
\text{find-det(a/R, R, some) : - } !.
\]

\[
\text{find-det(every/R, R, every) : - } !.
\]

\[
\text{find-det(the/R, R, Oper)} : - !.
\]

\[
\text{find-det(no/R, R, no) : !.}
\]

\[
\text{adj2(~/1, Ox, Pred-list) :- get-syn(A, X), get-adj(X), equal(Pred-list, X/List), adj2(Y, Ox, List).}
\]

\[
\text{adj2(In, In, nil) : - first(In, F), not(get-adj(F)).}
\]

\[
\text{get-syn(One, S) : - or(syn(One, S), equal(One, S)) .}
\]

\[
\text{noun(M/R, R, Main-form, Var, Oper, Qtype, Type) : - get-noun(M,Num,Main-form,Type,Var), equal(Qtype, which), set-op(Num, Oper).}
\]

\[
\text{noun(M/R, R, Main-form, Var, Oper, Qtype, Type) : - get-noun(M,Num,Main-form,Type,Var), not(equal(Qtype, which)), var(Oper), the-op(Num, Oper).}
\]
noun(W/R, R, Main-form, Var, Oper, Qtype, Type)  
  :-  get-noun(W, Num, Main-form, Type, Var),  
    not(equal(Qtype, which))  .

{ The set-op and the-op predicates determine an appropriate operator for question types 'which' and 'the' respectively. The decisions depends upon the number of the noun. }

set-op(sq, find-the).
set-op(pl, find-all).

the-op(sq, the).
the-op(pl, some) :- cur-relation(equal, X), !.
the-op(pl, every).

{ The COMPAT procedure takes as input a list or types and variables which represent the entities which have been found during the semantical analysis process. The Voice of the current clause and the Relation name of the main verb in the clause are also required. The voice is consulted to see whether the first two types and variables in Typlist and Varlist need to be reversed. Once the proper order has been determined, the types are checked to see if they are compatible with those required by the verb relation. If so, a list of ordered variables is returned which is appropriate to use with the verb relation. The variables are ordered in correspondence with the proper list of Types which has been discovered. }

compat(Typlist, Varlist, Voice, Relname, Useargs, First-type)  
  :-  nonvar(Voice),  
       memb( dummy, Typlist), !,  
       compat( Typlist, Varlist, Unbound,  
                  Relname, Useargs, First-type ).

compat(Typlist, Varlist, active, Relname, Useargs, First-type)  
  :-  blend(Typlist, Varlist, Corresp),  
       check-and-expand(Typlist, Relname, Usetypes),  
       transform(Corresp, Usetypes, Useargs),  
       (equal(Typlist, dummy/Rest )  
         -> first(Varlist, V1), var-pos(V1, Useargs, W),  
         argtype(Relname, Realtypes), find-nth(Realtypes, W ,  
                  First-type)
       ;  true ) , !.

compat(T1/T2/Rest, V1/V2/Rest2, passive, Relname,  
       Useargs , First-type)  
  :-  blend(T2/T1/Rest, V2/V1/Rest2, Corresp),  
       check-and-expand( T2/T1/Rest, Relname, Usetypes ),  
       transform(Corresp, Usetypes, Useargs),
(equal( T1, dummy) 
  -> var-pos(V1, Useargs, W),  
  argtype(Relname, Realtypes), find-nth(Realtypes,  
  W, First-type )  
  ; true ), !.

compat( T1/T2/Rest, V1/V2/Rest2,  
  Voice, equal, V1/V2,nil, First-type ) :-  
  or(equal(T1,T2), or(equal(T1,dummy),equal(T2,dummy)) ),  
  elim-nils( Rest, nil), (equal( T1, dummy) -> equal(First-type, entity) ; true ), !.

compat( Types, Vars, Voice, R, Useargs, First-type ) :-  
  assert( message( compat,  
    'There appears to be a semantic  
    or physical impossibility implied in 
    your question',  
    weird )), fail.

weird :-  
  not(parsed), assert( deviant), assert( weirddone ), fail.

{ The BLEND procedure is used to interweave the elements of two  
  lists of the same length. E.g., blending the lists a/b/nil and  
  w/q/nil, gives the result, a/w/b/q/nil . } 
blend(nil/R1, nil/R2, R3) :-  
  blend(R1,R2,R3), !.

blend(T/R1, V/R2, T/V/R3) :-  
  blend( R1, R2, R3), !.

blend(nil, nil, nil).

{ The procedure below is used to check whether the types passed  
  in typlist are an ordered subset of the types required by the  
  Relation passed as the second argument. If so, and if the  
  typlist is a PROPER subset of the set required, then the list is  
  padded with variables in appropriate places, and returned as  
  Usetypes. } 

check-and-expand( Typlist, Relname, Usetypes) :-  
  argtype(Relname, Arglist), checker(Typlist, Arglist),  
  elim-nils( Typlist, Shortlist),  
  fill-positions( Shortlist, Arglist, Usetypes ).

checker( nil, nil ) :- !.

checker(L, nil ) :- !, elim-nils(L, nil ).

checker( Dum/Suspects, F/Rest ) :-  
  or( equal(Dum, nil), equal(Dum, dummy) ),  
  checker( Suspects, Rest), !.

checker( F/Suspects, F/Rest)
:- checker( Suspects, Rest ).

{ The procedure below will eliminate all the nil elements in a list }

elim-nils(nil, nil) :- !.

elim-nils(nil/Rest, Newrest) :- elim-nils(Rest, Newrest), !.

elim-nils(F/Rest, F/Newrest) :- elim-nils(Rest, Newrest).

{ The CHECKORDER procedure, below, is used to check that the order of elements in the list provided as the first argument is the same as the order of elements in the list provided as the second argument. The first argument must be a subset (possibly proper) of the second argument, or the procedure will return a fail. }

checkorder(nil, L) :- !.

checkorder(L, nil) :- !, fail.

checkorder(F/Suspects, Knownlist) :- find(F, Knownlist, Remainder), checkorder(Suspects, Remainder).

{ The FIND procedure, below, will find whether its first argument is included in the list which is the second argument. If so, the remainder of the list following the found element is returned via the third parameter. Otherwise, the proc. returns a fail. }

find(nil, L, L) :- !.

find(F, F/Rest, Rest) :- !.

find(F, X/Rest, Rm) :- find(F, Rest, Rm).

