A NATURAL LANGUAGE INTERFACE FOR AN EXPERT ADVISOR SYSTEM

by

Sylvain Delisle

A Thesis

Presented to the University of Ottawa

in Partial Fulfillment of the Requirements for

the Master of Computer Science Degree

Department of Computer Science

University of Ottawa

Ottawa, Ontario, Canada

February 1987

ABSTRACT

This thesis describes the design and implementation of SEQAP, a natural language front-end which is interfaced to a prototype expert advisor system. This expert system handles questions about the fourth-generation language QUIZ.

Our goal was to construct an interface capable of meeting the natural language processing requirements of the prototype advisor system. The design of SEQAP has been based on the analysis of several QUIZ related samples.

SEQAP is a case-based system implemented in Prolog and organized into three components: lexical, syntactic, and post-processing. The third of these performs either a semantic verification or a format translation, depending on the requirements of the advisor subsystem that called the natural language module. Error treatment is performed at every level, though it is minimal. SEQAP can be used in either interactive or batch mode. The lexicon contains approximately 1700 entries of which 700 are different forms (roots).

A mes parents

ACKNOWLEDGEMENTS

My special thanks go to my supervisor, Dr. Stanislaw Szpakowicz, who has been invaluable as a teacher and as a mentor. While travelling the sometimes sinuous road that led to this thesis, I have often had occasion to appreciate his knowledge and guidance.

I would also like to thank several people who, in their own ways, have influenced my ideas during the period of my graduate study at the University of Ottawa (in alphabetical order): Dr. Stanislaw Matwin, Dr. Franz Oppacher (from Carleton University), Dr. Douglas Skuce, Branka Tauzovich, Charles Truscott, and Dr. Jorge Urrutia.

Finally, I want to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for their financial support.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Goals

The ability to naturally communicate with computer systems is probably what the user community desires the most. Natural language (NL) seems to be the perfect candidate to facilitate man-machine interaction.

Many years after the implementation of the first user interfaces accepting English (or any other natural language for that matter) as their input, we are at a stage where we can draw several conclusions from past experience.

Firstly, natural languages are very complex and the automation of their understanding is indubitably a challenging undertaking whose successful outcome no one should take for granted.

Secondly, even if we still have not achieved human-like competence in such systems, we have made important progress which allows us to build interfaces that possess interesting but restricted capabilities. This means that they accept only a restricted set of linguistic phenomena and that they are usually domain-oriented.

Thirdly, the application of these limited systems in specific situations of the real world can be of great value for simplifying human communication with computer systems.

Probably the best known example of NL front-end application is in the domain of interfaces to data base systems. Nowadays, many of commercial data base systems offer the possibility of being connected to more or less powerful NL interfaces.

Another type of computer system which is becoming an important field of application for NL interfaces is that of expert systems.

This thesis describes the design and implementation of a NL interface for a prototype expert system. The presentation is aimed at readers who have at least a first

degree in computer science or equivalent knowledge in the following areas: programming language theory, including grammars, parsing techniques, and theory of automata; artificial intelligence, including basics of knowledge representation (such as frames and property inheritance links); Prolog programming. We also assume that the reader has a reasonable understanding of English grammar.

The work presented here is not meant to be fundamental research in computational linguistics. As a matter of fact, we do not deal with any of the following linguistic phenomena: relative clauses, conjoined sentences, ill-formed inputs, ellipsis, anaphora and references (although we will have a way to cope with this), and speech acts.

Our main goal is the construction of a simple, although not trivial, NL module that will serve as a NL front-end for a specific expert system prototype.

As part of our methodology, we analysed NL samples (which will be described in chapter 3) in order to determine the necessary linguistic coverage that the interface needed to be endowed with. The system, called SEQAP for Subset of English for the Quiz Advisor Project, has a syntactically-oriented parser augmented with a simple semantic component.

In the next section we describe our research context and the expert system prototype to which our interface is connected.

1.2. Research Context

Our work is part of a research project involving in part the University of Ottawa Department of Computer Science and Ottawa-based software company Cognos Incorporated. Financial support was also provided by the National Research Council and the National Sciences and Engineering Research Council. The work described here was done at the Artificial Intelligence laboratory of the University of Ottawa.

The project is aimed at the development of a methodology for extracting and conveniently representing knowledge about fourth generation languages. The particular fourth generation language on which we will focus our attention is QUIZ, a product of Cognos, briefly described in section 1.3. More specifically, the main objective of the project is to design and to implement a QUIZ expert advisory system. This expert system will accept queries about QUIZ and provide answers in the form of QUIZ programs or "canned" English explanations.

The advisor prototype has two main components. The first one answers questions of the form "How do I do such and such ?" (in QUIZ); we call this part HDI (for How Do I). The answer is usually a partial QUIZ program. The HDI component is being implemented using KEE, an expert system construction tool which integrates frame-based and rule-based programming. The second component, called QAUZ [TAUZOVICH 87], concentrates on the reasoning based on a causal model of QUIZ. It answers several types of questions, first of all questions of the form "Why do I get this instead of that ?". This part has been implemented in Quintus Prolog. It should be mentioned that the Advisor Project was organized in two main subgroups: the expert system group (which was itself composed of the HDI and QAUZ subgroups) and the NL group.

The AI laboratory is equipped with two Sun-3 workstations, one with 4Mb and one with 8Mb RAM, and two Xerox-1186 Lisp machines with 4Mb RAM each.

1.3. An Introduction to QUIZ

QUIZ is a report writer [QUIZ 85] oriented towards the minicomputer market (HP 3000, VAX, and Data General's MV/Eclipse). It is incorporated in POWERHOUSE, a "fourth-generation application development language", which includes (in addition to QUIZ) a high volume transaction processor, called QTP, and an interactive screen developer and transaction processor called QUICK. QUIZ has access to the user's data base and its corresponding data dictionary which describes all the data elements and how they should be formatted on output.

Typically, one would use QUIZ to produce (or generate) a report containing specific information on certain items (e.g. customers, suppliers, employees, ...). To make this more concrete let us suppose that we have access to a data base containing information about the employees of a certain company and that we want to generate a report on which there would be listed, in alphabetical order of their last names, all the female employees working in department D. How would we do this in QUIZ ?

First we must tell QUIZ which (already existing) files of the data base we want to access. In our example, this would be the file which contained information on the company's employees. We can access it with:

```
ACCESS EMPLOYEES
```

where ACCESS is the QUIZ keyword specifying the file(s) that we want to read, and EMPLOYEES the filename.

Second, we want to have only the female employees working in department D. In QUIZ one would specify this with the help of the SELECT statement which selects records that pass a given test. More precisely (in the same example):

```
SELECT IF SEX = "F" AND DEPARTMENT = "D".
```

Third, we want to have the list of names sorted in alphabetical order (on last names). The SORT statement does that:

SORT ON LASTNAMES.

Fourth, we then have to tell QUIZ which items from the accessed file we want to be printed in the actual report. This is done with the REPORT statement:

REPORT EMPLOYEE FIRSTNAME LASTNAME SEX DEPARTMENT.

Notice that we could have reported only EMPLOYEE, FIRSTNAME, and LASTNAME; a title printed at the beginning of the report would say something like "Female employees working in department D". In this way we do not get repetitions of identical informations (SEX and DEPARTMENT).

Now let us put all this together as we would do during a typical interaction with QUIZ:

```
ACCESS EMPLOYEES
SELECT IF SEX = "F" AND DEPARTMENT = "D"
SORT ON LASTNAMES
REPORT EMPLOYEE FIRSTNAME LASTNAME SEX DEPARTMENT
GO
```

where GO is the final statement which causes the report to be executed.

This is a very simple example, but it gives a good idea how QUIZ works. The important point to notice is that QUIZ allows the user to define a report in terms of its specifications instead of a procedure that the system should follow to produce the report. This is the main reason why this kind of language is suited for technically unskilled users.

QUIZ has numerous useful and powerful features that will not be covered in this simple introduction: e.g. multiple-file access, formatting statements (for overriding the default layout), summary statements, a facility for designing production reports. In what follows the reader is not expected to know more about QUIZ than what was outlined in this section.

1.4. Organization of the Thesis

Chapter 2 presents an introduction to the world of natural language processing. The first half is rather general. It introduces the basic concepts that will be used later on. The second half discusses some approaches to the parsing of NL and also summarily describes many representative NL interfaces.

The design of the SEQAP NL module is the topic of chapter 3. There, we will discuss the reasoning and the methodology behind our decisions about what capabilities the interface should or should not possess.

Chapter 4 presents a detailed description of the parser's internals: inputs, outputs, and error treatment.

We conclude, in chapter 5, by an evaluation of SEQAP and an outline of the possible avenues for future development.

The thesis is completed by several appendices containing, among others, the Prolog code of SEQAP and results of its execution.

2. NATURAL LANGUAGE INTERFACES: AN OVERVIEW

2.1. Why Use Natural Language Interfaces ?

2.1.1. Natural Language Interfaces and Other Types of Interfaces

A user interface is an interactive computer program which accepts an input, in a certain form, and translates it into a representation understandable by the system that the user wants to use. In *natural language interfaces* (NLIs) this input form is a certain NL (English in our case). Strictly speaking, this is the definition of the front-end interface, it interfaces the user to the system. The back-end interface takes the system's output and returns a comprehensible answer to the user. In this thesis, the word *interface* should be understood to mean front-end interface.

For example, very frequently a data base system (DBS) uses an interface that translates the user's input, typically a request, into something comprehensible by the DBS. In this case the user does not have to deal with (and to learn) a very technical language (a programming language) to communicate with the DBS.

One of the first commercially successful NLIs was INTELLECT (commercialized by IBM). It is an interface for DBSs. Here is a brief comparison, from [MARTIN 85], between the NL form of a query and its equivalent in a typical fourth-generation language (4GL):

NL form: Show me a ranked percent of total sales by industry.

4GL form:
```
TABLE FILE FORTUNE
SUM PCT_81_SALES
BY INDUSTRY
END
TABLE FILE HOLD
PRINT INDUSTRY AND 81_SALES AND PCT
BY HIGHEST 81_SALES
END.
```

The interface is a communication means that simplifies the process of interaction between the user and the computer system. In contrast with programming languages which are technically oriented and relatively complex to use, NLs are (or are supposed to be) easy to use.

One point must be emphasized: strictly speaking, a NLI accepts only a subset of a NL. The complexity of NLs makes it virtually impossible to include in a computer program all the linguistic knowledge required for a comprehensive understanding of them.

But not all interfaces are NL based. There exist several other types: function-key use, menu-selection, line-by-line prompting, graphic or pictorial, linear keyword, positional keyword, etc. For an interesting comparative evaluation of those types (including NL) an their appropriateness for different kinds of users see [JARKE & VASSILIOU 85].

In recent years, NLIs have become so popular that it now seems that software companies expecting their interface products to succeed on the marketplace will have to include NL capabilities.

The next section discusses the reasons why NLIs are so important today, as well as their advantages and disadvantages.

2.1.2. Natural Language Interfaces: Advantages and Disadvantages

Historically, computer scientists and programmers have been the only direct users of computers. But recently the trend has changed dramatically. The spreading of computer technology has brought computers closer to nontechnical users. The main problem for these people, one of paramount importance, is to be able to easily communicate with computer systems. This situation has forced computer scientists to devote efforts to the construction of user-interfaces that would make computer systems usable by casual users and persons not extensively trained in computer science.

The ultimate way to simplify the interaction is probably to allow the user to converse with the system in his own (natural) language.

It is generally recognized that NLIs are more convenient, more flexible, and more powerful for non-technical users than any other type of interface [JARKE & VASSILIOU 85, MARTIN 85, SPARCK JONES 84, WILENSKY et al. 84].
Even for technically skilled people, some inputs (questions, requests, ...) would be very difficult, if at all possible, to formulate in any other way than by using NL.

The choice of the appropriate type of interface becomes much more difficult when the user community is not uniform. This happens when technically skilled users are mixed with non-specialized users. The ideal solution is almost impossible to find. Interesting options are multi-scheme interfaces (they combine NL with other means of communications like menus, function-keys, ...) and user-adaptable interfaces (they adjust their level to the user's skills) [BUNDY 84]. But these are not very common yet.

On the other side of the coin, NLIs have certain disadvantages:

the probability of introducing errors when typing in the NL input is fairly high, especially when compared with menu-selection and function-key interfaces;

it takes a non-negligible time to enter the NL input;

- the process of correcting errors can be quite demanding and frustrating. If an error is found in the input, the user is generally required to reenter it entirely (hopefully without error this time);

NLs lack absolute precision, ambiguities are quite frequent;

as the user improves his skills in using the NLI, he tends to think that the interface can understand anything and this leads to the overestimation of the interface's real capacities which in turn can be disapointing for the user.

The ideal, or almost ideal, NLI should overcome the above problems. It should cover a large set of linguistic phenomena, allow for extra-linguistic inputs (technical input forms), be robust and user-friendly in its way of interacting with the user, and be adaptable to the user's skills.

As NLIs are becoming more and more popular, *robustness* is certainly one of the most important aspects in their design. Robustness refers to the appropriate treatment of misspellings, syntactic errors, stylistic infelicities, missing words, abbreviations, and also some complex linguistic phenomena (e.g. ellipsis) [CARBONELL & HAYES 83, HENDRIX 77, JENSEN et al. 83, SELFRIDGE 86].

In fact, as mentioned before, people expect NLIs to behave like humans. The state-of-the-art is far from that. But even disenchanted, nontechnical users still prefer to use NLIs, instead of other kinds of interfaces, to communicate with computer systems. Learning programming languages is a complicated task for most unskilled people and for casual users it is just not worth the effort.

A market inquiry done in Germany on NLIs [MORIK 83], showed that potential customers believe in the proportion of 72% that they would be more in control of their system(s) with a NLI. It also showed that 65% of them think that the system's behavior would be more understandable.

2.2. An Introduction to the Basics of Natural Language Processing

2.2.1. Syntax and Semantics

Here is a definition of *syntax* [WINOGRAD 83]: "Syntax is the part of linguistics that deals with how the words of a language are arranged into phrases and sentences and how components (like prefixes and suffixes) are combined to make words."
A common way of representing the syntactic structure of a sentence is as a tree. Other representations will be discussed in 2.3. More will be said about syntax, from a computational standpoint, in the next section.

Semantics refers to the meaning, or the communicative aspect, of an expression. Leech, in [LEECH 81], identifies seven types of meanings:

conceptual	logical, cognitive or denotative content;
connotative	what is communicated by virtue of what language refers to;
social	what is communicated of the social circumstances of language use;
affective	what is communicated of the feelings and attitude of the speaker or writer;
reflected	what is communicated through association with another sense of the same expression;
collocative	what is communicated through association with words which tend to occur in the environment of another word;
thematic	what is communicated by the way in which the message is arranged in terms of order and emphasis.

In this thesis, we will concentrate on syntax. SEQAP's simple semantic component deals only with conceptual and, to a certain extent, connotative meaning. As the reader can easily judge, semantics is an extremely complicated domain. There are

still many unresolved problems in syntax and semantics. It is probably fair to say that the most challenging ones are those in semantics.

2.2.2. Parsing Natural Language

2.2.2.1. Parsing

For readers who are familiar with the parsing of programming languages, or compiler theory [AHO & ULLMAN 77, BARRETT et al. 86], let us simply say that similar principles are applied in the parsing of natural languages.

A *parser* is a computer program that translates its input, which is presented in a specific form, into its corresponding structural representation. This representation can simply be a syntax tree or, if the parser is semantically oriented, a parse tree or a picture of the meaning conveyed by the input. This pictorial representation may, for example, represent the input's objects, events and relations.

As it is the case for programming language parsers, a NL parser relies on a *grammar* which is a set of structural rules that define the legal combinations of words in sentences. In English, sentences are constructed from words which are composed of characters. Legal words are defined by spelling rules and by a dictionary. Finally, words are composed into sentences according to the grammar rules. The main difference between NL and programming languages is that the spelling and grammatical rules for NL are much more complicated and have many exceptions and ambiguities, whereas rules for programming languages are concise and highly structured, have very few exceptions, and generally no ambiguities (ambiguities are discussed in section 2.2.2.2).

Parsing is thus the process by which we determine the specific grammar rule applications that yield a given sentence; it roughly corresponds to determining the structure of an English sentence. In this thesis, the input to a NL parser is always a sentence. We

do not consider the parsing of multi-sentence inputs (e.g. paragraphs, texts, ...).

A grammar for English sentences can be represented by a set of rewriting rules, as in context-free grammars for example, that define the structure of the language (usually a subset of English). As we will see in section 2.3.1.1, the use of rewriting rules is not the only way to represent a grammar. Below we present an example of a very simple English grammar expressed in rewriting rules (in a Prolog-like notation).

sentence --> noun phrase & verb phrase.
noun phrase --> determiner & noun.
verb phrase --> verb & noun phrase.

To determine to what syntactic category (or *part of speech*) a word belongs, the parser has access to a dictionary (or *lexicon*) where all legal words are included along with their associated information (e.g. part of speech, number, ...).
Suppose we want the above grammar to recognize the sentence "The man eats the apple". The corresponding lexicon should contain at least four entries: "the" is a determiner; "man" is a noun; "eats" is a verb; "apple" is a noun.
Of course, in a more realistic parser we would have a much more complex grammar with a much larger lexicon. The lexicon can also contain other kinds of information. In addition to the part of speech, it may have, say, the number (singular or plural). In this way, the parser can use that information to make more detailed verifications of spelling rules and grammatical rules. With that type of information, a sightly improved version of our previous example would reject "The men eats the apple" because of the lack of agreement between the subject's number ("men" is *plural*) and the verb's number ("eats" is third person *singular*).

2.2.2.2. Ambiguity

An important notion in NL processing is that of *ambiguity*. It occurs when a parser can produce more than one representation (parse tree) for a given input. If there are two different derivation trees for a sentence, then it is possible that two different

meanings can be attached to it. Syntactically ambiguous sentences are frequent in NLs.

A syntactic ambiguity can be caused by a lack of information in the parser (rules, lexicon, ...) or by the nature of the input which can be truly ambiguous. In the first case, we do not have sufficiently detailed information in the parser to help disambiguate inputs which are not really ambiguous. For example, the question "How do I print 'X' after 'Y' ?" looks ambiguous to a grammar that does not differentiate whether "after" is used as a conjunction or as a preposition. As a conjunction it must link verb phrases (which is not the case here). As a preposition it introduces a prepositional phrase (which is the appropriate interpretation here). In the case of a truly ambiguous input, no matter how much information we have in the parser, the input has more than one valid interpretation and we cannot tell which one is the most appropriate. For example, in the question "How do I report an item with leading zeroes ?", we do not know which of the verb "report" or the head noun "item" is modified by the prepositional phrase "with leading zeroes" without using semantics (or a lexical convention).

2.2.2.3. Parsing Strategies and Determinism

Parsing strategies used in NL processing fall into two broad classes: top-down and bottom-up. To simplify the discussion we suppose that a sentence is read from left to right.

Even if we do not attempt to cover determinism here, let us mention that it is an important issue in parsing methods. LL(1) (top-down) and LR(1) (bottom-up) parsers, two fundamental types of programming language parsers, are deterministic. But they cannot be used to parse NLs because of their complexity and irregularity. Even if the notion of parsing NLs deterministically has gained some popularity quite recently, the majority of parsing strategies used in actual NL parsers are nondeterministic. Usually they also take a top-down approach rather than a bottom-up one. ATNs and logic grammars (see section 2.3.1) are of that type.

The parsing strategy used in the parser presented in this thesis is also top-down and nondeterministic. It is implemented in Prolog which possesses a built-in top-down depth-first (with backtracking) control mechanism.