{ The assertions shown below provide, for each relation in the relational database, a list of types of the arguments which are required by the relation. The order of the required argument types is specified since this is important for most relations. }

argtype(offers, dept/course/semest/nid).
argtype(teaches, prof/course/semest/nid).
argtype(prereq, course/course/nid).
argtype(member, prof/dept/nid).
argtype('E', sea/yr/nid).
The VAR-POS procedure is used to find the position of variable V in the list which is provided as the second argument. The ordinal position is returned via the third arg.:

```
var-pos(V, nil, 0) :- !.
var-pos(V, F/R, 1) :- V =\= F, !.
var-pos(V, F/R, M) :- var-pos(V, R, X), M is X+1.
```

The procedure below returns the nth element in a list. Its general form is "find-nth( List, N, Result)".

```
find-nth(List, 0, nil) :- !.
find-nth( F/Rest, 1, F) :- !.
find-nth( F/Rest, M, Result) :- X is M-1, find-nth(Rest, X, Result).
```

The rel-cl predicate, defined below, is used to process relative clauses which begin with 'that', 'who', and 'which'. It returns a value, 'intens1' which is set to a predicate involving a verb form which must be true if the relative clause is satisfied. If no relative clause is present, this intensional parameter is set to 'true' so that is will always succeed.

```
rel-cl(Ix, Ox, Var, Predic,nil, Type) :-
  or( or( equal(Ix, which/Y), equal(Ix, who/Y)),
      equal(Ix, that/Y)),
  fancy-verb(Y, Y1, Vb-rel, Voice, Vtense, Auctense, Neg),
  np(Y1, Y2, Oper2, Var2, Intens2, Qtype, Type2, Neg),
  adverbial(Y2, Ox, Auctense, Vtense, Var3, Intens3, Type3, War-inside, Oper3),
  compat(Type/Type2/Type3/nil, Var/Var2/Var3/nil, Voice, Vb-rel, Useargs, First-type),
  make-relation(Vb-rel, Useargs, Relation),
  partial-intens(Var, Var2, Relation, Intens2, Oper2, Neg, Done, Temp, War-inside, Intens3, Oper3),
  make-pred(Var, Temp, Oper3, Var-inside, Intens3, Neg, Predic, Done, Var2, Intens2, Oper2).
```

The special-rel-clause predicate shown below is used to process relative clauses which have been constructed in the process of parsing certain noun phrases which should contain certain 'filtered' phrases. When the constructed phrases is used to replace a preposition such as 'for' or 'of' it sometimes happens that the following quantifier interacts in subtle ways with the quantifier which precedes the noun which precedes the preposition. The Set-oper procedure is invoked to find what the
appropriate operator should be when these interactions occur.

special-rel-cl( In, Out, Var, Predic/nl, Type, Op, Oper )
   :- or( or( equal( In, which/Y), equal( In, who/Y ) ),
         equal( In, that/Y ) ),
   fancy-verb(Y, Y1, Vb-rel, Voice, Vtense, Auxtense, Neg ),
   np( Y1, Y2, Oper2, Var2, Intens2, Qtype, Type2, Neg ),
   set-oper( Op, Oper2, Oper, Newop2 ),
   adverbial( Y2, Out, Auxtense, Vtense, Var3, Intens3,
            Type3, Var-inside, Oper3 ),
   compat( Type/Type2/Type3-nil, Var/Var2/Var3-nil, 
           Voice, Vb-rel, Useargs, First-type ),
   make-relation( Vb-rel, Useargs, Relation ),
   partial-intens( Var, Var2, Relation, Intens2, Newop2, 
                  Neg, Done, Temp, 
                  Var-inside, Intens3, Oper3 ),
   make-pred( Var, Temp, Oper3, Var-inside, 
              Intens3, Neg, Predic, Done, 
              Var2, Intens2, Newop2 ).

{ The SET-OPER procedure, below, is used to pick the proper
interpretation for the operator which precedes a noun which
follows a preposition in certain prepositional phrases which
have been translated into special relative clauses. For example,
if we ask "What is the prerequisite for every math course?"
the word 'every' is a signal that we are looking for every
prerequisite which is a prerequisite for some math course.
This case is covered by the first implication shown below. On the
other hand, if we ask, "What is the prerequisite for some math
course?", we want the operators to be interpreted as they
naturally would be, and this case is covered by the fifth rule.

set-oper( find-the, Oper2, find-all, some )
   :- or( equal( Oper2, every), equal( Oper2, all ) ) !.

set-oper( the, Oper2, every, some )
   :- or( equal( Oper2, every), equal(Oper2, all) ), !.

set-oper( some, every, X, Y )
   :- assert( message(oper, 'This query is too complex
               for us at present', true)),
       nl, !, assert(parsed), fail.

set-oper( Op, every, Op, some ) :- !.

set-oper( Op, Oper2, Op, Oper2) .

{ The axiom below requires that several conditions be tested
which are already tested in the version just above. We must
eliminate these possibilities by using the negation operator
since using the cut operator would prevent backtracking which
may be required for semantic agreement. }
rel-cl(Ix, Ix, Var, nil, Type) :- not(equal(Ix, that/Y)), not(equal(Ix, who/Y)), not(equal(Ix, which/Y)).

{ The CASE-SOME procedure shown below is invoked at the end of each NP procedure and is used to check whether the operator which has been found for the main noun is 'some'. If so, the user is queried about the intended meaning for 'some', and the operator may be changed to some-particular, depending upon the user's answer. }

case-some( some, Intens, F/Rest, Nword, Var, some, Neg) :- or(equal(F, the), and(equal(F, a), var(Neg))) !.

case-some( Oper, Intens, Ix, Nword, Var, Oper, Neg ) :- not(equal(Oper, some)), !.

case-some( some, Intens, Ix, Nword, Var, some, Neg ) :- global( other, Ix ), !.

case-some( some, Intens, Ix, Nword, Var, some-particular, Neg) :- global( partic, Ix ), !.

case-some( some, Intens, Ix, Nword, Var, no, Neg) :- global( none, Ix ), !.

case-some( some, Intens, Ix, Nword, Var, Newoper , Neg) :- ask(Ix, Nword, Neg), repeat, get-answer(Answer, Neg), !, choose-flag(Answer, Ix, Intens, Var, Nword), case-some(some, Intens, Ix, Nword, Var, Newoper, Neg), !.

nonexist( Answer, Ix, Intens, Var, Nword ) :- nl, mywrite('There is a false presupposition concerning the existence'), mywrite('of a '), write(Nword), mywrite('which satisfies this phrase:') , nl, nl, printlist(Ix), !, assert(parsed), fail.

{ The ASK procedure is used to ask the user to disambiguate expressions which contain the quantifier 'some' or some other quantifier which has been translated into 'some'. }

ask( Ix, Nword, Neg) :- nl, mywrite('In the following phrase, which concludes your question:') , nl, nl, nl, printlist(Ix), nl, nl, mywrite('The words at the beginning are ambiguous.'), nl, mywrite('Do you mean some particular '), write(Nword), write('?'), nl,
mywrite('If so, just type the numeral 1'), nl,
mywrite('Or do you mean just some '),
write(Nword), write(' or other?'), nl,
mywrite('In this second case just type the numeral 2'),
( nonvar(Neg) -> negask(Ix, Nword); nl ).

negask(Ix, Nword) :- nl,
mywrite('Or do you mean no '), write(Nword),
write(' at all?'), nl,
mywrite('In this last case just type the numeral 3'), nl.

printlist(nil) :- !.
printlist(F/Rest) :- write(F), write(' '), printlist(Rest).

{ The GET-ANSWER procedure is used to read input from the user when the user has been asked to disambiguate a 'some' type question. The procedure begins by consuming blanks and carriage returns until some other character is found. This character is inspected by the DESIRED rules to see if it is a one or a two. If not, DESIRED prints an error message. Otherwise, all characters up to the next carriage return are consumed and the desired character is returned. }

get-answer(N, Neg) :- consume-junk(A), name(N, [A]), !,
desired(N, Neg), repeat, get0(X), cr(X).

desired(1, Neg) :- !.
desired(2, Neg) :- !.
desired(3, Neg) :- nonvar(Neg), !.

desired(Y) :- nl,
mywrite('Sorry, but your answer is unclear.'),
nl, mywrite('Please just enter the number 1 or the number 2'),
mywrite('But first, please hit return.'),
repeat, get0(X), cr(X), !, fail.

{ The CHOOSE-FLAG procedure is used to decide whether to assert a flag saying that the 'some-particular' interpretation has been chosen for the current portion of the input string, or whether to assert a flag saying that 'some or other' is the desired interpretation for the current portion of the input. }

choose-flag(Answer, Ix, Intens, Var, Nword)
:-
(equal(Answer, 1) ->
  assert(global(partic, Ix))
; (equal(Answer, 2)
  -> assert(global(orother, Ix))
  ; assert(global(none, Ix)) ))
.

{ The EXISTS procedure, below, will check that there exists a value for Var which satisfies Intens, without binding the
variable. }
exists( Var, Intens) :-
    testonce( truth-cond(Var, Intens)),
    assert( existsflag ), fail.
exists( Var, Intens) :-
    retract( existsflag ).
testonce( P ) :-
    P, !.

{ The FANCY-VERB rule is used to parse entire verb formations, 
including passive forms where auxiliaries immediately precede 
the verb. It also consumes particles which come immediately 
after the verb if they are present. }
fancy-verb( In, Out, Vb-rel, Voice, Vtense, Auxtense, Neg )
    :- verb( In, X1, Vb-rel, Voice, Vtense, Auxtense, Neg ),
       particle( X1, Out ).
particle( to/Rest, Rest).
particle( In, In ).

{ The 'verb' predicate defined below will test whether the word 
or words heading its input list constitute a verb form 
recognized by the program. If so, an internal verb relation will 
be returned together with the name of a predicate to be applied 
if the verb is active or passive. }

{ This first verb rule covers the case where an active verb is 
present with no auxilliary. }
verb(V/R, Rest, Vb-rel, active, Vtense, Auxtense, Neg) 
    :- get-verb(V,Vtense,Vb-rel,Num)
        , not( equal(R, by/W) )
        , not( verb(R, Out, Rel, Voice, Vtense, Auxtense, Neg) )
        , ( first(R, not) 
            -> rest(R, Rest), equal( Neg, not) 
            ; equal(R, Rest) )
        , store( Vb-rel, Vtense ).
verb( In, Out, Vb-rel, passive, Vtense, Auxtense, Neg )
    :- aux(In, V/by/Out, Auxtense, Neg),
       get-verb(V,past,Vb-rel,Num),
       store( Vb-rel, Vtense ).

{ The verb rule shown above recognises passive verb forms, and 
the one below deals with active verbs which are preceded by 
auxiliaries. }
verb( In, Out, Vb-rel, active, Vtense, Auxtense, Neg)
    :- first(In, F) , get-aux(F, Auxtense, Neg)
        , aux( In, V/Out, Auxtense, Neg) , not( equal(Out, by/Y) )
, get-verb(V, Vtense, Vb-rel, Num), !
, store( Vb-rel, Vtense ).

verb(In, Out, Vb-rel, active, Vtense, Auxtense, Neg)
    :- aux(In, Out, Auxtense, Neg),
       not(equal(In, Out)), first(Out, F),
       not(get-verb(F, T, R, M)),
       precedes(Out, In, V),
       get-verb(V, Vtense, Vb-rel, Num),
       store( Vb-rel, Vtense ).

{ The procedure below will return the element which precedes
the list L1 in the list L2, where the general form for invoking
the procedure is "precedes(L1, L2, Element) ". }

precedes(L1, M/L1, M) :- !.

precedes(L1, F/R, M) :- precedes(L1, R, M).

{ The predicates below will consume a series of auxiliaries
which precede a verb. If none are present the input list is
returned as the output list. }

aux(W/Y, Out, Auxtense, Neg)
    :- get-aux(W, Auxtense, Neg),
       aux(Y, Out, Auxtense, Neg), !.

aux(In, In, Auxtense, Neg).

{ The ADVERBIAL rules below are used to process adverbial time
phrases such as "in the spring", "during the spring of 84",
"last fall", "during every semester of 83", "in the year of 84
in the spring ", "next semester", etc. The rules consult the
tenses of the auxiliary and main verbs in cases where the
article alone does not suffice to determine future, present, or
past reference. The canonical form for the rules is:

adverbial(In, Out, Auxtense, Vtense, Var, Intens, Type, Var-inside, Oper).

adverbial(In, Out, Auxtense, Vtense, Var, nil, semest, dummy, Op)
    :- time-prep( In, X1),
       determ( X1, X2, Det),
       time-func(X2, X3, Auxtense, Vtense, Det, Op),
       equal(X3, N/Out ),
       get-noun( N, Num, Intens, Type, Var ),
       det-time( Op, Var, Intens, Type ).

{ The rule just below is used to process those time phrases
which involve quantifiers such as "during every fall", or "in
some semester of 83", or "in some year", etc. }

adverbial( In, Out, Auxtense, Vtense,
Var, Intens, semest, Vl, Op)
:- time-prep( In, Xi),
   np( Xi, Out, Op, Vl, Int1, Qdummy, Type, Neg ),
   (memb(Type, sea/yr/nill)
    -> (and( integer(Vl), equal( Int1, true/nill ))
    -> equal(Var, X6V1), equal( Intens, nil)
    ; gather( Op, Int1, Var, Intens, Vl, Type ))
   ; fail ).

adverbial( In, In, Auxtense, Vtense, nil, nil, nil, dummy, Op ).

{ The TIME-PREP rules check whether a list begins with 'in' or 'during'. If not, no portion of the input list is consumed. }

time-prep( in/Rest, Rest ) :- !.
time-prep( during/Rest, Rest ) :- !.
time-prep( In, In ).

{ The DETERM rules check whether a list begins with 'the' or 'this', and if so, that determiner is returned as an operator. Otherwise, no input is consumed in the input list. }
determ( the/Rest, Rest, the ) :- !.
determ( this/Rest, Rest, this ) :- !.
determ( In, In, var ).