2.3. A Brief Review of the Major Approaches for Natural Language Processing

2.3.1. Two Well-Known Approaches to Top-Down Nondeterministic Parsing

2.3.1.1. ATNs

Since its introduction in the seventies, the ATN (*Augmented Transition Network*) formalism has been widely used for parsing NLs. A good part of its success is due to the fact that ATNs were first implemented in Lisp and at that time Lisp was basically the only language used in the artificial intelligence (AI) community. But another reason is that ATNs happened to be the most powerful method suggested to date.

ATNs are based on automata theory. The ATN formalism represents a grammar as a set of labelled diagrams, instead of rewriting rules, where each diagram is similar to a finite state automaton. As in automata theory, ATN diagrams are directed graphs with labelled states, labelled arcs, start and final states. But these graphs are not automata because they allow non-terminal symbols and actions on arcs.

The label of an arc can be of two types: it is either a part of speech or the name of another diagram. In addition, conditions and actions can be associated with an arc. A condition may be, for example, the verification of subject-verb agreement. An action usually constructs a structural representation of the element just parsed and keeps it in a register. This makes the information collected so far available to the diagram-driven interpreter for subsequent processing.

Whenever an input is recognized, we make a transition in the network. A sentence is accepted when every word has been recognized and we reach a terminal state in the network.

Typically, an ATN interpreter is implemented in Lisp and uses the top-down parsing strategy [BATES 78, REICHMAN 85, SHAPIRO 82, WINOGRAD 83].

2.3.1.2. Logic Grammars

Logic grammars (LGs), introduced in the mid-seventies, have only become relatively popular in the eighties. This delay was probably caused by the immense resistance of the AI community to Prolog (LGs are implemented in Prolog). Prolog is to LGs as Lisp is to ATNs.

To put it simply, Prolog includes a formalism, preprocessed and converted into Prolog itself, which allows us to write grammars in a quite natural fashion almost exactly as one would write a context-free grammar, except that the use of parameters is permitted. Here is an example where Number is a parameter used to enforce number agreement.

```
sentence(Number) --> noun_phrase(Number), verb_phrase(Number).
noun_phrase(Number) --> determiner(Number), head_noun(Number).
```

Different types of LGs have been developed over the years. The first type, metamorphosis grammars (MGs), was introduced by Colmerauer in 1975 [COLMERAUER 75]. The appropriateness of this new programming language for natural language processing was Colmerauer's main goal when he created Prolog.

The second type of LGs, definite clause grammars (DCGs), which can be viewed as a simplification of the first type, was introduced in 1980 by Pereira and Warren [PEREIRA & WARREN 80].

More sophisticated grammars were invented to cope with more challenging problems (in natural language processing) such as extraposition (e.g. "The man in the red shirt is the one who John wanted to speak to" contains a left extraposition and "How many chickens were there which crossed the road ?" contains a right extraposition). These are called extraposition grammars (XGs) [PEREIRA 81] and gapping grammars (GGs) [ABRAMSON & DAHL 84].

Logic grammars use a top-down depth-first (left-to-right) parsing strategy. For a good introduction to metamorphosis grammars, see [CLOCKSIN & MELLISH 81, KLUZNIAK

& SZPAKOWICZ 85]. For a brief introduction to the four different formalisms (MGs, DCGs, XGs, and GGs) see [CONDILLAC 86].

An interesting comparison between ATNs and DCGs is presented in [PEREIRA & WARREN 80]. The authors, well-known for their logic-programming bias, argue that "DCGs rate at least as highly as ATNs, and that in several respects DCGs represent a significant advance" under the six following criteria: perspicuity, power and generality, conciseness, efficiency, flexibility, and suitability for theoretical work. The article also proposes a way for translating ATNs into DCGs.

2.3.2. The Syntax versus Semantics Dilemma

Before we go any further, it is important to briefly present a fundamental dilemma in NL processing. It can be expressed with the following question: is (surface) syntax essential to the understanding of NL ?

In relation to this dilemma, three views have been proposed. Some people think (or used to think) that syntax is the essential and sufficient feature in NL understanding.

Others, the majority it seems, see syntax as an aspect of NLs that helps, or facilitates, the understanding but is not absolutely necessary. In other words, the more correct the syntax of a sentence is, the easier it is to understand it. If the sentence has syntactic errors, we can still manage to understand it (up to a certain point).

Some others believe that syntax is not important. All that matters is the meaning attached to the words used in a sentence.

Sections 2.3.2.1 to 2.3.2.3 present approaches that, roughly speaking, correspond to the these viewpoints.

2.3.2.1. Parsing Natural Language with Syntax Only

Thirty years ago (1957), Noam Chomsky introduced his generative grammar theory, the only approach depending essentially on syntax. What was to become the transformational generative grammar approach [AKMAJIAN & HENY 75] is no longer seen as a serious candidate for the automated understanding of NL.

Even though the introduction of this model created a tremendous interest in the modelling of language understanding, it has never been fully accepted by artificial intelligence researchers. First, it is not considered as a probable model of the way humans process language. Second, because it is essentially syntax-oriented, the grammar can produce nonsense sentences. Finally, as the name suggests, transformational generative grammars are better for generation than for analysis (of NL). The latter happens to be the major weakness since the appropriateness for NL analysis is a fundamental feature of computational models of NL understanding. This lead people to think that parsers based only on syntax would never be quite powerful enough.

But the interest for syntactically oriented parsers did not disappear after serious drawbacks were found with the above approach. Researchers in NL processing tried to extract as much as they could from syntax, that is without using semantic processing. The idea is that the understanding (of NLs), more precisely the representation of the meaning, can be based essentially on the syntax tree of the input.

An interesting example is PARSIFAL [MARCUS 80]. Although this parser does not rely only on syntax, in fact it is a kind of case-based parser (see next section), PARSIFAL makes intensive use of syntactic information to understand the input. In other words, even if PARSIFAL is more than a purely syntactical parser, it is definitely syntax driven. His rather impressive results were based on the use of lookahead buffers, to remove ambiguities, and of relatively complex grammar rules. Still today researchers are working on ideas related to Marcus's approach and many of

them regularly comment, criticize or use his proposed solution [BAYER et al. 85, KING 83, LESMO & TORASSO 85, MILNE 86].

2.3.2.2. Parsing Natural Language with Syntax and Semantics

Nowadays, any powerful NL parser (one covering a large set of linguistic phenomena and having a certain understanding of the input's meaning) uses both syntax and semantics.

Certain methods introduce semantic processing very early during the parsing to help disambiguation. In *semantic grammars*, for example, words are categorized under their meaning instead of their part of speech. This means that rules in a semantic grammar try to recognize entities with certain semantic properties rather than syntactic ones. This is done by embodying domain-specific knowledge into the grammar rules. It makes semantic grammars very domain-dependent.

Other methods delay the semantic verification as much as possible. Typical systems of this type operate in two phases: the syntactic phase translates the input into a tree (or an equivalent representation), and the semantic phase checks the appropriateness of the meaning conveyed by every element as well as the overall meaning.

As an illustration of an approach based on both syntax and semantics, here is a brief description of case grammars. This presentation is particularly useful for the understanding of the work presented in this thesis since our approach is based on these ideas.

Introduced in 1968 [FILLMORE 68], *case systems* have been and are still frequently used in NL systems (see 2.4.2.1.4 for example). The power of case systems is derived from a focus on conceptual meaning rather than on syntactic constructions. For example, we probably have a concept of "eating" (somewhere in the back of our mind) which says that the activity of "eating" requires someone to do the eating and some-

thing to be eaten. By these two features one can tell that the sentence "Bill has eaten your sandwich" makes sense whereas "Your sandwich has eaten Bill" is, at least, very strange.

A *case structure* for "eating", in this simple example (assuming that the sentence is always in its active form), would contain a pair of cases that describe a legal sentence where "eating" is the activity. The first case might be the agent (the doer of the eating) and the second one the object (the object eaten by the agent). In addition, we could say that the agent must be animate and the object inanimate. These are semantic constraints on the case fillers. All this semantically describes a simple legal eating activity.

Now the question is how to relate case fillers in the input to the case structure. This is done by adding two other elements of information in the case structure: a case marker and a required/optional flag for each of the two cases. A *case marker* (or case indicator) indicates how to locate a particular case in a syntactic tree. This can be either by the position of an element (main verb's subject, direct object, ...) or by a specific preposition which introduces a prepositional phrase. The associated required/optional flag would indicate whether the case is mandatory (must be found in the input) or not for the sentence to make sense.

Let us refine our previous case structure by augmenting it with these new elements of information. A case will contain the following: a case name, a semantic restriction, a case marker, and a required/optional flag. Here is the case structure of "eating":

Case 1 of eating: agent, animate, subject, required
Case 2 of eating: object, inanimate, direct object, optional.

Let us get back to our previous example sentences. A case-based system would first syntactically parse the sentence "Bill has eaten your sandwich" and get the following information: the subject is "Bill" and it is animate (this would be found in the

lexicon); the main verb is "eaten" which refers to the activity "eating" (this would also be in the lexicon); the direct object is "sandwich" and it is inanimate. Next, the semantic level fetches the case structure of "eating" and verifies the acceptability of every case filler. Case 1 is accepted because the subject ("Bill") is present (as required) and animate. Case 2 is also accepted since the direct object ("sandwich") is inanimate. Thus the parser recognizes this input as meaningful.
The other sentence, "Your sandwich has eaten Bill", would be rejected since the semantic constraint of case 1, namely that the subject must be animate, is not respected. Notice that the semantic constraint of case 2 is not enforced.

This is only a simple introduction. More sophisticated systems have been proposed. Nevertheless, it is sufficient for the understanding of the approach proposed in this thesis.

In conclusion, case systems (or case grammars) consider that the main verb is the center of the sentence and all other elements (noun phrases and prepositional phrases) relate to it in a very precise way that is determined by the main verb's case structure.
For an interesting review of case systems for NL see [BRUCE 75]. For more details on systems using both syntax and semantics see [BARA & GUIDA 84, HARRIS 85, HIRST 84, LEHNERT & RINGLE 82, MARCUS 80, MELLISH 85, SIMMONS 84, SPARCK JONES & WILKS 85, WALLACE 84, WINOGRAD 83].

2.3.2.3. Conceptual Dependency: Deep Semantics

Other approaches, of which *conceptual dependencies* (CDs) [SCHANK 73] is probably the best illustration, focus essentially on the meaning of NL inputs. One of the major motivations of this model is that meaning is the primary issue and the study of syntax should only be guided by the demands of a theory of understanding. As Schank and Riesbeck say in [SCHANK & RIESBECK 81]: "Our point was that there should be no initial independent syntactic pass in understanding. We still do not believe that one

can, or would want to, process any part of language using syntactic information exclusively or primarily.". Another motivation was that any two sentences that have the same overall meaning should be represented in the same way, even if they do not have the same surface syntactic form.

The cornerstone of the theory of CDs, which can be viewed as a type of case-based approach, is the formal representation of events. Whatever the surface form is, if it describes the same event, the internal representation is the same. To enforce this, actions are expressed in terms of primitive acts (or primitive actions). These are intended to be the building blocks out of which verbs are constructed. This means that any verb that describes any action in any event can be expressed in terms of one or more primitive acts.

For example, [SCHANK 76], one of the eleven primitive acts is ATRANS. It means "the transfer of an abstract relationship such as possession, ownership, or control. ATRANS requires an action, an object, and a recipient." In the sentence "John bought a book from Mary", the verb "bought" (which refers to the event buying) is represented by an ATRANS of the book from Mary to John and another ATRANS of the money from John to Mary (with a causal link between the two ATRANSes).

Parsing into CDs means representing a sentence in terms of its events (introduced by verbs), which are themselves expressed as possibly inter-linked structures where actions are described by primitive acts.
For applications of CDs see [BAYER et al. 85, KOLODNER 84, SCHANK 76].

Another semantic approach was first introduced in the mid-seventies by Sowa [SOWA 84]. Conceptual graphs (CGs) form a knowledge representation language that can be seen as an extension to semantic networks. In CGs, nodes represent entities (as well as states, events, and attributes) and relations (between entities). This model is quite formal. Its applications are starting to develop: a first parser has been imple-

mented [SOWA & WAY 86] and other groups, like [FARGUES et al. 86], are using CGs for NL semantics and knowledge processing.

Finally, let us just mention that some other researchers take similar approaches to the understanding of discourse [BRADY & BERWICK 83, DYER 83]. This complex domain we do not consider in this thesis because it is not appropriate for our simple (but realistic) needs.

2.4. Natural Language Interfaces and Tools

2.4.1. Data Base and Expert System Interfaces

This section summarily describes, in a chronological order, ten representative NLIs for data bases and expert systems. The goal is to give an overview of the current state-of-the-art in applied NL processing.

2.4.1.1. RENDEZVOUS (1974)

This system is important not so much for its implementation but for the concepts introduced about dialogues in question answering systems. Codd, known for his work in relational data bases, proposed these steps to make communication between end-users and systems possible [CODD 74, ROSENBERG 80, WALTZ 78]: clarify the request in terms of what the system knows; restate the query in the system's terms and determine if it matches the user's intended meaning; separate query formulation from data base search; use multiple choice interrogation if the user's query fails (to extract the user's intentions); provide a definition capability.

Codd's work established the foundations of data base interfaces. RENDEZVOUS reduced words to their normal form, corrected simple spelling errors, and warned the user when unknown words were used. Its grammar was based on phrase transformation rules to translate user's requests into actual data base queries.

2.4.1.2. PLANES (1976)

PLANES [ROSENBERG 80, WALTZ 78] is a question answering system for a large relational data base of aircraft maintenance and flight data. It was implemented as a case system based on an ATN. The whole system formed a semantic grammar. The lexicon was quite small, only 900 words, but the program had interesting linguistic abilities such as recognizing ellipsis, resolving anaphoric references (pronouns could

only refer to noun groups), and understanding nongrammatical input (telegraphic style). But the system was, however, highly domain-dependent, a drawback of semantic grammars in general, and the linguistic component did not capture much of the regularity of NL. Extending the system was quite difficult. (This is related to the drawback of semantic grammars mentioned before, syntax and semantics are tightly linked and words are categorized in terms of the application domain.) An extension of PLANES, called JETS [FININ et al. 79], increased the completeness of the conceptual coverage (the set of concepts that the system can deal with).

2.4.1.3. LADDER (1977)

LADDER [HENDRIX 77, ROSENBERG 80] is the acronym of Language Access to Distributed Data with Error Recovery. It was developed at SRI International by Hendrix with the help of the LIFER language processing package. The goal was to construct an interface to very large distributed data bases. Since the linguistic capabilities of LADDER are determined by LIFER, details will be given in section 2.4.2.1.1.

2.4.1.4. ROBOT (1977)

ROBOT [HARRIS 79, ROSENBERG 80] is one of the first systems to come close to the notion of domain-independent NLI. Its processing was organized as follows: produce every possible parse (using an ATN); discard parses which present semantic anomalies by consulting the data base; select the most likely interpretation, prompting the user if necessary; retrieve a final answer.

An interesting feature of ROBOT is the use of two ATNs: one for full sentences and one for fragments. ROBOT was good for simple data bases and questions that could be answered by simple test-and-generate methods. A year and a half after its completion, ROBOT was already used in a dozen real-life applications. It was developed into the INTELLECT system (see 2.4.1.7).

2.4.1.5. CHAT-80 (1982)

Implemented in Prolog by Pereira and Warren [WARREN & PEREIRA 82], CHAT-80 is a prototype NL question answering system. Using extraposition grammars, the system translates questions submitted in a subset of English into a logical representation. This logical expression is then optimized (by a planning algorithm) and transformed into a relational database query.

CHAT-80 showed good performance and was meant to be easily adaptable to other domains of application (the one in their article was world geography). The authors gave special attention to the semantics of determiners. They used extraposition grammars to handle relative clauses. Their first experiments though were done on a very restricted lexicon (100 domain-dependent and 50 independent words) and the proofs to support their claim of easy adaptability were rather slender.

2.4.1.6. XCALIBUR (1983)

Lead by Jaime G. Carbonell, XCALIBUR [ANDRIOLE 85, CARBONELL et al. 83, SPARCK JONES 84] was a relatively big project aiming at NL comprehension and generation of NL for expert systems interfaces. Its first target was an interface to the XSEL expert system (Digital Equipment Corporation's automated salesman's assistant) which produced sale orders for automatic configuration by the R1 system (now called XCON). As of today, this system (XCALIBUR, XSEL, and XCON) is probably one of the most successful applications of NL processing and expert systems.

XCALIBUR had to understand imperative requesting actions, assertions of new information, and carry out task-oriented dialogs. It was based on previous work done at Carnegie Mellon University on the CASPAR and DYPAR parsers [HAYES & CARBONELL 81]. XCALIBUR's NL parser, DYPAR-II, is a case frame system (similar to case systems in 2.3.2.2) that recognizes case frames for verbs and noun phrases (in this thesis we have case structures for verbs only, not for noun phrases). The system is

reasonably robust [CARBONELL & HAYES 83] because of the combining of bottom-up recognition (of case headers) with top-down expectation-driven instantiation once a case frame has been recognized. It was able to deal with ellipsis, unknown words, misspellings and other imperfect inputs (simple cases of missing constituent, incorrect segmentation, ...). Two years latter, a modified version of XCALIBUR entered the marketplace under the name of Language Craft (see section 2.4.2.1.4).

2.4.1.7. INTELLECT (1983)

Generally recognized as the first commercially successful NLI, INTELLECT (formerly ROBOT) was produced by the Artificial Intelligence Corporation. It accepts NL inputs for querying data bases and generating reports. Implemented as a semantic grammar parser, INTELLECT deals with ellipsis, unknown words, and prompts the user when it cannot resolve an ambiguity [MARTIN 85]. Having the main disadvantage of semantic grammars (highly domain-dependent), INTELLECT is difficult to transform to different DBSs.

2.4.1.8. UC (1984)

Unix Consultant (UC) [WILENSKY et al. 84] is a prototype expert advisor system accepting queries in English about the Unix operating system. The NL component, which can function as a front-end as well as a back-end interface, is built of several CD-based sub-systems (see section 2.3.4 for CDs).

In order to answer a question, UC first translates it into a conceptual representation. This is done by PHRAN (PHRasal ANalyser) which is the NL front end of the system. PHRAN's output is passed to a goal analyser which makes a plan corresponding to the user's goal which, in turn, is interpreted to generate the answer. As of 1984, the answering component was incomplete. The NL component did not deal with ill-formed inputs and handled only a certain type of ellipsis.

2.4.1.9. FRED (1986)

FRED (for FRont-End for Databases) [JAKOBSON et al. 86] was designed to be an intelligent database assistant. It allows users to communicate via NL or menu selection. The latter can be used as the only way of communication or to solicit information from the user. Unlike INTELLECT, FRED can be interfaced to different DBSs. The parser is a case system coupled with a semantic grammar of the domain. This grammar contains hierarchically organized case frames, with Is-A and Instances (the inverse of Is-A) links, and domain entities. If the input happens to be ambiguous, FRED starts a dialogue with the user to clarify the intended meaning. It can cope with conjunctions and relative clauses.

Once the input has been understood, its case frame representation is used for producing the actual query that the DBS will interpret.

2.4.1.10. SESAME (1986)

SESAME [ALI et al. 86] is a prototype domain-independent NLI for databases. The parser uses separate domain-independent and domain-dependent modules. Its grammar is written in an augmented phrase structure formalism in which context-free rules are augmented with variables that can be bound to the semantic categories of subconstituents. These semantic variables serve to eliminate syntactically well-formed inputs that are semantically unacceptable. SESAME deals with ellipsis and pronoun reference.

Once a query has been syntactically and semantically analysed, it is translated into database commands (in the target database language).

2.4.2. Natural Language Tools

As NLIs become more and more popular, researchers are trying to significantly reduce the amount of resources (time, money, and human expertise) generally involved in their construction. Two options have been proposed so far: customizable NLIs (section 2.4.2.1) and ready-to-use systems (section 2.4.2.2).