{ The TIME-FUNC rule is used to determine whether the operator for the temporal phrase should be 'last', 'this', or 'next'. If any of these determiners are actually present, they are used as the operator. Otherwise the auxiliary and main verb tenses are used to determine what the operator should be. }
time-func(In, Out, Auxtense, Vtense, Det, Op )
:- orderword( In, Out, Op ),
   (nonvar(Op)
    -> true
    ; ( equal(Det, this) -> equal(Op, Det )
    ; (nonvar(Auxtense) -> get-syn( Auxtense, Op )
    ; get-syn( Vtense, Op )))).

orderword( F/Rest, Rest, Op ) :-
   get-syn( F, Op ),
   memb( Op, last/next/this/nill ), !.

orderword( In, In, Op ).

{ The DET-TIME predicate requires as arguments the operator (this, last, next) which has been found, the intension which has been found for the noun in the adverbial phrase, such as 'fall', 'semester', 'year', and the type which has been found which should be 'season', 'semester', or 'year'. It will return as a
value for VAR something of the form Season&Year, where both 
Season and Year will be bound to constants like 'fall', and '84' . }

det-time( Op, Var, Intens, Type ) :-
  ordertype(Type, Order),
  cons([Order,Op,Var,Intens,Type], Pred),
  Pred.

{ The INDEXED-CYCLE rules are used to find a value for SEAS and 
YEAR, based upon the current operator, OP, and Intension which 
have been passed. The intension is assumed to have the form 
equal( X, S&Y ), where S and Y may or may not be bound. The last 
argument to indexed cycle should either be 'season' or 
'semester'. When the argument is 'season' it is assumed that the 
value for Seas is already embedded in intension. If the argument 
is 'semester' then a season must be chosen by inspecting the 
operator. }

indexed-cycle( Op, Seas&Year, Intens, season ) :-
  current( semester, S), equal( Intens, equal( X, Seas&Year ) ),
  current( year, Y ), cycle( season, Cyc ), find(S, Cyc, Rem ),
  ( equal(Op, this)
      -> ( equal( S, Seas)
          -> equal( Year, Y)
          ; ( memb( Seas, Rem)
              -> equal( Year, Y)
              ; Year is Y+1 ))
      ; ( equal(Op, next)
          -> ( memb( Seas, Rem)
              -> equal( Year, Y)
              ; Year is Y+1 )
      ; ( equal(Op, last)
          -> reverse(Cyc, Rem), find(S, Rev, Left),
              ( memb( Seas, Left) -> equal( Year, Y)
              ; Year is Y-1 )
      ; nl, write('Unknown time operator'), fail )).

indexed-cycle( Op, Var, Intens, semester ) :-
  current( semester, S), current( year, Y ),
  equal( Intens, equal( Seas&Year, Seas&Year ) ),
  ( equal( Op, this)
      -> equal( Seas, S), equal(Year, Y)
      ; ( equal( Op, next)
          -> find( S, spring/summer/fall/spring/nl, Rem ),
              first( Rem, Seas ),
              indexed-cycle( Op, Var, Intens, season )
      ; ( equal(Op, last)
          -> find( S, spring/fall/summer/spring/nl,Rem),
              first( Rem, Seas ),
              indexed-cycle( Op, Var, Intens, season )
      ; mywrite('Unknown time operator'), fail )).
The SEQUENTIAL rule below is used to find a value for YEAR based upon whether the operator is Next, This, or Last. The procedure has been designed to be rather general. Any variable whose value is determined sequentially could be bound to Year, and its appropriate ordered value would be returned.

\[\text{sequential}(\text{Op}, \text{Seas}\&\text{Year}, \text{Intens}, \text{year}) : - \]
\[\begin{align*}
\text{equal}(\text{Intens}, & \text{equal}(\text{Seas}\&\text{Year}, \text{Seas}\&\text{Year}), \text{current}(\text{year}, \text{Y}), \\
(\text{equal}(\text{Op}, \text{this}) & \rightarrow \text{equal}(\text{Year}, \text{Y}), \\
(\text{equal}(\text{Op}, \text{next}) & \rightarrow \text{Year is } Y+1, \\
(\text{equal}(\text{Op}, \text{last}) & \rightarrow \text{Year is } Y-1, \\
\text{mywrite(}'\text{Unknown time operator}'\text{), fail}))))\).
\end{align*}\]

\[\text{reverse}(\text{nil, nil}).\]
\[\text{reverse}(\text{F}/\text{Rest}, \text{Result}) : - \text{reverse}(\text{Rest}, \text{Rev}), \text{append}(\text{Rev}, \text{F}/\text{nil}, \text{Result}).\]

The GATHER procedure takes the operator and intension which have been discovered during the analysis of a temporal noun phrase such as 'during some semester of every year', and it attempts to find a season-year variable to represent the time which is described by the phrase. If all the information contained in the original intension is in a form which the program can easily evaluate, then the intension (Int1) is returned as the Return intension. In one case, however, when the operator is 'every', a more complex intension must be returned. The type (season or year) of the original intension must also be consulted.

\[\text{gather}(\text{Op}, \text{Int1}, \text{Var-pair}, \text{Ret-intens}, \text{Var-inside}, \text{Type}) : - \]
\[\begin{align*}
(\text{and}(\text{dissect}(\text{Int1}, \text{Var-pair}), \text{retract}(\text{found-dissect})) \\
\rightarrow (\text{or}(\text{equal}(\text{Op}, \text{the}), \text{equal}(\text{Op}, \text{some})) \\
\rightarrow \text{equal}(\text{Ret-intens}, \text{Int1}) \\
; \text{equal}(\text{Ret-intens}, \text{Int1}) \\
; (\text{equal}(\text{Type}, \text{sea}) \\
\rightarrow \text{and}(\text{equal}(\text{Var-pair}, \text{Var-inside}\&\text{Y}), \\
(\text{equal}(\text{Op}, \text{some-particular}) \\
\rightarrow \text{equal}(\text{Ret-intens}, \text{Int1}) \\
; (\text{equal}(\text{Op}, \text{every}) \rightarrow \\
\text{equal}(\text{Ret-intens}, \text{apply-oper}(\text{Var-inside}, \text{Y}, \\
\text{Var-inside}\&\text{Y}, \\
\text{year}(\text{Y})/\text{nil}, \text{every})/\text{Int1}) \\
; \text{equal}(\text{Ret-intens}, \text{Int1})))) \\
; \text{and}(\text{equal}(\text{Var-pair}, \text{S}\&\text{Var-inside}), \\
\text{equal}(\text{Ret-intens}, \text{Int1}))).)
\end{align*}\]

The DISSECT procedure below is used to dissect the elements of an intensional list of predications and find all relations.
contained therein which have the form $S\&Y$. If all such relations can be unified the unified result is returned. If no such relations are found the unbound pair $S\&Y$ is returned. If such relations are found but cannot be unified, the the procedure fails.

\[\text{dissect}(\text{apply-oper}(V, V2, S\&Y, \text{Int2}, \text{Op2}), S\&Y)\]
\[=: \text{dissect}(\text{Int2}, S\&Y), !, \text{assert(found-dissect)}.\]

\[\text{dissect}(\text{apply-oper}(V, V2, \text{Rel}, \text{Int2}, \text{Op2}), S\&Y)\]
\[=: \text{dissect}(\text{Rel}, S\&Y), \text{dissect}(\text{Int2}, S\&Y), !.\]

\[\text{dissect}(\text{nil}, V) :- !.\]

\[\text{dissect}(\text{F/Rest}, S\&Y)\]
\[=: \text{dissect}(\text{F}, S\&Y), \text{dissect}(\text{Rest}, S\&Y), !.\]

\[\text{dissect}(\text{Pred}, V).\]

{ The store predicate is used to store the most recently parsed verb, (rather its internal relation name is stored), and the tense is stored at the same time. }

\[\text{store}(Vb\text{-rel}, Vtense)\]
\[=: \text{abolish}(\text{cur\text{-}relation}, 2),\]
\[\text{assert}(\text{cur\text{-}relation}(Vb\text{-rel}, Vtense)).\]

{ The \texttt{MAKE-PRED} procedure, below, is used to construct the predicate of a top level sentence or of a relative clause. This procedure is passed information regarding the adverbial phrase which has been found, if any. If the adverbial phrase contained information which had to be represented by an intension (Intens3), then the partial predicate which has already been constructed and stored in Temp, is embedded in a quantified apply-oper phrase which quantifies over the adverbial intension. If an argument named Done is equal to yes, then that composed predicate is returned. Otherwise, the negation flag is checked to see if the intension should be negated before it is returned. }

\[\text{make-pred}(\text{Var}, \text{Temp}, \text{Oper3}, \text{Var\text{-}inside},\]
\[\text{Intens3}, \text{Neg}, \text{Predic}, \text{Done}, \text{Var2},\]
\[\text{Intens2}, \text{Oper2})\]
\[=: (\text{equal}(\text{Intens3}, \text{nil})\]
\[\rightarrow \text{equal}(\text{Intens}, \text{Temp})\]
\[; \text{equal}(\text{Intens}, \text{apply-oper}(\text{Var}, \text{Var\text{-}inside}, \text{Temp},\]
\[\text{Intens3}, \text{Oper3}))),\]

\[\text{(equal}(\text{Done}, \text{yes})\]
\[\rightarrow \text{equal}(\text{Predic}, \text{Intens})\]
\[; \text{(Var}(\text{Neg})\]
\[\rightarrow \text{equal}(\text{Predic}, \text{Intens})\]
\[; \text{equal}(\text{Predic}, \text{mynot}(\text{Intens}))).\]
The PARTIAL-INTENS procedure, below, is used to construct the formal representation of the partial predicate of a clause. It checks the negation flag to see whether and in what way it should negate the 'apply-oper' phrase which represents the relation of the object phrase to the subject phrase in the clause. If certain forms of negation are present the Done flag is set to yes and the negation is performed. Otherwise, the Done flag is set to no, and the intension of the predicate phrase, which is not a list but a predicate, is returned unnegated, via the Temp variable.

\[
\text{partial-intens}(\text{Var1}, \text{Var2}, \text{Relation}, \\
\text{Intens2}, \text{Oper2}, \text{Neg}, \text{Done}, \text{Temp}, \\
\text{Var-inside}, \text{Intens3}, \text{Oper3})
\]

\[
= \begin{cases}
(\text{nonvar}(\text{Neg}), \text{var}(\text{Var2}), \text{equal}(\text{Intens2}, \text{true}/\text{nil})) \\
\rightarrow \text{equal}(\text{Op2}, \text{no}) \\
; \text{equal}(\text{Op2}, \text{Oper2})
\end{cases}
\]

\[
(\text{and}(\text{and}(\text{nonvar}(\text{Neg}), \text{equal}(\text{Op2}, \text{some})), \\
\text{not}(\text{equal}(\text{Relation}, \text{equal}(\text{QQ}, \text{WW})))) \\
\rightarrow \text{equal}(\text{Done}, \text{yes}), \\
\text{equal}(\text{Temp}, \text{mynot}(\text{apply-oper}(\text{Var1}, \text{Var2}, \text{Relation}, \\
\text{Intens2}, \text{every})))
\]

\[
; (\text{equal}(\text{Op2}, \text{no}) \\
\rightarrow \text{equal}(\text{Done}, \text{yes}), \\
\text{equal}(\text{Temp}, \text{mynot}(\text{apply-oper}(\text{Var1}, \text{Var2}, \text{Relation}, \\
\text{Intens2}, \text{some})))
\]

\[
; \text{equal}(\text{Done}, \text{no}), \\
\text{equal}(\text{Temp}, \text{apply-oper}(\text{Var1}, \text{Var2}, \text{Relation}, \\
\text{Intens2}, \text{Op2})))
\]

The 'replace' rules shown below will recursively dissect the elements of a list and replace every occurrence of a variable by the variable which is supplied as the second argument.

\[
\text{replace}(\text{Var}, \text{Newv}, L, \text{nil}) :- \text{nonvar}(L), \text{equal}(L, \text{nil}), !.
\]

\[
\text{replace}(\text{Var}, \text{Newv}, F/R, \text{Newv}/\text{Nrest}) :- \text{var}(F), \text{Var} == F, !, \\
\text{replace}(\text{Var}, \text{Newv}, R, \text{Nrest}).
\]

\[
\text{replace}(\text{Constant}, \text{Newv}, F/R, \text{Newv}/\text{Nrest}) :- \text{nonvar}(F), \text{nonvar}(\text{Constant}), \text{Constant} == F, !, \\
\text{replace}(\text{Constant}, \text{Newv}, R, \text{Nrest}).
\]

\[
\text{replace}(\text{Var}, \text{Newv}, \text{Fst}/\text{Rst}, \text{Fst}/\text{Newlst}) :- \text{or}(\text{atomic}(\text{Fst}), \text{var}(\text{Fst})) \\
, \text{replace}(\text{Var}, \text{Newv}, \text{Rst}, \text{Newlst}), !.
\]

194
replace(Var, Newv, Fst/Rst, Nfst/Nrst) :-
    cons(Ls, Fst), mylist( L, Ls)
    , replace(Var, Newv, L, L1)
    , mylist(L1, L2), cons(L2, Nfst)
    , replace(Var, Newv, Rst, Nrst) , !.

replace(Var, Newv, Fst/Rst, Fst/Nrst)
    :- replace(Var, Newv, Rst, Nrst).

{ The set of assertions below are used to define the predicate 'apply-oper' which will check which operator it is passed, and will determine whether the objects satisfying intens2, which get bound to var2, under the conditions required by the operator, also satisfy the relation it is also passed. }

{ The rule below checks that some entity which satisfies intens2 also satisfies the relation it is passed. }

apply-oper(Var, Var2, Relation, Intens2, some)
    :- ( or(var(Var), embedded-not(Relation) )
    -> truth-cond(Var2, Intens2) , Relation
    ; Relation, truth-cond(Var2, Intens2 ) ).

apply-oper(Var, Var2, Relation, Intens2, some-particular )
    :- Relation .