2.4.2.1. Customizable Natural Language Interfaces

We want to emphasize that although the concept of customizable NLIs seems legitimate and extremely attractive, one should certainly not view such systems (in their current state) as being "instantaneously customizable". Some of these systems are claimed to be generators of NLIs. This is, at the very least, slightly misleading. The economy of resources, especially in terms of the need for people highly skilled in linguistics and data base technology, remains to be proved. Nevertheless, this new field in NL research will probably gain more attention in the next few years.

2.4.2.1.1. LIFER (1977)

As an application-oriented package for the creation of NLIs, LIFER [HENDRIX 77, ROSENBERG 80] was designed more precisely for the construction of data base interfaces. It is composed of two modules: a language specification facility and a parser. The (semantic) grammar is composed of rules written in the form of production rules. These are context-free and give the grammar the power of Turing machine.
The parser is non-deterministic, top-down, and left-to-right (similar to ATNs). It handles spelling errors, ellipsis, and a simple case of pronominal reference. It is possible to extend the language accepted by the system through the use of synonyms and paraphrases. This can be done by casual users. The system also accepts simple English questions about the language (subset) recognized by the parser.

LIFER has been used in a number of applications for data bases and expert systems, including LADDER (see 2.4.1.3) and PIQUE [DAVIDSON & KAPLAN 83].

2.4.2.1.2. TQA (1985)

TQA [DAMERAU 85] is a domain-independent prototype English interface to IBM SQL-based systems. Organized in four modules (lexical analyser, parser, semantic interpreter, and two-way SQL translator), the interface translates NL inputs into SQL expressions that the DBS can evaluate. The two-way SQL translator works from English to SQL and back (to provide feedback to the user on how the system understands his request).

TQA components are table driven. An interactive customization program guides the adaptation process for dictionaries, system tables, and sometimes grammar rules. The person using the customizer is expected to be a database administrator. The whole process involves providing a large amount of domain specific data. It is very long and does not guarantee to result in a complete and robust interface to every SQL database. Many problems, mostly related to the semantic interpretation, remain to be solved.

Similar work is presented in [BALLARD & STUMBERGER 86]. This system is called TELI. An important difference though is that TELI allows customization to be done by end-users at any time during the normal processing of English inputs (instead of requiring a complete customization before any process can occur).

2.4.2.1.3. GENIAL (1985)

GENIAL [PELLETIER 85] is a system (written in Prolog) that generates French NLIs to Prolog relational data bases. It is claimed that the system allows an implementor of NL interfaces to easily create efficient and robust interfaces. This is a typical example of so-called generators of NLIs. Here is what it means to "generate" an interface: define the target subset of NL; update and complete the information contained in

the lexicon; define the semantic interpretation of new words by adding appropriate Prolog predicates; and finally, update and complete the module defining the NL subset to be handled as well as the hierarchical description of the data base structure.

The implementor is assumed to be knowledgeable in linguistics, data bases, and Prolog programming.

2.4.2.1.4. Language Craft (1985)

As an outcome of the research done in NL processing at Carnegie Mellon University (by Jaime G. Carbonell and Philip Hayes among others), Language Craft (LC) [LANGUAGE CRAFT 85] has emerged from their previous XCALIBUR project (see section 2.4.1.6).

As in the XCALIBUR parser, LC uses the case frame instantiation approach. It translates English inputs into a Lisp representation which in turn is translated into a format appropriate for the application. An important difference with other similar systems is that LC is completely domain oriented. This means that absolutely no knowledge of the meaning of words in general is provided. So the system needs to collect information about: the kinds of objects and operations that the application system deals with; the words and phrases by which the end-user refers to those objects and operations; the grammatical roles these words can play; how to translate the LC internal (Lisp) form to the one needed by the application.

Once again, as it was the case with other similar systems, the operation of LC requires a whole lot of work and highly skilled people. The developer must be familiar with linguistics and Lisp programming.

2.4.2.2. Ready-To-Use NLIs

NL processing products are reaching the marketplace ! We are not talking about huge systems, as described previously, that need highly skilled people and swallow

piles of money over long periods of time. We are now talking about small ready-to-use systems.

Q&A [HENDRIX 86] is an integrated package (file management system, report generator, word processor, spelling checker, "intelligent assistant") that accepts English questions for producing reports and manipulating databases. It is intended to be used on IBM-PC/XT/AT microcomputers. Q&A is, to a certain extent, a miniaturized LADDER that has been tailored to a microcomputer environment. (Hendrix was the project leader of the LADDER system, see 2.4.1.3)

SWAN [SWAN 86] is a bigger system (it requires between 2.5 and 5 Mb of space during execution) that includes a NLI, an expert system inference engine, a data base interface, and graphics capabilities. It can be used, at least that is what the company NLP claims, for complex database interrogation as well as for advisory systems.

The NLI deals with pronoun reference, ellipsis, conjunctions, negations, spelling errors, and certain ill-formed inputs. It can also accept new definitions from end-users. Although this all seems absolutely impressive, the information we have so far does not give a clear idea about the amount of work necessary to customize the system for particular database domains. This customization must be done through the program called Swan Connector (which is sold separately). The system is expected to run on a SUN with 8 Mb of RAM.

2.5. Making the Appropriate Choice

As discussed in chapter 1, our goals were fairly clear as to what the NLI for the prototype expert system should do. It would have to accept inputs in a subset of English, be they questions or statements, and translate them into a structured syntactically-based representation. This would then be interpreted by the prototype expert system (the latter is responsible for returning an answer to the user).

We first analysed the possibility of using NL tools like Language Craft. For reasons outlined in section 2.4.2, we soon realized that doing it ourselves would definitely be the best solution in terms of results, time, and money.

We then had to select our programming environment. Since the equipment was not causing any problems (we had access to Sun workstations, Xerox Lisp machines, and a Vax-750), we were left with the choice of a programming language. Given our knowledge of both Lisp and Prolog, and their respective pluses and minuses for NL processing (see section 2.3) we decided to use Prolog, mainly for its powerful logic grammar formalism.

3. DESIGN OF THE SEQAP PARSER

3.1. Analysis of Samples

3.1.1. Description of Our Samples

In order to determine the language accepted by the parser, we have analysed several NL samples (these are the samples mentioned in 1.2). We had five different sources of NL inputs to consider and account for in our grammar: queries collected at Cognos, the QUIZ manual, frames prepared by the group working on the HDI system (which we call activity frames), a part of the QAUZ system knowledge base (which we call QAUZ knowledge base elements), and a set of queries (prepared by the expert system group) that represented the expected capabilities of the first prototype (P1) of both systems (we call this set P1 queries). (At the time of this writing, QAUZ was completed and did meet its goals.) The following describes each of these samples. Section 3.1.2 explains how they have been useful in the design of SEQAP.

3.1.1.1. Queries

What we call a query is a question (about QUIZ) plus its environment. This environment usually describes the particular circumstances of a situation in which a QUIZ user encountered a problem. More precisely, the environment is given in the form of statements. So a query is composed of zero or more statements plus a question (e.g. I am printing negative numbers. Why am I getting the crosshatch sign ?).

Queries have been collected, in a raw form, by the Cognos consulting service where consultants help customers who have problems with one of Cognos' products. Every time a user requests their help (by phone, telex, or mail) the consultants record the problem on a special form. The expert system group analysed those forms and determined the question taxonomy to be used in the construction of the knowledge

base and in the design of question-answering strategies. A set of 2000 forms was filtered to select only those which met the following criteria: the problem was about QUIZ, the query was classified as not too difficult, the query was not too specific. This selection produced a set of almost 200 queries that were appropriate for our needs.

Then, queries were classified according to their question type. Six major categories were distinguished. Here is their brief description with respective percentages (relative to the total number of queries).

- HDI (or How Do I) : 51%

  (e.g. How do I report a numeric element with leading zeroes ?)

  WHY (or causal) : 21%

  (e.g. Why do I get blank lines when using PRINT AT ?)

  SYN (or syntax) : 13%

  (e.g. How many CHOOSE statements are allowed in a report ?)

  ERR (or error) : 7%

  (e.g. Why do I get the message "insufficient workspace" ?)

- DEF (or definition) : 4%

  (e.g. What is the definition of PERCENT ?)

  HYP (or hypothetical) : 4%

  (e.g. What happens if I use more than one ACCESS statement ?)

These results influenced the design decision of separating the Advisor Project in two systems: the HDI system which would take care of HDI questions, and the QAUZ system which would handle WHY questions plus some of SYN, ERR, DEF, and HYP.

As the first step of the design of the NL module, the NL group of the project was also interested in studying those 200 queries, but now from a NL processing

viewpoint. This is the topic of section 3.1.2.1.1.

In comparison with other methods of collecting samples, this one has the advantage of bringing NL utterances from the real world. Another approach to collecting samples, relatively popular in applied NL processing, is that of simulated dialogues [BATES & SIDNER 83].

3.1.1.2. The QUIZ Manual

The QUIZ manual [QUIZ 85] is a complete guide for beginners as well as for more experienced users. Organized in seven chapters and three appendices, the manual contains approximately 200 pages (more details are given in 3.1.2.2.2).

3.1.1.3. Activity Frames

Our third sample comes from the activity frames. They were created to organize knowledge about QUIZ which is expressed in terms of QUIZ activities and QUIZ objects. A QUIZ activity can be seen as a fundamental action which, at the lowest level, refers to a specific QUIZ statement. An example of a HDI frame follows where Rg and Rf can be considered as a kind of logical variable. Usually, they are used instead of pronouns. We refer to that kind of variable as SEQAP variables. The notion of a SEQAP variable comes from LESK [SKUCE 82, SKUCE 83], a language for expert knowledge description.

More precisely, our third sample was the set of the PURPOSE slots of the activity frames. This slot explains the purpose of the activity that the frame refers to. The notation shown below was used at an early stage of the design of the whole HDI system.

ACTIVITY reporting to a file

NARRATIVE reports to a human-readable file

ISA	reporting
INPUT	report group Rg
OUTPUT	report file Rf
PURPOSE	to report a report group Rg to a report file Rf
SKETCH	insert Rg as <report group>.

3.1.1.4. QAUZ Knowledge Base Elements

The fourth sample was composed of about 150 sentences and phrases found in the QAUZ knowledge base. We had to take into account this set of NL inputs since the SEQAP parser was going to be used as a compiler by the QAUZ system for representing them. Here are some examples where, as in the example of a HDI frame before, we have SEQAP variables. Notice the so-called QAUZ variables (a dot followed by a name) used only by QAUZ and interpreted specially during the parsing.

I have a character field 'C'
accessing multiple files
the skip specification is .SS
records of .FILE are optional
invalid file name
all record items are reported
I retain the default column heading
there is a wraparound
the trailing skip.

3.1.1.5. P1 Queries

Our last sample is a set of about 50 queries, provided by the expert system group, which constitute the test case for the first prototype of the Advisor. These queries are,

for the most part, simplified or clarified versions of some of the original 200 queries. Here are a few examples.

How do I report an average at a control break to a permanent subfile ?

How do I write using a FOOTING statement to a subfile ?

Why did I get the error message "Expected: file key" ?

3.1.2. Defining the Language

One of the interesting methodological aspects of this thesis is the way in which we determined our language. The analysis of the five different NL samples described above guided us in deciding what linguistic phenomena should or should not be covered by the parser. Here, our goal was to observe and generalize linguistic phenomena encountered in the samples.

3.1.2.1. Lexical Information

The analysis of the samples revealed several lexical problems that we had to circumvent:

SEQAP variables and QAUZ variables are necessary (as seen in 3.1.1.3 and 3.1.1.4);

QUIZ-specific proper nouns are sometimes used;

e.g. Can I run QUIZ with data from the *Image DB* from tape ?

e.g. Why do I lose digits when using the *ASCII* function ?

QUIZ error messages, fragments of QUIZ code, and QUIZ keywords are often found in queries;

e.g. Why do I get the error "insufficient work space, increase max data" ?

e.g. SORT A ON B REPORT C SUBTOTAL. C is reset at B, not at A. Why ?

e.g. Can I sort on a DEFINED item ?

- special characters are used in non-standard ways;

  e.g. Why am I getting the crosshatch (#) sign when printing negative numbers ?

- single and double quotes are used in non-standard ways;

  e.g. User was sorting on 'hours' field in his report, and getting more control breaks than expected. Why ?

  three classes of syntactic constructions had to be distinguished: questions, statements, and phrases (a phrase is either a noun phrase or a verb phrase, these are the building blocks of questions and statements);

- abbreviations are sometimes used;

  e.g. How do I override the max number of records printed ?

  noun-noun constructions are quite frequent. We avoid problems with those by introducing the notion of compound words;

  e.g. "record item" is considered as one compound word

- canonical synonyms are needed since we want to handle synonyms (see 3.2.1.3);

Solutions for most of these difficulties (some of which we do not handle) were proposed in the form of lexical conventions. These are enforced during the lexical phase of the parsing process and we describe them, along with that phase, in section 3.2.1. We also studied the lexical diversity of our samples and evaluated the size of the lexicon at approximately 1200 entries (an entry is a form, possibly a synonym of another form). Again, details are provided later on.

3.1.2.2. Syntactic Information

The main part of our linguistic characterization consists of a syntactical modelling of the language which is a subset of English augmented with the use of SEQAP variables. When taking free-form English as samples, as one can imagine, we are likely to encounter many linguistic features that are relatively difficult to handle or difficult to

capture in a simple regularized subset of NL. Ultimately, we aim at a simple model that would be implemented as a logic grammar and would cover most of the linguistic phenomena found in our samples. If a certain phenomenon is not directly accepted by the parser, there should be an alternate way of expressing the same meaning in another syntactic form. (Unfortunately, this is not always true for SEQAP.) Notice that descriptions of phenomena observed in one sample are not duplicated in subsequent descriptions.

3.1.2.2.1. On Queries

This sample is the one in which, among the five, we invested the longest period of time during the sample analysis. The reason is that these 200 queries offer the best example of what a NL interface would have to do, to be useful, in a real-life consulting situation (recall that the goal of the Advisor project is to construct a QUIZ advisor). Here are some of our observations. They are completed by our subset of the English grammar presented in 3.1.2.3:

parentheses are sometimes used as a way of providing an equivalent meaning or more information in order to make the sentence clear and understandable;

e.g. Why am I getting the crosshatch (#) sign when printing negative numbers ?

- most prepositional phrases modify the main verb, but some modify a noun. The solution for this is a lexical convention (see 3.2.1);

  e.g. How do I report a numeric item with leading zeroes ?

  (where "with leading zeroes" modifies the head noun "item")

  e.g. How do I report a numeric item with the REPORT statement ?

  many queries are (syntactically) ill-formed;

  e.g. Getting an error message "message 19: END OF FILE" ?

- SEQAP variables are frequently used;

e.g. a report group Rg

many queries are relatively complex because: they contain conjoined sentences, they contain indefinite references or relative clauses, they contain pieces of QUIZ code (or QUIZ programs);

e.g. How do I select from a certain file records which do not exist in another file ?

most of verb tenses are simple (present simple, present progressive, and past simple), passive form was infrequent.

The notion of "relatively complex" queries happened to be important in our analysis. We categorized queries as "simple" and "complex". This classification is purely syntactic. It does not mean that a simple query is necessarily simple to answer (from a semantic point of view). A simple query does not contain any of the following linguistic phenomena: ill-formed elements, pieces of QUIZ code, indefinite references and anaphora (unless they can be replaced by SEQAP variables), relative clauses, and conjoined sentences. Simple queries represent more than half of the set of 200 queries. They considerably influenced the definition of our set of basic linguistic phenomena, those that the parser absolutely needed to support.

Even if relative clauses, for example, are not complicated to handle, they have not been included in our design for several reasons. First, the Advisor project is oriented towards expert system research, not computational linguistics. So the NL group had to take a practical approach to the construction of the NL front end, respecting the project's plan and deadlines. Also, the project's goal is to produce a prototype. The emphasis is more on experimentation than on production of a ready-to-sell system. Second, HDI and QAUZ subgroups have taken completely different approaches to the modelling of the QUIZ domain and semantic interpretation of their respective inputs. Since we wanted to have only one NL module for both systems, it was to our advantage to keep things as simple as possible. It is not obvious whether the output of a

sophisticated NL parser would necessarily remain simple and relatively easy to process for both systems.

Complex queries require the use of syntactic restrictions or other mechanisms to be acceptable. More than half of these queries were WHY queries that often contained conjoined sentences as well as fragments of QUIZ programs (see appendix A). The decomposition of a query into distinct elements (statements and a question) appeared to be an interesting approach to cope with some of the above problems. That would permit the expression of queries in a constrained but conceptually clear fashion and allow the parser to deal with them.

3.1.2.2.2. On the QUIZ Manual

Since this source is, by far, the most sophisticated (linguistically speaking) sample that we had, it was clear that we would not try to cope with all its linguistic features. In fact the manual has been particularly helpful for terminological problems (although it also showed some deficiencies) and for the analysis of the verbs used in the QUIZ domain. The latter point is discussed in section 3.2.3.

3.1.2.2.3. On Activity Frames

The purpose slots of the activity frames did not bring new linguistic phenomena. More than 90% of them were classified as "simple". They frequently use SEQAP variables and QUIZ keywords.

3.1.2.2.4. On QAUZ Knowledge Base Elements

Essentially composed of statements, this sample convinced us of the importance of according special attention to the handling of statements by the parser. We categorized them under the following categories:

- *have statements*, in which the main verb is "have", they usually introduce a

SEQAP variable;

e.g. I have a character field 'C'.

- *gerund phrases*, which begin with a gerund;

  e.g. cancelling selection condition on records of a file.

  is-np statements, in which the main verb is "be" followed by a noun phrase, they usually introduce a SEQAP variable;

  e.g. 'G' is a subfile.

  is-adjective statements, in which the main verb is "be" followed by an adjective;

  e.g. records of .FILE are optional.

  QUIZ error message statements, in which the statement is a QUIZ error message (this has not been implemented, see 3.1.2.3.1);

  e.g. Expected: file LT LE EQ GE GT NE

  statements in which QUIZ is the actor (this has not been implemented, as being very infrequent and easily rephrased);

  e.g. skips .SS after the report group is printed.

  (rephrased as: the report group is printed after the skipping of .SS)

- *passive statements*, in which the main verb is in past participle form;

  e.g. item .N is printed in the default column.

  regular statements, in which the main verb is any verb other than "be" and "have";

  e.g. I use the ASCII function.

  there-is statements, which begin with "there is" followed by a noun phrase, they often introduce a SEQAP variable;

  e.g. there is no selection condition on record complexes.

- *noun phrases*, inputs in which there is no verb. They usually introduce objects and their attributes;

e.g. the number of subordinate files.

3.1.2.2.5. On P1 Queries

At the time the set of P1 queries was prepared, the design of the parser was finished. But these queries did not bring new problems. In fact the parser was able to handle each one of the 50 queries without any additions. This sample was, most of all, useful for testing purposes. It permitted us to verify what the parser was really able to do and provided important feedback to the expert system group on the actual parsing of their queries.

3.1.2.3. Linguistic Phenomena: Making Choices

After having analysed our samples, we had to decide what linguistic features the parser should have. (How they have been implemented is the subject of section 3.2) Solutions for minor lexical problems, rather technical, and the lexicon are presented in 3.2.1. Here we want to focus on more syntactic issues. The rest of this section deals with solutions of problems listed in the previous section (3.1.2.2). We also present a simplified English grammar that will define the subset of English recognized by the SEQAP parser. Finally, we discuss the coverage of the samples by our grammar.

3.1.2.3.1. On the Nature of the Parser's Inputs

First of all, the parser accepts only English inputs. QUIZ programs, as well as fragments, and QUIZ error messages are not parsed. Nevertheless, we reserved the possibility of keeping such elements in the input by telling SEQAP, by enclosing them in double quotes, that they were not to be parsed. In such a case the parser simply returns the QUIZ element as a separate output parameter (thus leaving a hole in the input).

3.1.2.3.2. On the Complexity of the Parser's Inputs

To handle queries in general, including complex ones, we had to ensure that only "simple" linguistic phenomena would be used. (We do not consider ill-formed inputs; SEQAP does not allow them.) Because of the prototype approach taken in the Advisor project, we did not need more than what we found in "simple" queries. If a query is not "simple", it has to be decomposed into simple elements which will be parsed one after one. In this way, the parser accepts one input at a time, be it a statement, a phrase, or a question, and returns the appropriate representation for each of them. This decision allowed us to use a simpler and more efficient grammar.

This means that when an input happens to be a long sentence composed of several parts (assumptions about the environment, QUIZ code, ...), or it contains indefinite references (or relative clauses, ...), it must be rephrased into simple elements: one or more statements and a question. (Our way to deal with references is to use SEQAP variables.) The problem of long inputs is not more complicated for the parser but rather for the expert system which has to interpret its meaning.

Let us consider an example. Suppose that the original query we want to parse is: "I have a character field which represents YYMM. How can I change it to MMYY ?". What makes this query unacceptable to SEQAP is the use of "which" (relative pronoun) in the first sentence and "it" (anaphora) in the second one. We can avoid such syntactic constructions by using SEQAP variables. Here is the reformulation of that query, equivalent in meaning, but now considered as "simple" and parsable without any problems or ambiguities (where 'C' is a SEQAP variable): "I have a character field 'C'. 'C' represents YYMM. How can I change 'C' to MMYY ?". In this case, the parser would process three elements, two statements and one question.

As mentioned before, a query always contains zero or more statements and one question (the order is not important). We can have as many statements as we wish, the

parser keeping the information on all SEQAP variables till the end of the query. This information is kept in a symbol table in which the variable name is associated with its reference, that is the representation of the object it refers to (e.g. "a file 'F'" causes the parser to add an entry to its symbol table in which 'F' is associated with the representation of "a file"). We can also ask another question (or query) with the same set of variables by telling SEQAP that we do not want to erase the information that it has about SEQAP variables.

3.1.2.3.3. On Verb Forms

We allow verbs in their active form in general for the following tenses: present (simple, progressive, perfect), past (simple, progressive, perfect), and future (simple). In our samples, most of the verbs are in present simple, present progressive, and future simple. Passive forms are infrequent except for that kind of statement that we called "passive statement". It was relatively frequent in QAUZ knowledge base elements.

3.1.2.3.4. On Conjunctions

We already said (in 3.1.2.2.1) that a "simple" query does not contain conjoined sentences. Since we want to base our grammar definition on that category of queries, this implies that a legal input (statement, question, or phrase) must contain only one main verb. Therefore, conjunctions cannot join verb phrases. A conjunction can only join one of the following:

- two noun phrases;

  e.g. How do I report 'X' and 'Y' ?

  two prepositional phrases;

  e.g. How do I report 'X' to 'W' and to 'Z' ?

  a prepositional phrase and conjoined noun phrases;

  e.g. How do I report 'X' to 'W' and 'Z' ?

Here the preposition "to" is elliptical (for 'Z') and this form is considered equivalent to the second example (to 'W' and to 'Z'). The SEQAP parser represents them in the same way (i.e. "to 'W' and to 'Z'" has the same representation as "to 'W' and 'Z'").

3.1.2.4. The Subset of English

This section describes, in a top-down fashion, the subset of English we have defined as sufficient for our needs. Its contents are based on linguistic phenomena observed in our samples as well as on restrictions established in the previous section. Many details have been left out to simplify the presentation.

The grammar is written in a Prolog-like (logic grammar) formalism augmented with the following symbols:

* means zero or more occurrences of the term to its left;

\+ means one or more occurrences of the term to its left;

[] means empty (the empty string).

Words in double quotes are terminal symbols. Words in small letters (and not in double quotes) are non-terminal symbols that refer to other non-terminal symbols of the grammar or parts of speech (other than verbs). Words in capital letters stand for verbs (in any tense, any person). Examples and comments are given in round parentheses. OR (exclusive or) indicates a choice between two alternatives.

3.1.2.4.1. The Top Level

input --> question. (e.g. How do I report an item ?)

input --> statement. (e.g. I have an item 'I2'.)

input --> phrase. (e.g. accessing a file;)

The parser handles questions, statements, and phrases. Phrases are needed mostly for

compiling knowledge elements contained in the expert systems' knowledge bases. (The QAUZ system, for example, stores parses of these elements and uses them for pattern matching when processing its inputs.) Another reason for handling phrases is to allow the possibility, not considered at the moment, of dealing with fragments in situations where the expert system would ask a question and get a phrase as an answer (e.g. From expert system: "What QUIZ statement did you use ?", from user: "the REPORT statement").

3.1.2.4.2. Questions

question --> general_question.
(e.g. How do I print a percent sign after a number ?)

question --> no_verb_phrase_question.
(e.g. Why is 'X' an accessible record item ?)
(Due to our representation of general questions, a header followed by a verb phrase, this form is viewed as having no verb phrase.)

question --> qauz_restricted_question.
(e.g. What is the syntax of ACCESS ?)

general_question --> header, verb_phrase.

header --> "how", rest_how_header.

rest_how_header --> DO, personal_pronoun. (e.g. How do I ...)

rest_how_header --> DO, noun_phrase. (e.g. How does 'X' ...)

rest_how_header --> modal, personal_pronoun. (e.g. How can I ...)

rest_how_header --> "to". (e.g. How to ...)

header --> "why", rest_why_header.

rest_why_header --> DO, personal_pronoun. (e.g. Why do I ...)

rest_why_header --> DO, noun_phrase. (e.g. Why does 'X' ...)

rest_why_header --> BE, personal_pronoun. (e.g. Why am I ...)

rest_why_header --> BE, noun_phrase. (e.g. Why is 'X' ...)

header --> "what", rest_what_header.

rest_what_header --> "happens", "if", personal_pronoun.
(e.g. What happens if I ...)

rest_what_header --> DO, personal_pronoun, "do", "if" OR "when", noun_phrase.
(e.g. What do I do if 'X' ...)

header --> "is", "there", noun_phrase. (e.g. Is there a message ...)

header --> modal, personal_pronoun. (e.g. Can I ...)

header --> DO, noun_phrase. (e.g. Does 'X' ...)

The header is the beginning of a question, from its first word up to the last one before the verb phrase begins. It identifies the question type. General questions fall in four syntactic categories, depending on the header. Those that begin with "how", with "why", with "what", and others. By far the most frequent ones are "how" (handled by the HDI system) and "why" (handled by the QAUZ system).

no_verb_phrase_question --> "is", "there", noun_phrase.
(e.g. Is there an option to the DEFINE statement ?)

no_verb_phrase_question --> "why", BE, noun_phrase, noun_phrase.
(e.g. Why is 'X' an accessible record item ?)

This type of question, rather ad hoc, does not contain a 'real' main verb. All it has is an auxiliary in the question header. The latter is followed by one (in the first case) or two (in the second case) noun phrases.

qauz_restricted_question --> "when", "is", "it", "true", "that", statement.
(e.g. When is it true that 'F' is a primary file ?)

qauz_restricted_question --> qauz_syntax_question.

Questions identified as QAUZ-restricted have an extremely constrained syntax where at most one element (usually a noun phrase) is not fixed. As the name suggests it, they are handled by the QAUZ expert system and cover questions about the syntax of a subset of all the possible QUIZ statements. Their description is given in appendix E.

3.1.2.4.3. Statements

statement --> is_adjective_statement.

statement --> is_a_statement.

statement --> assignment_statement.

statement --> comparison_statement.

statement --> have_statement.

statement --> there_is_statement.

statement --> passive_statement.

statement --> regular_statement.

is_adjective_statement --> noun_phrase, BE, adjectivep.
(e.g. the file 'F' is primary)

is_a_statement --> variable OR proper_noun, BE, noun_phrase.
(e.g. 'RI' is a record item)

assignment_statement --> noun_phrase, "of", noun_phrase, BE, noun_phrase.
(e.g. the number of SELECT statements is 3)

comparison_statement --> noun_phrase, BE, comparative_adjective,
"to" OR "than", noun_phrase.
(e.g. the width of the detail line is greater than the width of the print line)

have_statement --> noun_phrase, HAVE, adverb*, noun_phrase, adverb*.
(e.g. I have a file 'F')

there_is_statement --> "there", BE, adverb*, noun_phrase, adverb*.
(e.g. There is no selection condition on records of .FILE)

passive_statement --> noun_phrase, verb_phrase(passive form only).
(e.g. .TAB is established through an internal procedure)

regular_statement --> noun_phrase, verb_phrase.
(e.g. I defined an item 'Item1')

3.1.2.4.4. Phrases

phrase --> gerund_noun_phrase.

phrase --> noun_phrase.

(e.g. a numeric item with leading zeroes)

phrase --> verb_phrase.

gerund_noun_phrase --> verb_phrase(the main verb must be a gerund).
(e.g. printing an item)

verb_phrase --> "to", verb_phrase(the main verb must be infinitive).
(e.g. to access a file)
verb_phrase --> verb_phrase(the main verb is any tense, except gerund).
(e.g. report an item)

It should be noticed that even though gerund_noun_phrase is defined as a verb phrase, its interpretation seems much more natural as a noun phrase since it refers to the activity described by the gerund.

3.1.2.4.5. Noun Phrases

noun_phrase --> simple_noun_phrase.
(e.g. a new file)

noun_phrase --> conjoined_noun_phrase.
(e.g. a file and a subfile)

There are two types of noun phrases: simple ones (one noun phrase without conjunctions) and conjoined noun phrases in which two or more noun phrases are joined by conjunctions. Although the parser accepts any conjunction, our grammar was designed with "and" and "or" in mind.

simple_noun_phrase --> noun_phrase_form.

simple_noun_phrase --> "[", noun_phrase_form, prepositional_phrase, "]".
(e.g. [a numeric item with leading zeroes])
(for a discussion of square brackets see 3.2.1.1)

conjoined_noun_phrase --> simple_noun_phrase, rest_conjoined_noun_phrase+.

rest_conjoined_noun_phrase --> conjunction, simple_noun_phrase.

noun_phrase_form --> noun_complex.

noun_phrase_form --> noun_complex, quiz_code OR quiz_error_message.
(e.g. the error message "Expected ...")

noun_phrase_form --> noun_phrase_equivalent.

noun_complex --> specifier, adjective_group, common_noun, variable OR [].
(e.g. the primary file 'PF')

noun_complex --> specifier(only those appropriate for mass nouns),
adjective_group, mass_noun, variable OR [].
(e.g. the new data 'ND')

adjective_group --> adjective_p.

adjective_group --> adjective_p, quiz_noun.
(QUIZ nouns (REPORT,SELECT,...) are here considered as special noun modifiers, or pseudo-adjectives. Linguistically speaking, it would be more natural to consider them as compound nouns)

noun_phrase_equivalent --> SEQAP variable OR QAUZ variable.

noun_phrase_equivalent --> personal_pronoun.

noun_phrase_equivalent --> pronoun.

noun_phrase_equivalent --> quiz_noun.

noun_phrase_equivalent --> specifier, gerund. (the gerund as a noun)

noun_phrase_equivalent --> quiz_code OR quiz_error_message.

noun_phrase_equivalent --> number. (number is integer or real)

A noun phrase equivalent is a form which strictly speaking is not a noun phrase but something considered equivalent. This means it can play the same syntactic role as an ordinary noun phrase.

3.1.2.4.6. Adjectives

adjective_p --> adjective_phrase*.

adjective_phrase --> adjective, anded_adjectives*. (e.g. new and primary)

anded_adjectives --> "and", adjective.

3.1.2.4.7. Prepositional Phrases

prepositional_phrase --> simple_prepositional_phrase.
(e.g. to a subfile)

prepositional_phrase --> conjoined_prepositional_phrase.

(e.g. to a file and to a subfile)

prepositional_phrase --> sequence_pps.
(e.g. to a file at a control break)

Prepositional phrases joined by conjunctions are called conjoined prepositional phrases. As it is the case for conjoined noun phrases, any conjunctions are allowed but the parser's internal representation is most natural for "and" and "or". A list of (non-conjoined) consecutive prepositional phrases is called a sequence (of prepositional phrases). A single prepositional phrase is called simple.

simple_prepositional_phrase --> preposition, noun_phrase.

conjoined_prepositional_phrase --> simple_prepositional_phrase,
rest_conjoined_prepositional_phrase*.

rest_conjoined_prepositional_phrase --> conjunction,
simple_prepositional_phrase.

sequence_pps --> simple_prepositional_phrase+.

3.1.2.4.8. Verb Phrases

verb_phrase --> auxiliary, adverb*, verb, adverb*, verb_phrase_complement.
(e.g. "not work" in "Why does the code not work ?")
(e.g. "report an item" in "How do I report an item ?")
(e.g. "subtotal into 'X'" in "How do I subtotal into 'X' ?")
(e.g. "report a subtotal to a file" in "How do I report a subtotal to a file ?")

verb_phrase_complement --> prepositional_phrase OR [].

verb_phrase_complement --> noun_phrase, adverb*, prepositional_phrase OR [].

auxiliary --> auxil.

auxiliary --> auxil, "not".

auxiliary --> [].

auxil --> BE.
auxil --> DO.
auxil --> HAVE.
auxil --> modal.

modal --> "can" OR "cannot" OR "may" OR "must" OR

"shall" OR "should" OR "will" OR "would".
(possibly combined with BE, DO, or HAVE; e.g. must be)

3.1.2.4.9. Specifiers

specifier --> article. (e.g. a, an, the)
specifier --> numeral. (e.g. one, two, three, ...)
specifier --> quantifier_adjective. (e.g. every, some, ...)
specifier --> possessive_adjective. (e.g. my, your, ...)
specifier --> integer. (e.g. 1, 2, 3, ...)
specifier --> integer, "or more". (e.g. 1 or more, 2 or more, ...)
specifier --> integer, "-", integer. (e.g. 1-5; this represents a range)
specifier --> [].

Quantifier adjective is an ad hoc part of speech that refers to the following words: all, any, any1, each, every, some, tsoa (stands for The Set Of All), and unspec. Any1, tsoa, and unspec are specific to the HDI system.

3.1.2.5. Coverage

This section discusses the coverage of our samples by the selected set of linguistic phenomena we have just described. We also denote this set by the word *grammar*, since it is implemented as a Prolog logic grammar.

The grammar covers 90% of simple queries (see 3.1.2.2.1 for "simple/complex" categories), 12% of these with minor modifications to the original query. A minor modification is a simple syntax rearrangement. Here are two examples.
"What do I use to save compiled reports ?" would be simplified as "How do I save compiled reports ?".
"Getting insufficient workspace in version 1.04. What do I do ?" would be simplified as "Why am I getting insufficient workspace in version 1.04 ?".
Complex queries are covered in a proportion of 50%, half of these are acceptable after minor modifications.

Queries not handled by the grammar generally need major modifications. These include the creation of a new syntactic construction, which will respect SEQAP's

grammar, for expressing the same meaning. In other words, to be accepted by SEQAP, the query must be rephrased within the restrictions of our set of linguistic phenomena. Often, such a modification is not easy to find. Here is an example of a query difficult to rephrase: "How do I select a record complex if the selection criteria are met or there is no matching record in the other file ?".

We did not try to measure the coverage of the QUIZ manual since it was way beyond our goals. QAUZ knowledge base elements are covered almost 100%. Activity frames and P1 queries are the only samples completely accepted.

It is worth mentioning that the grammar covers more than what was stricly needed for the first prototype. For example, it can recognize several question forms that the Advisor system's answering algorithm is not yet ready to handle (e.g. How does PERCENTAGE work ? What are the report limits in batch and on-line ?).

3.2. The Technical Design

This section describes, without going to the lowest level of detail, how choices made during the analysis of the samples have been implemented. The SEQAP parser is composed of a lexical analyser, a syntactic parser, and a semantic verifier.

3.2.1. Lexical Phase

3.2.1.1. Lexical Conventions

In order to cope with lexical problems and ambiguities, as outlined in section 3.1.2, the following lexical conventions have been adopted:

single quotes — denote a SEQAP variable. This is a name used by the user and treated as a SEQAP variable. A legal variable name is an atom (a sequence of consecutive characters) in single quotes (e.g. 'File1', 'record_item_X', ...);

capital letters — denote a proper noun or a QUIZ noun. They must start with a capital letter and must be in the lexicon (see 3.2.1.2). The user is expected to use the same spelling (including capital letters) (e.g. REPORT, SELECT, ...);

square brackets — make prepositional phrase attachment explicit. A pair of square brackets must be put around a complete noun phrase, i.e. a specifier plus a noun group plus a prepositional phrase that modifies the noun group. The default is that a prepositional phrase modifies the main verb. Square brackets change this default (e.g. "How do I report a numeric item to a subfile ?" does not need square brackets since "to a subfile" modifies the main verb "report". But "How do I report [a numeric item with leading zeroes] ?" needs square brackets if "with leading zeroes" is to modify the noun phrase "a

numeric item");

double quotes — delimit QUIZ code or QUIZ error messages. The parser will not try to analyse anything that is in double quotes, it just accepts it as it is (e.g. Why am I getting the error message "Cannot find NAME OF BRANCHES" ?);

period — terminates a statement (e.g. I have a file 'F23'.);

semicolon — terminates a phrase (e.g. printing a record item;);

question_mark — terminates a question (e.g. How do I report an item ?);

underscore — links compound words (see 3.1.2.1). Its use is optional, except for proper nouns or QUIZ nouns, where it is considered as part of the spelling. Details about how compound words are processed during the lexical phase are presented in 3.2.1.3 (e.g. data file or data_file);

unspec — the word "unspec" is reserved for identifying an unspecified (missing) specifier (e.g. to print unspec files);

QAUZ variables — QAUZ variables are not interpreted as SEQAP variables. This means that the parser does not try to store or to get the reference that corresponds to a QAUZ variable. It just considers a QAUZ variable as a noun phrase equivalent. These variables have a notation of their own: they start with a period followed by a capital letter and all subsequent characters are letters, slashes or periods (e.g. .ITEM, .ITEM/.RITEM, .FILEn, ...);

Everything else (e.g. parentheses, special characters, or abbreviations) is not accepted by the lexical analyser and is interpreted as an illegal input.

3.2.1.2. The Lexicon

SEQAP's lexicon contains syntactic information for every word recognized by the system. The lexical phase (see 3.2.1.3) fetches that information for words found in the input and makes it available to the syntactic phase which uses it to decide whether the input respects the grammar. We now explain how the lexicon for the SEQAP parser was built.

First, Yannick Toussaint (a research assistant) constructed a concordance program to facilitate the lexical analysis of our samples. In fact we focused our analysis on the set of queries and on activity frames. The concordance program attaches to each word an identification of its origin (the number of the query or frame it comes from) and its one-sentence long context. Following is an example for the word "above". The group of information is called a unit.

above question number: Q1/109
What default heading appears above an item ?

This example says that the word 'above' comes from the question number Q1/109 and was used in the sentence "What default heading appears above an item ?". The context is particularly useful for associating lexical information with every word. This is important for words that may belong to more than one part of speech. The concordance analysis produced huge files in which there were several thousand units.

In the second step we removed duplication in the resulting files. The concordance program does not try to minimize the number of units for each word. If a certain word appears 150 times in the input, it produces 150 units for that word.

We were now ready to start the actual building process of the Prolog lexicon. We wanted to have both a lexicon and an inverted lexicon. The first provides lexical information such as the part of speech; specific lexical information, like number and person, that depend on the word's part of speech; possible continuations, or expected words that can follow it, for compound words; canonical synonym; and the infinitive

form of the word if it is a verb. The inverted lexicon is indexed on syntactic categories and does not contain any other type of information. This allows the listing of all the words that belong to a certain part of speech.