{ The rule below checks that there is a unique entity that satisfies intens2, and then checks that that entity also satisfies the relation it is passed. }

apply-oper(Var, Var2, Relation, Intens2, the)
    :- incrcount(N, level-unig),
       unique(Var2, Intens2, Relation , N)
    , Relation .

{ The predicate below tests that Var stands in the Relation to every Var2 which satisfies intens2. The first of the rules shown below is used when the first argument is bound to a constant. The second rule is used when the Var is unbound. }

apply-oper( Var, Var2, Relation, Intens2, every)
    :- nonvar(Var ) , !,
       incrcount(N, level-every ),
       test-every( Var, Var2, Intens2, Relation, N) .

apply-oper( Var, Var2, Relation, Intens2, every)
    :- embedded-not(Relation), !,
       entity(Var),
       incrcount(N, level-every),
       test-every( Var, Var2, Intens2, Relation, N) .
apply-oper(Var, Var2, Relation, Intens2, every)  
  :-  replace(Var2, Newvar, Intens2, Newint),  
      replace(Var2, Newvar, Relation\nil, Newrel/\nil),  
      rename-all-but(Var2, Intension2),  
      truth-cond(Var2, Intension2),  
      rename-all-but( Var, Relation/\nil, Result ),  
      truth-cond( Var, Result),  
      incrcount( N, level\-every ),  
      test-every(Var, Newvar, Newint, Newrel, N).

{ The procedure below will rename all variables in the  
  Intension, except the variable V. The result, Result, will  
  contain arbitrary variable names for variables other than V. }

rename-all-but(V, Intens, Result)
  :-  equal( V, dumdum), assert(temp(Intens)), fail.

rename-all-but( V, Intens, Result)
  :-  retract( temp(P)), replace( dumdum, V, P, Result).

{ The predicate rules below will test that any V which satisfies  
  Intens does so uniquely. This is done by first checking that  
  some individual satisfies that intension, this is done by the  
  second rule. The third rule checks that no other individual  
  satisfies the intension, and the fourth rule is taken only if  
  such a unique entity was found. If none was found, then the  
  fifth rule is taken. }

{ The first rule below performs an error check to make sure that  
  this goal has not been tried before. }

unique(V, Intens, Relation, N)
  :-  or(message( unique, Y, emptyset(Intens)),  
        message( unique, Y,  
                nonunique(Intens ))),  
        !, fail.

unique( V, Intens, Relation, N )
  :-  truth-cond(V, Intens), assert2(stack(V, N)),  
        checkadd( flag(N, uniq)),  
        fail.

{ Note that we could test for uniqueness by using the builtin  
  SET predicate to find the set of all items which satisfy the  
  given intension, but this might be inefficient if there are many  
  items in the set. My procedure stops as soon as more than one  
  such item is found. }

unique(V, Intens, Relation, N)
  :-  stack(Y, N) , stack(V, N)  
      , not( equal(Y, V)) , ! , assert(global(sub, deviant))
 assert(message(unique, ' ', nonunique(Intens, Y, V)))
 assert2(stack(ZZ, N)), fail.

 nonunique( Intens, Y, V)
 :-
 , nl
 , mywrite('there is a false
 presupposition concerning the uniqueness')
 , mywrite('of the values which satisfy the description ')
 , mywrite('which begins the portion
 of your question shown or described below: ')
 , nl, nl
 , ( global(X, Intens)
 -> printlist(X)
 ; write('The subject of your
 question is the intended phrase.') )
 , nl, nl
 , mywrite('At least the two values below were found. ')
 , nl, nl
 , mywrite(V) , mywrite(Y) , nl
 , assert(deviant), fail.

 unique( V, Intens, Relation, N)
 :-
 retract(flag(N, uniq) ), retract(stack(V, N) ), !.

 unique(V, Intens, Relation, N )
 :- !, assert(global( sub, deviant) ),
 assertonce(message( unique, ' ', emptyset(Intens)) ),
 fail.

 assert2( stack(X, N))
 :- stack(Y, N), stack(Z, N),
 not(equal(Y, Z)), !, fail.

 assert2( stack(X, N)) :- assertonce(stack(X, N)).

 emptyset( Intens) :-
 global(X, Intens), nl,
 mywrite('There is a false presupposition
 concerning the existence'),
 mywrite('of a value which satisfies
 the description which appears'),
 mywrite('at the the beginning of the portion
 of your question shown below: '),
 nl, nl, printlist(X), assertonce(deviant), !, fail.

 emptyset( Intens) :-
 nl, write('There is a false presupposition
 concerning the existence'),
 mywrite('of a value which satisfies
 the description of the '),
mywrite('subject of your question.'),
assertonce(deviant), fail.

{ The rule below is used to retrieve the topmost individual
which has been placed on a stack under the predicate ,
stack(x). }

getstack(Y) :- stack(Y), !.  

{ The rules for 'test-every' below are used to check that every
individual which gets bound to newvar by satisfying 'newint',
i.e. some intension which is passed as the third argument, also
satisfies the relation 'newrel' which involves both Var and
Newvar. }

{ The first rule below repeatedly finds individuals which
satisfy the intension, newint, and then checks to see that such
individuals satisfy the newrelation. If any such individual
fails to satisfy the newrelation then a flag will be left in
place. This flag is then checked by the third and fourth rules
to decide whether 'test-every' should succeed or fail. }

test-every(Var, Newvar, Newint, Newrel, N) :-
  truth-cond(Newvar, Newint), checkadd(flag(N))
  , test(Newrel, Var, Newvar)
  , retract(flag(N)), fail.

test-every(Var, Newvar, Newint, Newrel, N) :-
  not(flag(N)),
  ( truth-cond(Newvar, Newint) -> true
    ; ( member(season(Newvar), Newint)
        -> fail
      ; assert(message(vacuous, ' ', vacuous(Newint)))
      , assert(global(sub, deviant)), fail) ) , !.

test-every(Var, Newvar, Newint, Newrel, N) :-
  unstack(flag(N)), fail.

vacuous(Intens) :-
  ( global(X, Intens)
    -> write('There is a false presupposition
            in your question'), nl,
      write('Concerning the existence of a set
            of entities which'), nl,
      write('Satisfies a description in
            your question appearing'),
      mywrite('At the beginning of the phrase')
shown or described below.