For example, the word *file* is represented in the lexicon as *file(c_noun,sing,nil,nil,nil)* and as *c_noun(file)* in the inverted lexicon, where *c_noun* means common noun and *sing* means singular. The verb *reports* is represented in the lexicon as *reports(v_pre,sing3, nil,nil,report)*, where *v_pre* means present (third person singular) form, *sing3* means number is third person singular, and the fifth parameter (*report*) is the infinitive form. In the inverted lexicon, it is represented as *v_pre(reports)*.

So, in the third step of the lexicon construction process, the input was a list of units and the desired output was a pair of Prolog lexicons in two different files (one for the lexicon and one for the inverted lexicon). To do that effectively, we used the EMACS editor. My supervisor and I programmed little macros that did the following for each word in the file containing all the units:

(1) fetch the next word (the beginning of the next unit) and have it ready for adding its Prolog form containing the lexical information. To have it ready means erasing its origin (it was no longer needed) and put an opening parenthesis next to it. When this was done we then had to type in the Prolog form as shown previously;

(2) complete the Prolog term by adding ")." and have the inverted list entry automatically added in another buffer reserved for this. Before going back to step (1), delete all information about the word just processed (i.e. these macros transfer information from a file to two other files).

We had other macros that would skip an entry (delete a unit), repeat the most recent word (when we wanted to have more than one entry for a certain word), and so

on.

Finally, after having processed all the units and having obtained a first lexicon, we made it complete by adding plurals for all nouns and missing verb tenses for verbs. We also verified the lexicon's consistency and validity, often by consulting English dictionaries and grammars [DRAPS 82, THOMSON & MARTINET 80].

We obtained a lexicon containing approximately 1700 entries (700 different forms): 200 nouns (not counting plurals), 170 verbs (not counting different tenses), 80 adjectives, 80 QUIZ nouns, and 70 adverbs (see appendix C for an excerpt). Of course we have the same number of entries for the inverted lexicon.

3.2.1.3. The Lexical Analyser

The lexical phase has four parts. First, the input is read in as a list of characters and a verification is performed to ensure that they are all legal. Each token must be one of the following: a punctuation mark, an atom, a SEQAP variable, a QAUZ variable, or a string (for QUIZ programs or QUIZ error messages). Also checked is the balancing of square brackets and the end of input (which must be a period, a question mark, or a semicolon). Invalid characters cause a fatal error (it stops the parsing process). The information on every token is preserved in the intermediate output of this first sub-phase. For example, if the input is "I have a file 'F'", the corresponding intermediate output is: [atom(i),atom(have),atom(a),atom(file), var('F'),'.'].

Next, every atom (word) is looked up in the lexicon. If an unknown word is found, the user is prompted with the following choices: stop the processing right there, skip the missing word and continue the parsing, or replace the missing word. Once every word is known, the second part of the lexical phase builds a mini-lexicon. This is a sub-lexicon only for the words found in the input. The information is simply copied from the large lexicon to the sub-lexicon for every word. This approach was chosen to reduce the time spent by the parser in the lexicon. It is clearly more efficient

to restrain the backtracking (during the syntactic phase) to the sub-lexicon, instead of the large lexicon.

Third, compound words are detected and added to the mini-lexicon. Most often a compound word is composed of two nouns (e.g. data file, record item, ...). We do not allow compounds that have more than 2 words. A compound is replaced by a single atom which is the result of appending the composite atoms to one another, linked by underscores (e.g. data file is transformed into data_file).

Finally, words which have a canonical synonym (in the lexicon) are replaced by this synonym. This is especially useful for semantic interpretation (or verification) of the parser's output since the semantic knowledge about the domain is organized in terms of fundamental QUIZ activities and objects. Essentially, a canonical synonym associates a fundamental word with a non-fundamental word (e.g. the canonical synonym of the verb "output" is "report").

The output of the lexical phase is composed of five elements: the original input as processed by the first sub-phase, the canonical input where words that have canonical synonyms have been replaced by these synonyms, the synonym mapping list where every word that has been replaced by its canonical synonym is paired with the latter, the type-of-input flag (question, statement, or phrase), and the status flag which is either "success" or a brief error message. This phase also constructs the mini-lexicon which is used during the syntactic phase.

A feature of the SEQAP system is the possibility of being executed in an interactive mode as well as in a batch mode. In batch mode, interaction with the user is removed. This has consequences on the lexical phase more than anywhere else in the parser because of the prompting done when, for example, an unknown word is encountered. In batch mode, if an error is encountered during the lexical phase (e.g. an unknown word), is it necessarily fatal and the output provides an appropriate error mes-

sage.

3.2.2. Syntactic Phase

In this thesis, the approach we have taken to NL processing is similar to that of case systems outlined in 2.3.2.2. We justify this choice by two arguments. First, it offers an interesting model of NL understanding, from the artificial intelligence point of view, and it is one of the most frequently used (with a certain success) in applied NL systems. Second, it is particularly appropriate in this project because of the importance of verbs (and activities) in the QUIZ domain. In general, verbs found in NL utterances that are related to QUIZ refer (directly) to a QUIZ activity. Basic QUIZ activities correspond to QUIZ statements (e.g. report, access, ...). Recall that the HDI question algorithm organizes its knowledge about QUIZ according to those activities.

The syntactic phase takes as its input the output of the lexical phase and produces a representation based on cases instead of syntax (parse tree) only. It means that the grammar decomposes the input, during the recognition process, into its case constituents instead of outputing a traditional syntax tree. Because the verb has a central function in any case system, the constituents are the main verb, its subject and direct object, and prepositional phrases that modify the main verb.

However, it should be noticed that the parser is not case-driven. The main verb's case structure does not affect the syntactic analysis. The latter is done separately but its resulting tree is expressed in a form suitable for the next phase which is the semantic verification using the main verb's case structure.

SEQAP's syntactic analyser is the implementation of the grammar of section 3.1.2.3. It enforces number agreement between the subject and the verb as well as between the specifier and the head noun of a noun phrase. It also manages the symbol table for SEQAP variables: adding an entry (the reference of a SEQAP variable) when the SEQAP variable is introduced for the first time or fetching its reference when it is

used later on.

3.2.3. Semantic Phase

The input is first syntactically analysed and decomposed into its case constituents: the main verb and nominal groups (main verb's subject, main verb's direct object, prepositional phrases). The semantic phase then performs a simple semantic verification on the main verb's nominal groups. These nominal groups are the surface cases that connect conceptual relations to the surface structure. The semantic phase consists in checking whether the nominal group's head noun belongs to the associated filler type found in the main verb's case structure. For example, in "How do I report an item ?", the nominal group is the noun phrase "an item". The semantic phase verifies that the head noun "item" is a legal filler for the expected type of the direct object of the verb "report". In this example, "item" passes the test. But if we had had "printer", instead of "item", it would have been rejected since reporting a printer does not make any sense in QUIZ. No checking is done for prepositional phrases that modify nouns, only for those modifying verbs (for reasons of simplicity, as mentioned in 3.1.2.2).

In order to implement the semantic verification we had to analyse the verbs found in our domain. This is what the following section describes.

3.2.3.1. Verbs and Cases

The building of case structures requires a good knowledge of the different verbs used in a certain domain. To learn about verbs in our QUIZ domain, we have analysed all the verbs we had in our samples. This produced a list of 200 verbs. For a good part of them we also defined a first set of cases.

We then submitted that list to Cognos experts to verify the validity of our understanding and coverage of the QUIZ domain. We received feedback both from a

linguist and a QUIZ consultant. This helped us to classify verbs into three categories. The first contains verbs that we consider fundamental in the sense that they are associated with a QUIZ activity (e.g. the verb "report" is associated with the QUIZ statement "REPORT") or that they are essential in that there is no other verb to replace them (see appendix B1).

In a second category there are verbs for which a canonical synonym exists in the first category or verbs that are considered not essential for the first prototype. These verbs are categorized as such because they are too complex for the current state of the expert system, they are too complex for our simple model of the QUIZ world, or they are not frequently used (see appendix B2).

The third category contains verbs considered as secondary because they are used in a very broad sense instead of being used in a QUIZ-specific sense. Generally, there is no obvious relationship between such a verb and a QUIZ activity (e.g. need, relate, see, try, ...). Most of them are used infrequently (see appendix B3).

For our needs, we concentrated on verbs of the two first categories. We implemented all case structures of the verbs from the first category. This set of 34 verbs was sufficient to demonstrate the capabilities of our system.

The next step, once we had a fixed set of verbs and their associated case structures, was to attach an allowed filler type to every case. This filler type is the semantic constraint that must be enforced to make sure that the case filler is meaningful.

Deciding about the legal types of case fillers is not an obvious task. It demands a fairly good knowledge of QUIZ semantics. To aid us in this understanding we received the cooperation of the HDI subgroup. They provided us with a hierarchy of QUIZ objects essential for the first prototype. It was helpful in organizing the knowledge on filler types in our case structures. QUIZ objects are part of the HDI model of QUIZ in which actions (events) are expressed in terms of QUIZ activities and QUIZ objects

(e.g. the activity "reporting" acts upon QUIZ items; the thing being reported may be an item). Following is the hierarchy of QUIZ objects that we have considered in our implementation. For details about QUIZ see [QUIZ 85].

QUIZ object --> nameable_object.
QUIZ object --> quiz_statement.
QUIZ object --> quiz_option.

nameable_object --> file.
nameable_object --> item.
nameable_object --> control_break. (point where the value of a sort-key changes)
nameable_object --> report_object.
nameable_object --> device.

file --> display_file.
file --> data_file.

  data_file --> primary_file.
  data_file --> secondary_file.
  data_file --> subfile.

item --> summary_item.
item --> defined_item. (an item defined by the user)
item --> record_item.
item --> sort_key.

  summary_item --> average.
  summary_item --> count.
  summary_item --> maximum.
  summary_item --> minimum.
  summary_item --> percentage.
  summary_item --> subtotal.

report_object --> page.
report_object --> skip.
report_object --> footing.
report_object --> heading.
report_object --> report_column.
report_object --> line.
report_object --> report_group.

  skip --> line_skip.
  skip --> page_skip.

  footing --> final_footing.
  footing --> page_footing.

heading --> initial_heading.
heading --> page_heading.

device --> printer.
device --> terminal.
device --> disc (or disk).

quiz_statement --> ACCESS ; CHOOSE ; DEFINE ; DISPLAY ; EXECUTE ;
FINAL_FOOTING ; FOOTING ; HEADING ; INITIAL_HEADING ;
PAGE_FOOTING ; PAGE_HEADING ; REPORT ; SAVE ;
SELECT ; SET ; SORT ; SORTED.

quiz_option --> LINK.

Augmented with filler types, our case structures now contain the following information (for each case): a case indicator, a mandatory/optional flag, a filler type (semantic constraint), and a case name. Here is an example for the verb "multiply":

Case 1 of multiply: direct object, mandatory, item or number, argument1
Case 2 of multiply: by, optional, item or number, argument2.

The above means that in a legal (meaningful) input in which "multiply" is the main verb, two conditions must be true. First, the direct object must be present and the head noun must refer to either an item or a number. The direct object's case name is argument1. Second, if a prepositional phrase is found, it must be introduced by the preposition "by" and the head noun of its noun phrase must also be either an item or a number. The second case's name is argument2. "How do I multiply printers ?" would be rejected since the filler type of the direct object (printers) is neither an item nor a number. But "How do I multiply numbers ?" would be accepted.

3.2.3.2. The Verification Process

The semantic phase checks the appropriateness of the case fillers found in the input. For every possible meaning of a verb, it tries to match the requirements of its case structure with the actual input's nominal groups.

An input may be rejected when one of the following problems is encountered: a mandatory case is missing, an invalid preposition introduces a prepositional phrase modifying the main verb, an invalid filler type is found. A warning is issued when the main verb does not have a case structure.

The verification process is organized in three modules: the case table which contains case structures for our set of 34 verbs, the QUIZ object hierarchy for organizing the knowledge about filler types, and the verification module which uses the two previous components to perform its task.

3.2.4. Post-Processing Formatting

SEQAP can adjust its output in accordance to the system (HDI or QAUZ) that calls it. If the parser is called by HDI or QAUZ, the output of the syntactic phase is translated into a form more appropriate for those systems, instead of doing the semantic verification (see appendix G). For the time being, no semantic verification is done for either system.

For the HDI system, the parser's normal Prolog output is translated into a Lisp symbolic expression. This is necessary since the HDI system is implemented in Lisp.

For the QAUZ system, the output is simplified but remains a Prolog expression (QAUZ is implemented in Quintus Prolog). This simplification removes elements of information not needed by QAUZ and reformats the parser's output.

It must be pointed out that these two translators do not handle all possible outputs accepted by SEQAP's grammar. They translate only what is necessary, for the first prototype, for each of the two systems. More precisely, for QAUZ, the translator does not handle conjunctions of noun phrases and prepositional phrases. For HDI, the following inputs are not translated: the "have statement", the "passive statement", QAUZ syntax questions (and a few other restricted QAUZ questions), and phrases.

When SEQAP is used in its "independent" mode, neither for HDI or QAUZ, then the semantic phase is always called. The output is given in the parser's internal Prolog form (see chapter 4).

4. THE INTERNAL REPRESENTATION OF SEQAP

In chapter 3 we described the general design of the SEQAP parser. An important part of that chapter was the description of the grammar. In the first part of this chapter we will show the parser's internal representation for the set of linguistic phenomena it recognizes. These representation forms are Prolog expressions that constitute the parser's output. Next, we present our approach to error treatment.

4.1. Representation of Natural Language Inputs

In order to facilitate the reading of the Prolog expressions presented throughout this chapter, we now list the most important abbreviations (these are predicate names or constants used in the Prolog implementation). Details can be found in appendix D. Syntactic categories are listed in appendix C.

d_o : main verb's direct object

ng : noun group (a noun phrase without its specifier; see 3.1.2.4.5)

np : noun phrase

np_pp : standardized noun phrase (see 4.1.1)

p : phrase (noun phrase or verb phrase)

pp : prepositional phrase

q : question

s : statement

spec : specifier

sub : main verb's subject

var : variable (SEQAP or QAUZ).

4.1.1. Noun Phrases and Noun Phrase Equivalents

Here is the Prolog representation of a noun phrase:

np_args(Case_Name, Specifier, Noun_Group).

In this chapter, we denote it by *Noun_Phrase*. A list of noun phrases is denoted by *Noun_Phrases*.

The first parameter, *Case_Name*, is instantiated only during the semantic phase when case structure fillers are verified. If the noun phrase passes the test then *Case_Name* represents the case name of the associated case, otherwise it is uninstantiated.

The second parameter, *Specifier*, is represented as *spec(SPEC)* where the variable SPEC is instantiated to the specifier found in the noun phrase (e.g. In "the record item", *Specifier* is spec(the)).

The third parameter represents the noun group itself (as presented in 3.1.2.4.5, a noun group is a noun phrase without its specifier). Its Prolog form is:

ng(Adjectives, HN_Type, HN, var(Variable), Prepositional_Phrases)

The variable *Adjectives* is the list of adjectives that modify the head noun. *HN_Type* is either c_noun (common noun), m_noun (mass noun), or quiz_proper_noun (QUIZ noun or proper noun). *HN* is instantiated to the noun phrase's head noun (which can be a compound word). If a SEQAP variable is found in the noun phrase, *Variable* keeps its name. The list of prepositional phrases that modify the head noun is kept in *Prepositional_Phrases* (see 4.1.2).

For example, in the noun phrase "a new record item 'R1' with leading zeroes", the following instantiations would take place: *Adjectives* to [new], *HN_Type* to c_noun (common noun), *HN* to record_item (a compound word), *Variable* to 'R1', and *Prepositional_Phrases* to the prepositional phrase representation of "with leading zeroes". In this example, a variable is introduced with its reference (e.g. the reference

of 'R1' is the representation of "a new record item with leading zeroes"). In the case where a noun phrase contains only a SEQAP variable (e.g. How do I report 'R1' ?), we call this a noun phrase equivalent (we recognize seven types of noun phrase equivalent). This means that its reference has been introduced earlier, and if not it is an error because the parser does not know what the variable "means" (or refers to). The parser fetches the reference from the symbol table and attaches it to the variable by creating a normal noun phrase representation in which all parameters are instantiated by associated elements of information found in the reference.

The parser also handles restricted conjoined noun phrases. These are represented as

conj(CONJUNCTION, Noun_Phrases).

The variable *CONJUNCTION* is instantiated to the conjunction that links the set of noun phrases (which is kept in *Noun_Phrases*). SEQAP does not accept noun phrases joined by different conjunctions (e.g. "the file and the subfile or the display file" is not accepted (it is ambiguous); whereas "the file and the subfile and the display file" is accepted).

Noun phrase equivalents form a special case of noun phrase. They are not real noun phrases but they play the same syntactic role in a sentence. Their representation is the same as that of noun phrases except that only certain parameters are instantiated. Below, we present our 7 noun phrase equivalents.

QAUZ variable: *ng(_, qauz_var, QV. _, _)* where *QV* is the QAUZ variable found in the input. These variables are special because the parser does not try to keep or get their references (as it always does for SEQAP variables). More precisely, QAUZ variables are parsed as special proper nouns which are unknown to the lexicon but still accepted.

Personal pronoun: *ng(_, pers_pro, PersPronoun, _, _)* where *PersPronoun* is the personal pronoun found in the input.

Pronoun: *ng(_, pronoun, Pronoun, _, _)* where *Pronoun* is the pronoun found in the input (e.g. *nothing* is reported).

QUIZ noun or proper noun: *ng(_, quiz_proper_noun, QN_PN, _, _)* where *QN_PN* is the QUIZ noun or the proper noun found in the input.

Gerund as a noun: *ng(_, gerund, G, _, _)* where *G* is the gerund found in the input.

QUIZ error message or QUIZ code: *ng(_, quiz_code_errmessage, _, _, _)*. When a fragment of QUIZ program or a QUIZ error message is found in the input, it is kept in a separate global assertion to simplify processing. Nevertheless, a noun phrase structure is produced in which *Head_Noun_Type* is instantiated to quiz_stuff.

Number: *ng(_, number, N, _, _)* where *N* is the number (integer or real) found in the input.

In statements, phrases and questions, noun phrases (and adjectives in the *Is-Adjective* statement, see 4.1.6.1) are represented as pseudo-prepositional phrases with the following Prolog notation:

np_pp(Sub_Do, Noun_Phrase).

Sub_Do is either sub (for the main verb's subject) or d_o (for the main verb's direct object), except for statements not represented with cases (see 4.1.6). This is to standardize the representation of noun phrases and prepositional phrases since they are both

fillers in our case system (they are considered as nominal groups, see 4.1.4).

Note: from now on, the term "noun phrase" will refer to either a real noun phrase or a noun phrase equivalent.

4.1.2. Prepositional Phrases

We have three types of prepositional phrases: simple, conjoined, and so-called sequences. Simple ones are represented as

pp(Preposition, Noun_Phrase).

This representation is denoted by *Simple_Prepositional_Phrase.* A list of simple prepositional phrases is denoted by *Simple_Prepositional_Phrases. Preposition* is instantiated to the preposition that introduces the prepositional phrase (e.g. "to" in "to a file") and *Noun_Phrase* is the noun phrase representation of "a file" (as discussed in the previous section).

As in conjoined noun phrases, conjoined prepositional phrases must be linked by identical conjunctions (which is kept in *CONJUNCTION*). This is represented as

pp(Preposition, conj(CONJUNCTION, Simple_Prepositional_Phrases)).

When *Noun_Phrase* is a set of conjoined noun phrases, we consider the preposition to be distributed over each of the noun phrases. This means that the parser produces the same representation as in the case where we have explicit conjoined prepositional phrases (e.g. "to a file and a subfile" is interpreted as "to a file and to a subfile").

A sequence of prepositional phrases (e.g. at a control break to a file) is represented as a list of simple prepositional phrases.

Note: from now on, the term "prepositional phrase" will refer to either a simple, a conjoined, or a sequence of prepositional phrases.

4.1.3. Verb Phrases

A verb phrase is represented as

vp(MV, MV_Form, Positivity, MV_Modifiers, NP, PPs).

MV is instantiated to the main verb. *MV_Form* conveys information on the main verb's tense. If the verb phrase is part of a question, the header's auxiliary acts as a constraint on the main verb's tense (e.g. In "How do I report an item ?" the header is "How do I" and forces the main verb to be in its infinitive form). *Positivity* indicates whether the main verb is negated. Adverbs are kept as a list in *MV_Modifiers*. *NP* represents fillers that do not have an associated preposition (essentially the main verb's direct object). *PPs* contains all the prepositional phrases that modify the main verb.

For statements and phrases, we use three secondary verb phrase representations which are slightly modified versions of the one above. They handle a restricted passive form (for passive statements) and inputs in which the main verb is either "be" or "have".

4.1.4. Roles

If the main verb does not have a case structure in our system or if the parser is called by the HDI or the QAUZ system, the semantic phase is not performed and roles (cases) cannot be assigned. Nominal groups (subject, direct object, and prepositional phrases) are represented as

no_roles(Noun_Phrases, Prepositional_Phrases)

where *Prepositional_Phrases* is a list of prepositional phrases.

If the semantic phase is performed and the main verb does have a case structure, case names (see Case_Name in 4.1.1) are instantiated and the *np_pp* and *pp* notations (see 4.1.1 and 4.1.2) are changed to *nomi* (for nominal groups) for noun phrases and prepositional phrases (e.g. pp(Prep, Noun_Phrase) is changed to nomi(Prep, Noun_Phrase)). This time, the resulting representation is

roles(Nominal_Groups)

where *Nominal_Groups* is instantiated to the list of the nominal groups found in the input.

4.1.5. Phrases

The representation of a phrase is

p(Phrase_Representation).

If the phrase is simply a noun phrase, then *Phrase_Representation* is instantiated to *regular_np(Noun_Phrase).*

If the input is a verb phrase, *Phrase_Representation* is *vp(Positivity, MV_Infinitive, MV_modifiers, Roles). Positivity* and *MV_Modifiers* are the same as the ones in the previous section. *MV_Infinitive* is the infinitive form of the verb. It is used for accessing the case table during the semantic verification. *Roles* is instantiated to the role (or non-role) format of the nominal groups (see 4.1.4).

4.1.6. Statements

We have 8 different types of statements (see 3.1.2.4.3): is-adjective, is-a, assignment, comparison, have, there-is, passive, and regular. Only the last two, passive and regular, can have a representation expressed in terms of roles (cases), if the semantic phase is executed (as mentioned in 4.1.1, for inputs not represented with cases, *Sub_Do*, in *np_pp(Sub_Do, Noun_Phrase)*, is instantiated to a term that differs from sub or d_o). A statement has the general form *s(Statement_Representation).* The following sub-sections describe what *Statement_Representation* is instantiated to for each of the 8 classes.

4.1.6.1. Is-Adjective Statement

is_adj(Positivity, np_pp(subject,SUB), np_pp(attribute,Atts)).

SUB is instantiated to either a SEQAP variable, a QAUZ variable, or a noun phrase. *Atts* is always instantiated to a list of one or more adjectives (e.g. In "'F' is primary": *Positivity* is yes, *SUB* is var('F'), and *Atts* is [primary]).

4.1.6.2. Is-A Statement

is_a(Positivity, np_pp(identifier,Id), np_pp(identified_object,Id_Obj)).

Id is instantiated to either a SEQAP variable, a QAUZ variable, a proper noun, or a QUIZ noun. The representation of a proper noun and a QUIZ noun is given in 4.1.1. *Id_Obj* is always instantiated to a noun phrase (e.g. In "'F' is a data file": *Positivity* is yes, *Id* is var('F'), and *Id_Obj* is the noun phrase representation of "a data file").

4.1.6.3. Assignment Statement

assign(Positivity, np_pp(object,NP1),
np_pp(object_modifier,NP2),
np_pp(value,NP3)).

The typical form of an assignment statement is "the <object> of <object_modifier> is <value>". Here, the constants "object", "object_modifier", and "value" separate the three elements of such a statement. *NP1* can only be a simple noun phrase. *NP2* and *NP3* can be either simple or conjoined noun phrases (e.g. In "[the number of records] of the file 'F' is 350": *Positivity* is yes, *NP1* is the noun phrase representation of "the number of records", *NP2* is the noun phrase representation of "the file 'F'", and *NP3* is the noun phrase equivalent representation of the number 350).

4.1.6.4. Comparison Statement

comparison(Positivity, np_pp(compar_object1,NP1),
Comparator,
np_pp(compar_object2,NP2)).

This statement allows the comparison of two elements which are noun phrases. *Comparator* is a comparative adjective (e.g. In "the number of [records of [the file 'F']] is

not larger than the number of [records of [the file 'G']]": *Positivity* is no, *NP1* is the noun phrase representation of "the number of [records of [the file 'F']]", *Comparator* is larger, and *NP2* is the noun phrase representation of "the number of [records of [the file 'G']]").

4.1.6.5. Have Statement

have(Positivity, np_pp(subject,NP1), np_pp(direct_object,NP2)).

(e.g. In "I have a file 'F'": *Positivity* is yes, *NP1* is the noun phrase equivalent representation of "I", and *NP2* is the noun phrase representation of "a file 'F'".)

4.1.6.6. There-Is Statement

there_is(Positivity, MV_Modifiers, np_pp(object,NP)).

MV_Modifiers represent the list of adverbs that modify the main verb "be". The object is a noun phrase (e.g. In "there is a file 'F'": *Positivity* is yes, *MV_Modifiers* is [], and *NP* is the noun phrase representation of "a file 'F'"). However, this representation is linguistically inadequate for statements like "there is only one record in the file 'F'" where "in the file 'F'" ought to be attached to the verb "is". Here, it would be attached to the noun phrase "one record". This problem is related to our decision of not having cases for the verb "be".

4.1.6.7. Passive Statement

passive(Positivity, MV_Infinitive, MV_Modifiers, Roles).

The representation of this statement is exactly the same as in the case of a verb phrase (see 4.1.5). A passive-to-active transformation is performed in the representation of that statement. Here, the place of the (active form) subject is held by the direct object. The real subject is unknown or, in our domain, is often QUIZ itself (e.g. In "a link for .FILE is established", the subject is unknown (represented as nil) and the direct object

is "a link for .FILE").

4.1.6.8. Regular Statement

regular(Positivity, MV_Infinitive, MV_Modifiers, Roles).

Once again, as for the passive statement, see 4.1.5 for the representation of this statement.

4.1.7. Questions

Questions are represented by Prolog expressions of the following form:

q(Question_Type, Positivity, MV_Infinitive, MV_Modifiers, Roles).

This representation is similar to the ones for passive and regular statements. The only new element of information is *Question_Type*. This is determined by the question's header. It is important to identify (in the parser's output) the type of question SEQAP parsed for further processing by HDI and QAUZ systems (e.g. In "How do I report an item ?", the header is "How do I" and causes *Question_Type* to be instantiated to hdi; the parser's output will be handled by the HDI system). Here is the list of question types considered for the first prototype (see 3.1.1.1). We show every question header, its associated question type, and the subsystem which handles it. The elements are triplets of the form: question-header / question-type / subsystem.

how do I / hdi / HDI
how to / hdi / HDI
why do I / err,why, or why_not / QAUZ
why am I / err,why, or why_not / QAUZ
why does / why / QAUZ
why is / why / QAUZ
how can I / hci / QAUZ
what happens if / hyp / QAUZ
is there / ? / Not handled in P1
how does / ? / Not handled in P1
what do I do / ? / Not handled in P1
can I / ? / Not handled in P1
does "np" / ? / Not handled in P1.

4.2. Error Treatment

4.2.1. Introduction

It is a known fact that error treatment in Prolog is complicated by the backtracking control mechanism. What we mean by error is a fatal error, not an unsuccessful exploration of one possibility when at least one other remains. A fatal error would be, for example, the detection of lack of agreement between subject and verb number. In other words, an error occurs when we can tell for sure that, at a certain point during the computation, one necessary condition is not true or no other possibility remains for that clause.

The problem is that when such an error is encountered somewhere in the tree representation of the search space, we would like to stop the parsing right there and output an error message, which would explain to the user what the error is, and supply actual values of important parameters at that time. Unfortunately this is not always possible in Prolog. A few implementations will allow for this type of error handling. Prolog II [GIANNESINI et al. 85], for example, has a pair of predicates, called *block* and *block-exit*, that permits to go directly from the point where the error was found to the ancestor goal that generated the call to that module (or block). In implementations that do not have a similar mechanism, such as ours (Quintus Prolog), one has to implement his own machinery to deal with errors.

Desirable features of such machinery are: the capability of keeping meaningful information on the circumstances of the error, good performance, and implementation neatness. The first feature means that the error mechanism must facilitate the collection of relevant elements of information on the parsing state at the time the error was found. It usually involves the taking of a "snapshot" of the appropriate variables or simply the fetching of a human-readable error message. The second feature, performance, means that error treatment should be as efficient as possible. The execution

time for the parsing of an invalid input should, ideally, be the same as the one for a legal input. The third feature, neatness of implementation, refers to the approach taken in the coding of the error mechanism. A neat solution should not make the code unreadable, should not complicate the normal processing, and should not use global assertions.

4.2.2. Our Approach

The approach we have taken to error treatment in SEQAP is based on the use of one error parameter, called ERR, and one flag which indicates whether a constituent under consideration is mandatory or optional. This flag is called M_O.

4.2.2.1. The Error Parameter

The error parameter ERR is added to all the clauses in which an error can occur. At the beginning of the parsing, ERR is uninstantiated. As soon as one error is encountered, ERR is instantiated to a Prolog expression composed of a human-readable error message and the remainder of the input (e.g. In "I have a files", ERR is instantiated to *error(invalid_noun_phrase, [a,files])).* The idea is to get as far as we can in the parsing and provide an appropriate error message as well as an indication on where, in the input, the error was found.

With this approach, the parser does not stop when an error is found. It tries to find other possibilities in the search space. But since we have ERR instantiated (because an error has been found before) the parser does not try to change its value if another error is found. This means that even if the input is invalid, we let it "go through" the parsing process. The parsing never fails. It always returns the last structure it tried to satisfy, when it encountered a fatal error, and a status flag. If the status flag is different from "success" then an error has been found and status is instantiated to *error(Message, Rest_of_the_Input),* as described above. The error information is

always on the first error found.

ERR also facilitates the skipping of certain steps of the parsing process. We just need to test whether it is instantiated or not. If it is, then there is no point in trying to go further. For example, if an error is found during the syntactic phase, the post-processing phase is not even started.

4.2.2.2. The Mandatory/Optional Flag

We have shown how to extract information on an error, but how do we determine that an error has effectively been discovered ? First, it can be done with ERR alone in situations where the error happens at a relatively high-level. As an illustration, let us suppose that the input we want to parse is a question and that the high level grammar is expressed as follows:

```
question(_)   --> question1 or question2.
question(ERR) --> {ERR=error(invalid_question)}.
question1 --> ...
question2 --> ...
```

If during the parsing of a question we get to the second clause of *question*, we know that the input is invalid but we do not know where exactly. So all we can output as an error message is "invalid_question". This is not very helpful.

What we need is a way of detecting errors at the lowest possible level or, in other words, a finer-grain error detection mechanism. This is done with the help of the M_O flag. This new parameter is added to the noun phrase and verb phrase clauses. The goal is to help the parser in deciding whether a clause was absolutely needed when it was invoked. If so, M_O is instantiated to "mandatory" by the clause that calls the one that it absolutely needs to succeed. If the called clause does not succeed then we know for sure that this is an error and that we must instantiate ERR.

Let us illustrate the general idea with a first example. A prepositional phrase is composed of a preposition and a noun phrase. Thus, in the grammar, the clause for

prepositional phrase must call the noun phrase clause with M_O instantiated to "mandatory". If the noun phrase clause does not succeed then an invalid prepositional phrase has been found and ERR must be instantiated accordingly. In short, in a prepositional phrase we expect only a noun phrase after a preposition, and anything else is incorrect.

To conclude this section, let us present a second example, this time more detailed. Suppose that the input submitted to SEQAP is "How do I report an items". This question is unacceptable because of the lack of number agreement between the article "an" (singular) and the common noun "items" (plural). A question is decomposed into two main parts: a header (here, "How do I") and a verb phrase (here, "report an items"). A verb phrase is itself decomposed into two parts: a main verb (here, "report") and a noun phrase (here, "an items"). Since the lack of agreement is discovered at the level of a noun phrase, the error message will be *error(invalid_np,[an,items])*. This message facilitates the correction of the input by the user. Notice that without the appropriate grain of error detection, the error message could have been *invalid_vp*, or even worse, *invalid_question*.

This approach to error treatment is easy to implement, fairly efficient (see 5.1), and fairly neat.

5. CONCLUSION

In carrying out the work described in this thesis, the author acquired a better understanding of two important aspects of applied NL processing: methodological issues in the constructing of a NL interface and linguistic issues in the parsing of NLs.

We believe that a methodology is necessary for ensuring that an interface will meet its requirements. However, this methodology should also make room for linguistic issues; the resulting system must be based on valid linguistic hypotheses. A crucial phase in the design of a practical NL interface is certainly the definition of the language (or set of linguistic phenomena) recognized by the parser. One has to be careful when making compromises on the generality of the set of linguistic phenomena to be dealt with, lest solutions become ad hoc and system adequacy be compromised.

It is our conviction that the methodology presented in this thesis is particularly interesting to apply in situations where: i) valuable domain-related samples can be obtained ii) the knowledge, about the domain, can be organized in terms of activities (and case structures).

Presently, SEQAP is used as the NL front end of the QUIZ Advisor. Both subsystems, HDI and QAUZ, take SEQAP's output as their initial input.

5.1. Performance

In this section we present a brief evaluation of the overall performance of the SEQAP system. Despite the fact that minimizing execution time was not a primary goal for us, we managed to get good results. In what follows, figures were obtained from the parsing of some 50 representative inputs on a Sun-3 workstation.

For short inputs (e.g. 'F' is a file), the average CPU time spent in the lexical phase is 125 ms as compared with 100 ms for the rest of the parsing process, where the latter consists of the syntactic phase and post-processing (which is either the

semantic verification or the formatting required by the HDI or QAUZ systems). Formatting takes up to 20% longer than the semantic phase. For long and more complicated inputs (e.g. How do I report an item at a control break to a file ?), 280 ms are spent in the lexical phase and 240 ms in the rest of the processing, on average. So the average total time for parsing short inputs and long inputs are, respectively, 225 ms and 520 ms.

As can be noticed, the time spent in the lexical phase is 20% longer than the time spent in the rest of the parsing process. This is accounted for by the amount of processing done in the first phase: lexical conventions must be dealt with, the large lexicon must be accessed and used to build the mini-lexicon, compound words must be handled and synonyms must be "canonicalized".

For syntactically or semantically erroneous inputs, the time spent after the lexical processing is at most twice the average time needed to parse the same input without errors. The average increase is approximately 45%. The variation depends on the nature of the error and how soon it was detected in the input. Although this proportional increase may seem important, the parser only spends an average of 150 ms (instead of 100 ms) for short erroneous inputs and an average of 345 ms (instead of 240 ms) for long erroneous inputs. These figures are still very reasonable.

5.2. Advantages and Disadvantages of the System Design

5.2.1. Advantages

To begin with, the parser behaves in a consistent and relatively transparent way. It displays relevant information during its execution (see appendix F) and always returns an error message with the parse tree (possibly anomalous in the event that an error is detected in the input).

Secondly, the overall performance (speed of execution) of the system is good, especially when one takes into account the size of the parser (approximately 1000 lines of Prolog code) and the size of the lexicon (see 3.2.1.2).

Thirdly, some of the parser's components are easy to extend. The syntactic module can be almost effortlessly augmented with the ability to handle new questions and statement forms, as long as they use the allowed set of linguistic phenomena (as defined in chapter 3). Adding the ability to process truly new linguistic phenomena (like relative clauses or ellipsis) would clearly be more complex. The semantic module is also easy to extend. Two of its subcomponents, namely the case table (which contains case structures) and the hierarchy of QUIZ objects, can be augmented with new case structures or QUIZ objects almost as easily as a new entry can be added to the lexicon.

Fourthly, the semantic module handles certain ambiguous inputs. It is typical of such inputs that the verb has more than one case structure (meaning), or the verb has a case indicator that can introduce more than one case. An illustration of the former possibility is given by questions containing the verb *add*. SEQAP distinguishes the two different meanings of *add* in "How do I add the LINK option to the REPORT statement ?" and "How do I add numeric items after a control break ?". The latter case is illustrated by the questions "How do I print a file *on the printer* ?" and "How do I print an item *on a new page* at a control break ?". SEQAP associates "the printer" and

"the new page" with different cases despite the fact that both phrases are introduced by the same preposition ("on").

Finally, SEQAP has the following features: it can be used in either interactive or batch mode, it can translate most of its outputs into a Lisp representation, and it deals with anaphora in a simple way with SEQAP variables.

5.2.2. Disadvantages

Even if SEQAP did meet its goals, and exceeded some of them, it nevertheless has some drawbacks. From a linguistic point of view, SEQAP is not a sophisticated parser. The set of linguistic phenomena that it deals with is simple and restricted. We did not try to cope with 'classic' problems, like relative clauses and ellipsis, for reasons mentioned in chapter 3.

Our system also incorporates a certain number of linguistic adhockeries: the way we treat determiners and quantifiers (which we call specifiers), our treatment of QUIZ nouns (which are sometimes considered as a special kind of adjective), the way we treat auxiliaries and verb tenses (they are not represented in the parser's output), and the definitions of some statement forms.

On a more technical level, the syntactic component does not perform total (syntactic) disambiguation. If a word has more than one potential part of speech, it can happen that the first one tried is parsed successfully even if it is not linguistically adequate. For example, in "How do I report after an item ?", the word "after" is a preposition. But in general, "after" is either a preposition, a conjunction or an adverb. If we first try to parse "after" as an adverb, the above question will be parsed 'successfully' and "an item" will be considered to be the direct object (!).

A related weakness is the incompleteness of the syntactic constraints. SEQAP does not perform a complete filtering of all the possible ungrammatical inputs. For example, the statement "I printing an item" would be accepted.

5.3. Possible Avenues for Future Work

This section presents a realistic outline of possible improvements of SEQAP under the constraint that they should be implementable with a few months' work.

In a first effort we would improve the quality of the interaction between SEQAP and the user to allow for the possibility of interactively adding new words to the lexicon. This would be accomplished by a small module which would obtain syntactic information about new words from the user.

Secondly, error correction could be made more convivial by requiring only that the user reenters the invalid construct (noun phrase, verb phrase or question header) instead of the whole input. If the input is "How do I report an items ?", the user should only be required to reenter the noun phrase "an item" (if it is what he intended).

Two related problems mentioned in section 5.2.2 would need to be addressed in a third step: limited syntactic disambiguation and incomplete syntactic constraints. The goal here would be to improve the 'purity' of the subset of English accepted by SEQAP.

In a fourth stage, the semantic module could be improved by extending the case system. This could be done by increasing the number of verbs that have a case structure in the system and by extending the scope of the semantic verification by checking the modifiers of noun phrases. The latter is certainly a challenging task that would require the modification of every component in the semantic module. The hierarchy of QUIZ objects would need to be refined and the case table would need to be augmented with semantic constraints for the modifiers of noun phrase. This would allow SEQAP to reject inputs like "How do I report an item at [a control break of [a printer]] ?", in which "of a printer" does not make any sense as a modifier of "a control break".