\[
\text{nl, nl,}
\]

\[
\text{printlist}(X)
\]

\[
; \text{nl, nl,}
\]

\[
\text{write('There seems to be a false presupposition in your question')},
\]

\[
\text{mywrite('Concerning the existence of a certain set of entities.'))},
\]

\[
\text{assertonce( deviant), nl.}
\]

\[
\text{checkadd}(F) :- F , ! , fail.
\]

\[
\text{checkadd}(F) :- \text{assert}(F).
\]

\[
\text{assertonce}(P) :- P , !.
\]

\[
\text{assertonce}(P) :- \text{assert}(P).
\]

\[
\text{unstack}(Ax) :- Ax , \text{retract}(Ax) , fail.
\]

\[
\text{unstack}(X).
\]

\[
\text{test}(\text{Rel}, \text{Var}, \text{Newvar}) :- \text{Rel} , !.
\]

\[
\{ \text{The Truth-Cond procedure, shown below, is used to test that the value of Var (if Var is bound) satisfies the list of predicates provided as the second argument. If all the values which satisfies this list have already been found, a TRIEDALL flag is present, and stored assertions are consulted to find the answer. Otherwise, the SATISFIES procedure is invoked to find the answer. If the value of Var is unbound, then Truth-Cond will try to find a value for Var which satisfies the list on conditions in Conditions. Once again, if all values have already been found, then these values are consulted.} \}
\]

\[
\text{truth-cond}(\text{Var}, L1) :- \text{order-nots}( L1, L2),
\]

\[
\text{help-truth-cond}( \text{Var}, L2 ).
\]

\[
\text{help-truth-cond}(\text{Var}, \text{Conditions}) :-
\]

\[
\text{triedall}( \text{Conditions}), \text{Conditions}.
\]

\[
\text{help-truth-cond}(\text{Var}, \text{Conditions})
\]

\[
\text{ :- triedall}( \text{Conditions}), !, fail.
\]

\[
\text{help-truth-cond}(\text{Var}, \text{Conditions})
\]

\[
\text{ :- satisfies}( \text{Var}, \text{Conditions}), \text{assertonce}( \text{Conditions}).
\]

\[
\text{help-truth-cond}(\text{Var}, \text{Conditions})
\]

\[
\text{ :- var}(\text{Var}), \text{assert( triedall( Conditions )}), \text{fail}.
\]

\[
\{ \text{The ORDER-NOTS procedure will rearrange a list of predications so that all the predicates of the form 'mynot(X)' will come last in the list.} \}
\]
order-nots(nil, nil) :- !.

order-nots(F/Rest, L) :- embedded-not(F),
            order-nots(Rest, R1),
            append(R1, F,nil, L), !.

order-nots(F/Rest, F/Newrest) :- order-nots(Rest, Newrest).

embedded-not(mynot(P)) :- !.

embedded-not(apply-oper(V1, V2, Rel, I2, Op)) :- embedded-not(Rel).

{ The rules for 'satisfies', shown below, will check whether some entity can be found which satisfies a list of predications given as the second argument. The variable which appears as the first arg. to satisfies, appears in this list of predications. The list may contain predications of the form 'bind(type, adj)' which must be composed into a predicate of the form adj(type, var), (where the third argument is the variable passed as the first argument to satisfies), before it can be evaluated. }
satisfies(Var, nil) :- !.

satisfies(Var, B/Rest) :- equal(B, bind(T, A)),
                     cons([adj,A,T,Var], Adjpred),
                     Adjpred,
                     satisfies(Var, Rest).

satisfies(Var, C1/Rest) :- not(equal(C1, bind(X, Y))), C1,
                        satisfies(Var, Rest).

{ The rule shown below is used to find the individual which satisfies intens1 and Predication together, and this individual gets bound to var1. }

{ Below are listed a series of rules which apply an operator, such as, 'find-the', or 'the' or 'some' or 'every' or 'find-all' or count-all' to a list of other arguments. Details are provided in comments which follow. }

apply(find-the, Var1, Intens1, Newvar2, Predication) :- ( embedded-not(Predication)
                     -> append(Intens1, Predication/nil, Hold)
            ; equal(Hold, Predication/Intens1) ),
            apply-oper(Newvar2, Var1, true, Hold, the)
The rule below finds the individual, if there is one, which satisfies Intensf, and then checks that this individual satisfies Predication. This rule differs from the one above in that only one individual can satisfy Intensf for this rule to be satisfied. The rule above only requires that uniqueness obtain for any individual which satisfies both Intensf and Predication.

apply(the, Var1, Intensf, Newvarf, Predication) :- apply-oper(Newvarf, Var1, Predication, Intensf, the), printyes, !.

apply(the, Var1, Intensf, Newvarf, Predication) :- printno.

The rule below tries to find some individual which satisfies Intensf and which also satisfies Predication.

apply(some, Var1, Intensf, Newvarf, Predication) :- truth-cond(Var1, Intensf), Predication, printyes, !.

apply(some, Var1, Intensf, Newvarf, Predication) :- printno.

The rule below checks that every individual which satisfies Intensf also satisfies Predication. It calls upon the 'test-every' rules to do this, but it interchanges some arguments in doing so.

apply(every, Var1, Intensf, Newvarf, Predication) :- incrcount(N, level-every), test-every(Newvarf, Var1, Intensf, Predication, N), printyes, !.

apply(every, Var1, Intensf, Newvarf, Predication) :- printno.

printyes :- nl, mywrite('yes it is true'), nl, nl.

printno :- not(message(X, Y, Z)), asserta(message(true-that, 'No it is not true', true)), fail.
The four rules below are used to find all the entities which satisfy \texttt{Intens1}, which also satisfy \texttt{Predication}. The first rule creates a non redundant stack of such individuals and fails. The second rule prints out the elements of that stack if there be any. The third rule prints a message and destroys the stack. And the fourth rule is tried only if the stack was empty.

\begin{verbatim}
apply( find-all, \texttt{Var1}, \texttt{Intens1}, \texttt{Newvar2}, \texttt{Predication}) :-
    assert(cstack(''),
          truth-cond(\texttt{Var1}, \texttt{Intens1}),
          ( \texttt{var(Var1)} -> \texttt{Predication}
            ; testonce(\texttt{Predication}) )
          , maybestack(\texttt{Var1})
          , fail.

apply( find-all, \texttt{Var1}, \texttt{Intens1}, \texttt{Newvar2}, \texttt{Predication}) :-
    cstack(\texttt{V}),
    output(\texttt{V})
    , fail.

apply( find-all, \texttt{V1}, \texttt{I1}, \texttt{V2}, \texttt{Pred} ) :-
    retract( cstack(\texttt{X}) ),
    retract( cstack(\texttt{Y}) ),
    \texttt{nl, nl}
    , mywrite('that is all'),
    unstack(cstack(\texttt{V})),
    !.

apply( find-all, \texttt{V1}, \texttt{I1}, \texttt{V2}, \texttt{Pred} ) :-
    assert(message( find-all,
        'Nothing satisfies the conditions you specify.',
        \texttt{true}) )
    , \texttt{nl, unstack(cstack(V))},
    fail.

maybestack(\texttt{V}) :-
    cstack(\texttt{V}),!

maybestack(\texttt{V}) :-
    atomic(\texttt{V}),
    assert( cstack(\texttt{V}) ),!

maybestack(\texttt{S\&Y}) :-
    atomic(\texttt{S}),
    atomic(\texttt{Y}),
    assert(cstack(\texttt{S\&Y})),!
\end{verbatim}
maybestack(Var1)
fail.

apply(count-all, Var1, Intens1, Newvar2, Predication)
:- cstack(V), incrcount(N, b)
fail.

apply(count-all, Var1, Intens1, Newvar2, Predication)
:- count(1, b),
assert(message(count-all, 
' ', printzero)), !,
setcount(count(0,b)), unstack(cstack(X)), fail.

apply(count-all, Var1, Intens1, Newvar2, Predication)
:- count(N, b), R is N-1, nl,
mywrite('The numerical answer is '), write(R)
, setcount(count(0,b)), unstack(cstack(X)), nl.

printzero :- mywrite('There are none'), nl.