Finally, the parser would be linguistically more complete if it were capable of accepting passive forms in general (instead of only the restricted case, passive statements). But this is true of any linguistic phenomena added to SEQAP. One has to be cautious here and keep in mind the constraint stated at the beginning of this section. Attempting to cope with complex phenomena, like ellipsis (intrasentential and intersentential) and conjunctions (SEQAP accepts only certain conjoined forms), might require a major redesign of the parser. Conjunctions, for example, are recognized as one of the most difficult problems in NL processing [DAHL & McCORD 83, FONG & BERWICK 85, WINOGRAD 83]. In order to handle such linguistic phenomena, we would most likely have to reimplement the parser in another Prolog formalism (e.g. gapping grammars) and this would definitely be more than a 'natural' extension.

REFERENCES

REFERENCES

[ABRAMSON & DAHL 84] Abramson, H. and Dahl, V., *Gapping Grammars*, In: Proceedings of the Second International Logic Programming Conference, University of Uppsala, Sweden, July 1984.

[AHO & ULLMAN 77] Aho, Alfred V. and Ullman, Jeffrey D., *Principles of Compiler Design*, Addison-Wesley 1977.

[AKMAJIAN & HENY 75] Akmajian, Adrian and Heny, Frank W., *An Introduction to the Principles of Transformational Syntax*, The MIT Press - 1975.

[ALI et al. 86] Ali, Yawar, Aubin, Raymond and Hall, Barry, *A Domain-Independant Natural Language Database Interface*, In: Proceedings of the Sixth Canadian Conference on AI (1986), pp.62-66.

[ANDRIOLE 85] Andriole, Stephen J. (editor), *Applications in Artificial Intelligence*, Petrocelli Books - 1985.

[BALLARD & STUMBERGER 86] Ballard, Bruce W. and Stumberger, Douglas E., *Semantic Acquisition in TELI: A Transportable, User-Customized Natural Language Processor*, In: Proceedings of the 24th Annual Meeting of the ACL (1986), pp.20-29.

[BARA & GUIDA 84] Bara, B.G. and Guida, G. (editors), *Computational Models of Natural Language Processing*, Elsevier Science 1984.

[BARRETT et al. 86] Barrett, William A., Bates, Rodney M., Gustafson, David A., Couch, John D., *Compiler Construction: Theory and Practice* (2nd edition), Science Research Associates 1986.

[BATES 78] Bates, Madeleine, *The Theory and Practice of Augmented Transition Network Grammars*, In: L. Bolc (editor), Lecture Notes in Computer Science, Volume 63 pp.191-259, Springer-Verlag.

[BATES & SIDNER 83] Bates, Madeleine, and Sidner, Candace L., *A Case Study of a Method for Determining the Necessary Characteristics of a Natural Language Interface*, In: P.Degano and E. Sandewall (editors), Integrated Interactive Computing Systems, North-Holland - 1983.

[BAYER et al. 85] Bayer, Samuel, Joseph, Leonard and Kalish, Candace, *Grammatical Relations as the Basis for Natural Language Parsing and Text Understanding*, In: IJCAI-85 proceedings, pp.788-790.

[BRADY & BERWICK 83] Brady, Michael and Berwick, Robert C. (editors), *Computational Models of Discourse*, The MIT Press 1983.

[BRUCE 75] Bruce, Bertram, *Case Systems for Natural Language*, Artificial Intelligence, Volume 6, pp.327-360.

[BUNDY 84] Bundy, Alan, *Intelligent Front Ends*, In a book containing selected papers of an England conference on Expert Systems held in 1984.

[CARBONELL et al. 83] Carbonell, Jaime G., Boggs, W. Mark, Mauldin, Michael L., and Anick, Peter G., *The XCALIBUR Project: A Natural Language Interface To Expert Systems*, In: IJCAI-83 proceedings, pp.653-656.

[CARBONELL & HAYES 83] Carbonell, Jaime G. and Hayes, Philip J., *Recovery Strategies for Parsing Extragrammatical Language*, In: AJCL, Volume 9, Number 3-4, pp.123-146.

[CLOCKSIN & MELLISH 81] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, Springer-Verlag 1981.

[CODD 74] Codd, E.F., *Seven Steps to RendezVous with Casual Users*, In: J.W. Klimbie and K.J. Koffeman (editors), Data Base Management, North-Holland 1974.

[COLMERAUER 75] Colmerauer, Alain, *Metamorphosis Grammars*, In: L. Bolc (editor), Lecture Notes in Computer Science, Volume 63, pp.133-189, Springer-Verlag.

[CONDILLAC 86] Condillac, M., *Prolog: Fondements et Applications*, Dunod - 1986.

[DAHL & MCCORD 83] Dahl, Veronica and McCord, Michael C., *Treating Coordination in Logic Grammars*, In: AJCL, Volume 9, Number 2, pp.69-91.

[DAMERAU 85] Damerau, Fred J., *Problems and Some Solutions in Customization of Natural Language Database Front Ends*, In: ACM Transactions on Office Information Systems, Volume 3, Number 2 (special issue), pp.165-184.

[DAVIDSON & KAPLAN 83] Davidson, J. and Kaplan, S. Jerrold, *Natural Language Access to Data Bases: Interpreting Update Requests*, In: AJCL, Volume 9, Number 2, pp.57-68.

[DRAPS 82] Draps, J., *Grammaire de l'Anglais Contemporain*, Didier Hatier 1982.

[DYER 83] Dyer, Michael George, *In-Depth Understanding*, The MIT Press - 1983.

[FARGUES et al. 86] Fargues, Jean, Landau, Marie-Claude, Dugourd, Anne and Catach, Laurent, *Conceptual Graphs for Semantics and Knowledge Processing*, In: IBM J.RES. DEVELOP., Volume 30, Number 1, pp.70-79.

[FILLMORE 68] Fillmore, Charles J., *The Case for Case*, In: E. Bach and R.T. Hams (editors), Universals in Linguistic Theory, Holt, Rinehart, Winston 1968.

[FININ et al. 79] Finin, Tim, Goodman, Bradley, and Tennant, Harry, *JETS: Achieving Completeness Through Coverage and Closure*, In: IJCAI-79 proceedings, pp.275-281.

[FONG & BERWICK 85] Fong, Sandiway and Berwick, Robert C., *New Approaches to Parsing Conjunctions Using Prolog*, In: IJCAI-85 proceedings, pp.870-876.

[GIANNESINI et al. 85] Giannesini, F., Kanoui, H., Pasero, R. and Van Caneghem, M., *Prolog*, InterEditions 1985.

[HARRIS 79] Harris, Larry R., *Experience With ROBOT in 12 Commercial Natural Language Data Base Query Applications*, In: IJCAI-79 proceedings, pp.365-368.

[HARRIS 85] Harris, Mary Dee, *Introduction to Natural Language Processing*, Reston 1985.

[HAYES & CARBONELL 81] Hayes, Philip J. and Carbonell, Jaime G., *Multi-Strategy Construction-Specific Parsing for Data Base Query and Update*, In: IJCAI-81 proceedings, pp.432-439.

[HENDRIX 77] Hendrix, Gary G., *Human Engineering for Applied Natural Language Processing*, In: IJCAI-77 proceedings, pp.183-191.

[HENDRIX 86] Hendrix, Gary G., *Bringing Natural Language Processing to the Microcomputer Market*, In: Proceedings of the 24th Annual Meeting of the ACL (1986), p.2.

[HIRST 84] Graeme, Hirst, *A Semantic Process for Syntactic Disambiguation*, In: AAAI-84 proceedings, pp.148-152.

[JAKOBSON et al. 86] Jakobson, G., Lafond, C., Nyberg, E. and Piatetsky-Shapiro, G., *An Intelligent Database Assistant*, In: IEEE Expert, Volume 1, Number2.

[JARKE & VASSILIOU 85] Jarke, Matthias and Vassiliou, Yannis, *A Framework for Choosing a Database Query Language*, In: ACM Computing Surveys, Volume 17, Number 3, pp.313-340.

[JENSEN et al. 83] Jensen, K., Heidorn, G.E., Miller, L.A. and Ravin, Y., *Parse Fitting and Prose Fixing: Getting a Hold on Ill-Formdness*, In: AJCL, Volume 9, Number 3-4, pp.147-160.

[KING 83] King, Margaret (editor), *Parsing Natural Language*, Academic-Press 1983.

[KLUZNIAK & SZPAKOWICZ 85] Kluzniak, Feliks and Szpakowicz, Stanislaw, *Prolog for Programmers*, Academic Press - 1985.

[KOLODNER 84] Kolodner, Janet L., *Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model*, Lawrence Erlbaum Associates 1984.

[LANGUAGE CRAFT 85] Language Craft Reference Manual (Version 3.0), Carnegie Group Inc - 1985.

[LEECH 81] Leech, Geoffrey, *Semantics*, Penguin Books 1981.

[LEHNERT & RINGLE 82] Lehnert, Wendy G. and Ringle, Martin H., *Strategies for Natural Language Processing*, Lawrence Erlbaum Associates 1982.

[LESMO & TORASSO 85] Lesmo, Leonardo and Torasso, Pietro, *Weighted Interaction of Syntax and Semantics in Natural Language Analysis*, In: IJCAI-85 proceedings, pp.772-778.

[MARCUS 80] Marcus, Mitchell P., *A Theory of Syntactic Recognition for Natural Language*, The MIT Press 1980.

[MARTIN 85] Martin, James, *Fourth-Generation Languages Volume 1 (Principles)*, Prentice-Hall - 1985.

[MELLISH 85] Mellish, C.S., *Computer Interpretation of Natural Language Descriptions*, Ellis Horwood 1985.

[MILNE 86] Milne, Robert, *Resolving Lexical Ambiguity in a Deterministic Parser*, In: AJCL, Volume 12, Number 1, pp.1-12.

[MORIK 83] Morik, Katharina, *Demand and Requirements for Natural Language Systems*, In: IJCAI-83 proceedings, pp.647-649.

[PELLETIER 86] Pelletier, Bertrand, *Systeme d'Interrogation de Banque de Donnees en Langue Naturelle*, Master's Thesis, University of Montreal, March 1986.

[PEREIRA & WARREN 80] Pereira, Fernando C.N. and Warren, David H.D., *Definite Clause Grammars for Language Analysis- A Survey of the Formalism and a Comparison with Augmented Transition Networks*, In: Artificial Intelligence, Volume 13, pp.231-278.

[PEREIRA 81] Pereira, Fernando C.N., *Extraposition Grammars*, In: AJCL, Volume 7, Number 4, pp.243-256.

[QUIZ 85] Quiz User Manual (Version 5.01), Cognos Inc. 1985.

[REICHMAN 85] Reichman, Rachel, *Getting Computers to Talk Like You and Me*, The MIT Press 1985.

[ROSENBERG 80] Rosenberg, Richard S., *Approaching Discourse Computationally: A Review*, In: L. Bolc (editor), Representation and Processing of Natural Language, Carl Hanser Verlag - 1980.

[SCHANK 73] Schank, Roger C., *Conceptual Dependency: A Theory of Natural Language*, In: R.C. Schank and K. Colby (editors), Computer Models of Thought and Language, W.C. Freemand and Company 1973.

[SCHANK 76] Schank, Roger C., *The role of Memory in Language Processing*, In: Charles N. Cofer (editor), The structure of Human Memory, W.H.Freeman 1976.

[SCHANK & RIESBECK 81] Schank. Roger C. and Riesbeck, C.K., *Inside Computer Understanding*, Lawrence Erlbaum Associates 1981.

[SELFRIDGE 86] Selfridge, Mallory, *Integrated Processing Produces Robust Understanding*, In: AJCL, Volume 12, Number 2, pp.89-106.

[SHAPIRO 82] Shapiro, Stuart C., *Generalized Augmented Transition Network Grammars for Generation from Semantic Networks*, In: AJCL, Volume 8, Number 1, pp.12-25.

[SIMMONS 84] Simmons, Robert F., *Computations from the English*, Prentice-Hall 1984.

[SKUCE 82] Skuce, Douglas, *LESK: A User-Friendly Language and System for Expert Knowledge Acquisition*, Technical Report TR-82-06, Department of Computer Science, University of Ottawa, 1982.

[SKUCE 83] Skuce, Douglas, *The LESK Tutorial*, Technical Report TR-83-03, Department of Computer Science, University of Ottawa, 1983.

[SOWA 84] Sowa, John F., *Conceptual Structures*, Addison-Wesley 1984.

[SOWA & WAY 86] Sowa, John F. and Way, Eileen O., *Implementing a Semantic Interpreter Using Conceptual Graphs*, In: IBM J.RES. DEVELOP., Volume 30, Number 1, pp.57-69.

[SPARCK JONES 84] Sparck Jones, Karen, *Natural Language Interfaces for Expert Systems: an Introductory Note*, In a book containing selected papers of an England conference on Expert Systems held in 1984.

[SPARCK JONES & WILKS 85] Sparck Jones, K. and Wilks, Y., *Automatic Natural Language Parsing*, Ellis Horwood 1985.

[SWAN 86] Swan Database System, Natural Language Products 1986.

[TAUZOVICH 87] Tauzovich, Branka, *Representing Causal Relationships in an Expert Advisor for a Fourth Generation Language*, Ph.D. Thesis, Department of Electrical Engineering, University of Ottawa, forthcoming.

[THOMSON & MARTINET 80] Thomson, A.J. and Martinet, A.V., *A Practical English Grammar* (3rd edition), Oxford - 1980.

[WALLACE 84] Wallace, Mark, *Communicating with Databases in Natural Language*, Ellis Horwood 1984.

[WALTZ 78] Waltz, David L., *An English Language Question Answering System for a Large Relational Database*, In: Communications of the ACM, Volume 21, Number 7, pp.526-539.

[WARREN & PEREIRA 82] Warren, David H.D. and Pereira, Fernando C.N., *An Efficient Easily Adaptable System for Interpreting Natural Language Queries*, In: AJCL, Volume 8, Number3-4, pp.110-122.

[WILENSKY et al. 84] Wilensky, Robert, Arens, Yigal and Chin, David, *Talking to Unix in English: An Overview of UC*, In: Communications of the ACM, Volume 27, Number 6, pp.574-593.

[WINOGRAD 83] Winograd, Terry, *Language as a Cognitive Process Volume 1: Syntax*, Addison-Wesley 1983.

APPENDIX A : A SAMPLE OF COMPLEX QUERIES

A SAMPLE OF COMPLEX QUERIES

Report is fine, but I'm getting a data conversion error. Why?

Why does PAGE FOOTING work only if reporting more than 5 detail lines?

How do I change page title but retain the default column headings?

In the following sequence of QUIZ statements:
```
  DEFINE
  SELECT IF
  SORT ON
  DEFINE
```
Is the second DEFINE calculated after the SELECT?

Can I link two files where the keys have different lengths?

How do I print something when the value on which I sort changes?

How do I select from file2 records which do not exist in file1?

Can I undertake a specified action if a certain group of info exceeds one page?

How do I select on a date field so that I will be able to calculate subtotals for each month of 84?

If I have two identical item names in two files linked to each other, how do I say which one I mean in a given statement?

How do I bypass all the records such that a value of a certain detail is not 'R'?

Saving two compiled reports with:
```
  - - -
  BUILD fn1

  BUILD fn2
```
when I EXECUTE fn2, shall that only execute the latter report?

Why, when subscripts are reported to a subfile, does only the first get accessed?

Can the user set up a report as

```
ACCESS A
SELECT IF X=...
REPORT
GO
SELECT
GO
```

Is the first SELECT taken into account when the second report is executed?

Does a SET REP FORMS defer the job until the forms are put on a printer?

How do I set up a way in which the report limit may be variable?

How do I select a record complex if the selection criteria are met or there is no matching record in the other file?

How do I get the record I want when claim_no and claim_line_no together are unique, but only claim_no is the key?

Can I use AVERAGE in a report statement and write it to a subfile?

Can a subfile be linked to a subfile?

How do I subtotal a number and print that subtotal to a subfile at a control break?

Which of the two statements SELECT and SORT gets executed first?

APPENDIX B : QUIZ DOMAIN VERBS

QUIZ DOMAIN VERBS

B1: FUNDAMENTAL VERBS

These verbs have their case structures implemented under this format (see appendix D):
a-) Nominal Group
nomi(case indicator, mandatory/optional flag, filler type, case name)
b-) Case Structure
role_table(verb, roles[Nominal Groups]).

1- access d_o:source file (e.g. file 'F')
1- access with/using:instrument (e.g. ACCESS statement)

1- add d_o:added object (e.g. QUIZ option)
1- add to:recipient object (e.g. ACCESS statement)

1- add after/before:location (e.g. sort key)
1- add d_o:argument1 (e.g. number)
1- add to:argument2 (e.g. number)

1- average at:location (e.g. control break)
1- average d_o:item (e.g. item)
1- average in:location (e.g. heading)

1- choose d_o:object (e.g. keys,values)

1- compare d_o:item1 (e.g. item 'X')
1- compare to:item2 (e.g. item 'Y')
1- compare with:item2 (e.g. item 'Y')

1- count at:location (e.g. control break)
1- count d_o:object (e.g. item)

1- create d_o:object (e.g. report,record complex)

1- define d_o:object (e.g. size of character string)
1- define in:location (e.g. dictionary)
1- define with/using:instrument (e.g. DEFINE statement)

1- direct d_o:report (e.g. report)
1- direct to:device (e.g. printer)

1- display d_o:object (e.g. items)
1- display on:device (e.g. terminal)

1- divide by:argument1 (e.g. item1)
1- divide d_o:argument2 (e.g. item2)

1- execute d_o:object (e.g. QUIZ statement)

1- find d_o:item (e.g. item 'I1')
1- find in:location (e.g. file)
1- find with:QUIZ statement (e.g. SELECT statement)

1- format d_o:object (e.g. report)
1- format with/using:instrument (e.g. LINK option)

1- get d_o:object (e.g. information)

1- link d_o:source file (e.f. file 'F1')
1- link to/with:co-source file (e.g. file 'F2')

1- multiply by:argument1 (e.g. item 'I1')
1- multiply d_o:object2 (e.g. item 'I2')

1- print at/in/on/under:location (e.g. bottom of next page)
1- print d_o:object (e.g. items)
1- print on:device (e.g. screen)

1- read d_o:file (e.g. file 'F9')

1- report at:location (e.g. control break)
1- report d_o:item (e.g. item 'I27')
1- report for:condition (e.g. each item)
1- report from:source file (e.g. two files)
1- report in:location (e.g. column 'X')
1- report in:order (e.g. ascending order)
1- report on:location (e.g. next page)
1- report to:destination (e.g. file 'F22')

1- reset at:location (e.g. control break)
1- reset d_o:item (e.g. item 'I99')
1- reset to/with/using:value (e.g. zero)

1- right justify d_o:item (e.g. items)

1- run d_o:report (e.g. report)

1- save d_o:object (e.g. report)
1- save in:location (e.g. file)

1- select d_o:item (e.g. items)
1- select from:file (e.g. data file)
1- select on:condition (e.g. condition X)

1- set d_o:item (e.g. item 'I45')
1- set to/with/using:value (e.g. valueX)

1- skip after/before:relative location (e.g. line X)
1- skip d_o:object (e.g. line)
1- skip to:direct location (e.g. next line)

1- sort d_o:item (e.g. items)
1- sort in:order (e.g. alphabetical order)
1- sort on:sort field (e.g. last names)

1- specify d_o:item (e.g. item)
1- specify in:location (e.g. CHOOSE statement)

1- start d_o:object (e.g. QUIZ)

1- store d_o:object (e.g. data)
1- store in:location (e.g. file)

1- subtotal at:location (e.g. control break)
1- subtotal d_o:item (e.g. items)
1- subtotal for:condition (e.g. each customer)
1- subtotal in:location (e.g. report column)

1- total: considered equivalent to subtotal

1- use at:location (e.g. end of report group)
1- use d_o:object (e.g. QUIZ statement)

B2: Verbs that have a canonical synonym in B1 or are not considered for prototype 1

2- accept at:time (e.g. execution time)
2- accept d_o:object (e.g. items,values)
2- accept from:source (e.g. user)

2- align d_o:object (e.g. title,columns)
2- align with:format (e.g. heading)

2- append d_o:object1 (e.g. file1)
2- append to:object2 (e.g. file2)
2- append-COMMENT: append to a file, concatenate to an item

2- assign d_o:object1 (e.g. sort-key)
2- assign to:object2 (e.g. value)
2- assign-COMMENT: assign is equivalent to set (set is canonical)

2- begin at:preset value (e.g. preset value)
2- begin d_o:object (e.g. column,editing process,next record)
2- begin from:value (e.g. zero)
2- begin on/with:format (e.g. new page)
2- begin-COMMENT: begin is equivalent to start (start is canonical)

2- build d_o:object (e.g. record complex)
2- build with:instrument (e.g. QUIZ statement)
2- build-COMMENT: 2 meanings; as a synonym of create,
2- build-COMMENT: as compile (vs BUILD stat)

2- bypass d_o:object (e.g. default)
2- bypass-COMMENT: bypass is a synonym of override

2- calculate d_o:object (e.g. item)
2- calculate in:location (e.g. report)
2- calculate until:condition (e.g. end of the list)

2- cancel d_o:object (e.g. statement)

2- change d_o:object (e.g. defaults,value)
2- change-COMMENT: a very general verb

2- check d_o:object (e.g. list)
2- check for:pattern (e.g. key item)
2- check in:location (e.g. file)
2- check with:reference (e.g. data dictionary)
2- check-COMMENT: a very general verb

2- clear d_o:object (e.g. file)

2- close d_o:object (e.g. file)

2- combine d_o:object (e.g. records)
2- combine with:option (e.g. X option)
2- combine-COMMENT: combine records = create a complex

2- compile d_o:object (e.g. report)

2- concatenate d_o:object (e.g. items)
2- concatenate-COMMENT: after N.C. from Cognos, not used in our samples.