{ The 'count' rule below sets the 'b' count to 0 inititally.
the two flag assertions are merely dummies to insure that error
messages do not occur when checking for axioms with the same
predicate name but with different arguments. }

count(0,b).
count(0, level-every).
count(0, level-uniq).

set-all-counts
:-
setcount(count(0, b)),
setcount(count(0, level-every)),
setcount(count(0, level-uniq)).

{The 'incrcount' rule below will increment the count indexed by
its second argument, and return the current count as its first
arguments. This allows us to maintain an infinity of distinct
counters. }

incrcount(N, I)
:- count(N,I), !, retract(count(N, I))
, X is N+1
, assert(count(X, I)), ! .

{ The 'setcount' rule below is used to set the count indexed by
the second argument to the value provided as the first
argument.}

setcount(count(Number, Countname))
:- retract(count(X, Countname))
, assert(count(Number, Countname) ), !.
{ Below are listed common utility predicates which ought to be included in any decent high level language. Their meanings are made explicit by their names. }

subset(X, Y) :- equal(nil, X), !.
subset(U/Z, Y) :- memb(U, Y), !, subset(Z, Y).
eqsets(L1, L2) :- equal(L1, L2), !.
eqsets(L1, L2) :- subset(L1, L2), subset(L2, L1), !.
equal(X, Y).
memb(X, X/R) :- !.
memb(X, Y/R) :- memb(X, R).
anymemb(X, X/R).
anymemb(X, Y/R) :- anymemb(X, R).
or(P, Q) :- P, !.
or(P, Q) :- Q.
and(P, Q) :- P, Q.
first(X/Y, X).
rest(X/Y, Y).
append(nil, L, L) :- !.
append(X/Y, L, X/R) :- append(Y, L, R).
cons(['S', S, Y], S&Y) :- !.
cons(List, Pred) :- Pred =.. List, !.
cons(List, Pred) :- write('wrong list arg to cons'), abort.
mylist(nil, []).
mylist(X/Rest, [X | L]) :- mylist(Rest, L).
mywrite(X) :- nl, write(X).
mynot(P) :- P, !, fail.
mynot(P) :- not(global(sub, deviant)).
output(X) :- atomic(X), nl, convert(X).
output(S&Y) :- nl, write(S), write(' '), write(Y).
{ The CONVERT procedure is used to convert internal code to a
form readable by users. E.G., cmpt-405 becomes "computing 405",
and 'phil' becomes philosophy. The procedure also prints the
results of the conversion. }

convert( X ) :- get-syn( Y, X ), not( equal( X, Y ) ), !, write( Y ).

convert( X ) :- name( X, [ A,B,C,D, Dash, One, Two, Three ] ),
name( N, [ One, Two, Three ] ),
integer( N ), name( DD, [ A,B,C,D ] ),
get-syn( Y, DD ),
write( Y ), write( ' ' ), write( N ), !.

convert( X ) :- write( X ).

{ The procedure below is used to start up the program once
prolog has been entered. One simply invokes the predicate START
to get the query system going. }

start :- nl, nl,
mywrite('Welcome to the realm of the SHADOW,'), nl,
mywrite('A Natural Language Query System.'), nl,
mywrite('Please enter your queries
using lower case only.'), nl,
mywrite('Questions should be followed
by one or more blanks'), nl,
mywrite('And then a question mark.'), nl,
mywrite('To stop, just type "stop"
and hit return.'), nl, nl,
!,
repeat,
mywrite('Ready for your question.'), nl, nl,
get-ques( Q ),
nl, nl,
F is cputime,
compact( Q, Qlist ),
ans( Qlist ),
G is cputime, Diff is G-F, nl,
write('seconds used = '), write(Diff),
nl,
fail.

{ The COMPACT procedure is used to go through a list of words
and convert expressions like "computing 354" to expressions like
cmpt-354. }

compact( nil, nil ) :- !.
compact( F/S/Rest, Result/Newrest )
:+- integer( S ),
name( S, [ One, Two, Three ] ), get-syn( F, X ), dept( X ),
name( X, [ A,B,C,D ] ),
compact(F/Rest, F/Newrest) :- compact(Rest, Newrest).

{ The GET-QUEST procedure is used to fetch an entire question from the input stream. }

get-ques(Q) :- makeword(W), !, checkstop(W), !, listwords(W, Q).

{ The LISTWORDS procedure is used to list the words which are found in an input question, in the form of a list like, 'a/s/á/f/nil'. The make-word procedure is called upon to fetch individual words from the input stream. }

listwords('?', nil) :- retract(gfound), !.

listwords(W, X) :- retract(gfound), !,
                     mywrite('misplaced question mark'), nl,
                     mywrite('please reenter the query'), fail.

listwords(W, W/Rest) :- !, makeword(Wd), !, listwords(Wd, Rest).

checkstop(stop) :- printstop.

checkstop('stop.') :- printstop.

checkstop(prolog) :- abort.

checkstop(P).

printstop :- nl, nl, nl,
          write('The SHADOW bids you farewell.'),
          nl, nl, nl, halt.

{ The MAKE-WORD procedure fetches the next word from the input stream. It calls upon the CONSUME-JUNK procedure to consume all blanks and carriage returns up to the next non-junk character, which is returned via the parameter C. Once such a "good" character is found, the listch procedure is called upon to read and list all characters up to the next blank or carriage return. }

makeword(W) :- consume-junk(Goodone), !, listch(Goodone, X), name(W, X).

consume-junk(C) :- repeat, get0(C), not(blank(C)), not(cr(C)).
{ When listch finds a question mark it checks the next character after it to make sure it is a blank or a carriage return. If so, a flag is asserted showing that an entire question has been found. }

listch( 63, [63] )
    :- !, get0(C), !, checkch(C), assert( qfound ).

listch(32, [] ) :- !.

listch(10, [] ) :- !.

listch( Ch, [Ch | Rest] ) :- get0(C), listch( C, Rest).

{ The number 10 is ascii for a carriage return. The number 32 is a blank, and the number 63 is ascii for a question mark. }

cr(10).

blank(32).

checkch(X) :- cr(X), !.

checkch(X) :- not(blank(X)),
            mywrite("Only blanks may follow a question mark.'),
            mywrite("Please reenter your question.'), !, repeat, get0(Q), cr(Q), !, fail.

checkch(X) :- repeat, get0(Y), cr(Y).

sqappend([ ], L, L) :- !.

sqappend([F | Rest], L, [F | New ])
    :- sqappend(Rest, L, New ).
REFERENCES