2- contain d_o:object (e.g. information)

2- control d_o:object (e.g. destination)
2- control-COMMENT: control is a synonym of direct (direct is canonical)

2- convert d_o:object1 (e.g. date)
2- convert to:object2 (e.g. string)

2- copy d_o:object (e.g. statements)
2- copy into:location (e.g. file)
2- copy to:location (e.g. file)

2- delete
2- delete-COMMENT: a very general verb

2- edit d_o:object (e.g. statements)

2- eliminate d_o:object (e.g. repetitions)
2- eliminate in:location (e.g. report)

2- enter d_o:object (e.g. QUIZ statement,question mark)
2- enter in:format (e.g. upper/lower case)
2- enter on:destination (e.g. system)

2- establish between:objects (e.g. files)
2- establish d_o:object (e.g. link)

2- evaluate d_o:object (e.g. DEFINE statement)

2- exit from:object (e.g. system,editor)

2- expand d_o:object (e.g. report1)
2- expand into:destination (e.g. report2)

2- extend d_o:object (e.g. item X)
2- extend over:format (e.g. X lines)

2- flag d_o:object (e.g. record)

2- group
2- group-COMMENT: could be a synonym of sort (to group = to sort)

2- include after/before/between:location (e.g. between item X and item Y)
2- include d_o:object (e.g. option)
2- include in:location (e.g. DEFINE statement,report)

2- increase d_o:object (e.g. number of X)

2- insert between:location (e.g. parts of the date)
2- insert d_o:object (e.g. separator)

2- issue d_o:object (e.g. error message)

2- join d_o:object1 (e.g. item1)
2- join to:object2 (e.g. item2)
2- join-COMMENT: concatenate is the canonical synonym of join

2- keep d_o:object (e.g. file)
2- keep-COMMENT: a subtle synonym of "not losing" or "saving" a file

2- leave after/before:format (e.g. first line)
2- leave d_o:object1 (e.g. blanks)
2- leave for:object2 (e.g. leave room for 3 lines)

2- list by:condition (e.g. employee number)
2- list d_o:object (e.g. items)
2- list in:sorting order (e.g. alphabetical order)
2- list-COMMENT: a synonym of report (e.g. report is canonical)

2- load d_o:object (e.g. tape)
2- load on:destination (e.g. system)

2- lose d_o:object (e.g. digits,field)

2- make

2- modify d_o:object (e.g. statement)
2- modify-COMMENT: a general verb

2- number d_o:object (e.g. pages)

2- occur at:location (e.g. control break)

2- open d_o:object (e.g. file)

2- order d_o:object (e.g. items)
2- order-COMMENT: sort is the canonical synonym of order

2- output
2- output-COMMENT: used as 'display' or 'report';
2- output-COMMENT: report is the canonical synonym of output

2- override d_o:object (e.g. layout)

2- pad d_o:object1 (e.g. item X)
2- pad with:object2 (e.g. blanks)

2- place d_o:object (e.g. X statement)
2- place in:location (e.g. control footing)

2- position above/under:format (e.g. item Y)
2- position d_o:object (e.g. characters X)

2- present d_o:object (e.g. information)
2- present in:format (e.g. ...)
2- present-COMMENT: could be a synonym of report

2- prevent by:solution (e.g. using X option)
2- prevent d_o:object (e.g. report)
2- prevent from:undesired feature (e.g. printing on two lines)

2- prompt at:time (e.g. execution time)
2- prompt d_o:object1 (e.g. user)
2- prompt for:object2 (e.g. code)

2- redefine

2- remove d_o:object (e.g. blanks)
2- remove from:location (e.g. item X)

2- replace d_o:object (e.g. spaces)
2- replace with:replacer (e.g. character X)

2- reroute
2- reroute-COMMENT: used as a 'special' output;
2- reroute-COMMENT: direct is the canonical synonym

2- retain
2- retain-COMMENT: used as keep; a synonym of keep

2- route
2- route-COMMENT: used as output; direct is the canonical synonym of route

2- send
2- send-COMMENT: used as output; direct is the canonical synonym of send

2- show d_o:object (e.g. items)
2- show-COMMENT: a synonym of display

2- stop d_o:object (e.g. QUIZ,process)
2- stop from:action (e.g. processing)

2- stream
2- stream-COMMENTS: HP terminology

2- suppress d_o:object (e.g. repetitions)
2- suppress with:instrument (e.g. X option)

2- suspend d_o:object (e.g. session)

2- truncate d_o:object (e.g. result of...)

2- type after/before:location (e.g. file name)
2- type at:format (e.g. end of ...)
2- type d_o:object (e.g. QUIZ statement)

2- write
2- write-COMMENT: used as output; a synonym of report

B3: SECONDARY VERBS (relatively infrequent)

3- appear in/on/to:format (e.g. picture)

3- assemble d_o:object (e.g. report-items)
3- assemble into:new-arrangement (e.g. report-group)

3- assume

3- avoid d_o:object (e.g. windowed headings)
3- avoid-COMMENT: in general, could be rephrased

3- belong

3- cause d_o:object (e.g. control footing)

3- code

3- complete d_o:object (e.g. statement)

3- compress d_o:object (e.g. item)
3- compress into:result (e.g. item)

3- conform

3- consist d_o:object1 (e.g. item,report)
3- consist of:object (e.g. report items)

3- correspond

3- customize d_o:object (e.g. report)

3- declare

3- defer

3- depend on:object (e.g. item X)

3- describe d_o:object (e.g. records)

3- design d_o:object (e.g. report)
3- design with:instrument (e.g. sort-keys)

3- destroy d_o:object (e.g. file)

3- detect
3- detect-COMMENT: used as select; check could be more appropriate

3- determine d_o:object (e.g. item X)

3- disable

3- discuss

3- eject

3- enable

3- encounter d_o:object (e.g. statement)

3- ensure

3- exceed

3- expect
3- expect-COMMENT: used only as "expected"

3- facilitate d_o:object (e.g. design)

3- fill

3- fix

3- follow

3- force d_o:object (e.g. page skip)

3- generate d_o:object (e.g. detail)

3- give d_o:object (e.g. instructions,information)

3- guarantee

3- happen

3- hold (e.g. down) d_o:object (e.g. control key)

3- imply

3- incorporate d_o:object (e.g. feature)
3- incorporate in:location (e.g. report)

3- indicate
3- indicate-COMMENT: probably a synonym of flag (e.g. to flag = to indicate)

3- inherit

3- initiate d_o:object (e.g. QUIZ)
3- initiate-COMMENT: start is the canonical synonym of initiate

3- intervene

3- introduce

3- involve

3- like

3- look
3- look-COMMENT: used as find or retrieve

3- manipulate d_o:object (e.g. information)

3- match

3- mean

3- meet
3- meet-COMMENT: only in "criteria are met"

3- miss
3- miss-COMMENT: used as "there is no"

3- name d_o:object (e.g. element,file)
3- name in:QUIZ statement (e.g. SELECT statement)

3- need

3- partition

3- perform d_o:object (e.g. selection)
3- perform on:location (e.g. every item)

3- plan d_o:object (e.g. report)

3- prepare d_o:object (e.g. report)
3- prepare for:destination (e.g. production)

3- press d_o:object (e.g. Y key)

3- process d_o:object (e.g. statements)

3- produce d_o:object (e.g. report)
3- produce for:condition (e.g. each item)

3- program

3- provide d_o:recipient (e.g. user)
3- provide for:object1 (e.g. characters)
3- provide in:location (e.g. items,file)
3- provide with:object2 (e.g. means ...)

3- put
3- put-COMMENT: used as output

3- reduce d_o:object (e.g. processing time)

3- refer to:object (e.g. information)

3- refine d_o:object (e.g. report)

3- relate

3- represent

3- require d_o:object (e.g. decimal point)
3- require d_o:object (e.g. space)
3- require in:location (e.g. report display)

3- reserve

3- respond with:instrument (e.g. message)

3- resume d_o:object (e.g. session)

3- return d_o:object (e.g. report)
3- return d_o:object1 (e.g. item1)
3- return for:object2 (e.g. item2)
3- return to:destination (e.g. terminal)
3- return-COMMENT: used in the manual as a general verb

3- say

3- scale

3- scratch

3- search d_o:object (e.g. file)

3- see

3- seem

3- subscript

3- summarize d_o:object (e.g. information)

3- supply d_o:object (e.g. headings)

3- support
3- support-COMMENT: used as "is there" or "Can I have";
3- support-COMMENT: "Does Quiz support ..."

3- take

3- think

3- try

3- undertake

3- want

3- work with:object (e.g. files)

APPENDIX C : AN EXCERPT FROM THE LEXICON

AN EXCERPT FROM THE LEXICON

C1: ABBREVIATIONS FOR SYNTACTIC CATEGORIES

adj: adjective
adv: adverb
c_noun: common noun
comp_adj: comparative adjective
conj: conjunction
demonstr_adv: demonstrative adverb
inter_adv: interrogative adverb
inter_pro: interrogative pronoun
m_noun: mass noun
num_adj: numeral (see 3.1.2.4.9)
pers_pro: personal pronoun
poss_adj: possessive adjective
prep: preposition
quant_adj: quantifier adjective (see 3.1.2.4.9)
reg_adj: regular adjective
reg_adv: regular adverb
reg_pro: regular pronoun
rel_adj: relative adjective
v_aux: auxiliary verb
v_ger: (verb) gerund form
v_inf: (verb) infinitive form
v_past: (verb) past form
v_ppart: (verb) past participle form
v_pre: (verb) present tense third person singular form

C2: THE EXCERPT

```
a(article,sing,nil,nil,nil).
access(v_inf,_,nil,nil,access).
accessed(v_past,_,nil,nil,access).
accessed(v_ppart,_,nil,nil,access).
accesses(v_pre,sing3,nil,nil,access).
accessing(v_ger,_,nil,nil,access).
an(article,sing,nil,nil,nil).
and(conj,_,nil,nil,nil).
at(prep,_,nil,nil,nil).
can(v_aux,_,nil,nil,nil).
criteria(m_noun,plur,nil,nil,nil).
each(quant_adj,sing,nil,nil,nil).
error(c_noun,sing,[message,messages],nil,nil).
error_message(c_noun,sing,nil,nil,nil).
error_messages(c_noun,plur,nil,nil,nil).
every(quant_adj,sing,nil,nil,nil).
explicitly(reg_adv,_,nil,nil,nil).
first(rel_adj,_,nil,nil,nil).
i(pers_pro,sing1,nil,nil,nil).
larger(comp_adj,_,nil,nil,nil).
my(poss_adj,_,nil,nil,nil).
new(reg_adj,_,nil,nil,nil).
not(particle,_,nil,nil,nil).
nothing(reg_pro,sing,nil,nil,nil).
one(num_adj,sing,nil,nil,nil).
only(reg_adv,_,nil,nil,nil).
or(conj,_,nil,nil,nil).
primary(reg_adj,_,nil,nil,nil).
report(v_inf,_,nil,nil,report).
reported(v_past,_,nil,nil,report).
reported(v_ppart,_,nil,nil,report).
reportes(v_pre,sing3,nil,nil,report).
reporting(v_ger,_,nil,nil,report).
smaller(comp_adj,_,nil,nil,nil).
there(demonstr_adv,_,nil,nil,nil).
to(prep,_,nil,nil,nil).
what(inter_pro,_,nil,nil,nil).
why(inter_adv,_,nil,nil,nil).
will(v_aux,_,nil,nil,nil).
you(pers_pro,sing2,nil,nil,nil).
your(poss_adj,_,nil,nil,nil).
```

APPENDIX D : PROLOG CODE OF SEQAP

Appendix D: Table of Contents

```
/* ----------------------------------------------------------------------
   A modified version of the Quintus READ_SENT package;
   Modifier : Sylvain Delisle
   Project  : SEQAP parser
   File     : read-in.pl
   Version  : November 4, 1986

   This package, called during the lexical phase (by lex-ph.pl), accepts
   an input (which is supposed to be a question, a statement or a phrase)
   as a list of characters from the user and verify its acceptability.
   Each token is one of the following: a punctuation mark, an atom, a varia-
   ble ('normal' or QAUZ), or a string. Invalid characters cause an error.
   The information (category) on every token is preserved in the output.
   This phase also makes sure that the input ends with one of the following
   terminal symbol: '?' (for questions), '.' (for statements) or ';' (for
   phrases).

   EX: Input = I have a file 'F'.
       Output= [atom(i),atom(have),atom(a),atom(file),var('F'),'.']

   For further details see comments below.
----------------------------------------------------------------------*/


%   Package: read_sent
%   Author : R.A.O'Keefe
%   Updated: 9/9/85
%   Purpose: to provide a flexible input facility

:- mode
        chars_to_atom(-, ?, ?),
        chars_to_string(+, -, ?, ?),
        chars_to_words(+, -),
        chars_to_words(-, ?, ?),
        chars_to_word(-, ?, ?),
        read_line(-),
        read_until(+, -),
        read_until(+, +, -).

:- ensure_loaded([
        library(basics),
        library(ctypes),
        library(lists)]).



/******************************** READ_UNTIL ********************************/

/*  read_until(Delimiters, Answer)
    reads characters from the current input until  a  character  in  the
    Delimiters  string  is  read.  The characters are accumulated in the
    Answer string, and include the closing  delimiter.  The end of file
    character varies from Prolog to Prolog.  Dec-10 Prolog and C Prolog
    use ^Z (regardless of what key you actually press), while Quintus
    Prolog uses -1 (whatever you press).  The library predicate is_endfile
    conceals this difference.  The end of file character is always a
    delimiter, even if it is not in the list of characters you supply.
*/
read_until(Delimiters, [Char|Rest]) :-
        get0(Char),
        read_until(Char, Delimiters, Rest).

read_until(Char, _, []) :-
        is_endfile(Char),
        !.

read_until(Char, Delimiters, []) :-
        memberchk(Char, Delimiters), !.

read_until(_, Delimiters, Rest) :-
        read_until(Delimiters, Rest).



/****************************** CHARS_TO_WORDS ******************************/

/*  chars_to_words(Chars, Words)
    parses a list of characters (read by read_until) into a list of
    tokens, where a token is
        'X' for X a period or other punctuation mark;
        atom(X) for X a sequence of letters, e.g. atom(the);
        pn_qn(X) for X a proper noun or a Quiz noun (starts with cap. letter);
        qauz_var(X) for X a QAUZ variable (starts with a period);
        var(X) for X a (normal) variable (in single quotes);
        string(X) for X "..sequence of any..' (This will be Quiz stuff).
*/
chars_to_words(Chars, Words) :-
        chars_to_words(Words, Chars, []).

chars_to_words([Word|Words]) -->
        chars_to_word(Word), !,
        chars_to_words(Words).

chars_to_words([]) --> [].


chars_to_word(Word) -->
        [Char], {Char =< 32}, !,  % 32=space, Char<32 = control character.
        chars_to_word(Word).

chars_to_word(atom(Word)) -->
        [Char], {Char=73, Char2=105}, !,  % 73=I, 105=i.
        chars_to_atom([]), {name(Word,[Char2])}.

chars_to_word(atom(Word)) -->
        [Char],  % 97-122=small letters, 48-57=digits.
        {(Char > 96, Char < 123); (Char > 47, Char < 58)}, !,
        chars_to_atom(Chars),
        {name(Word, [Char|Chars])}.

chars_to_word(pn_qn(Word)) -->
        [Char], {Char > 64, Char < 91}, !,  % 65-90=capital letters.
        chars_to_atom(Chars),
        {name(Word, [Char|Chars])}.

chars_to_word(var(Word)) -->
        [Char], {Char=39}, !,  % 39=single quote.
        chars_to_atom(Chars), !,
        {reverse(Chars,Chars2),
         nth1(1,Chars2,39),  % the token must also end with a single quote
         append([39],Chars3,Chars2), reverse(Chars3,Chars_OK)),
        {name(Word2, Chars_OK), append([Word2],[],Word3), nth1(1,Word3,Word)}.

chars_to_word(qauz_var(Word)) -->
        [Char], {Char=46},  % a QAUZ variable starts with a period.
        chars_to_atom2(Chars), {Chars\==[]},  % checks not only a period !
        {name(Word, [Char|Chars])}.

chars_to_word(string(Word)) -->
        [34 /* " */], !,  % 34=double quotes.
        chars_to_string(34 /* " */, String),
        {name(Word, String)}.
```

```
chars_to_word(Punct) -->
        [Char],
        {name(Punct, [Char])}.



/******************************** CHARS_TO_ATOM ******************************/

/*  chars_to_atom(Tail)
    reads the remaining characters of a word.  Case conversion  is  left
    to  another  routine.   In this application, a word may only contain
    letters but they may be in either case.  If you want to parse French
    you will have to decide what to do about accents.  I suggest putting
    them after the vowel, and adding a clause
        chars_to_atom([Vowel,Accent|Chars]) -->
                [Vowel],        {accentable_vowel(Vowel)},
                [Accent],       {accent_for(Vowel, Accent)},
                !.
        with the obvious definitions of accentable_vowel and accent_for.
    Note that the Ascii characters ' ` ^ are officially  designated  the
    "accent acute", "accent grave", and "circumflex".  But this file was
    originally written for an English parser and there was no problem.

   Characters allowed in an atom (from character#2 to character#n);
   letters, digits, single quotes(39), hyphens(45), periods(46), slashes(47)
   and underscores(95).
*/
chars_to_atom([Char|Chars]) -->
        [Char],
        {is_alnum(Char) ; member(Char,[39,45,46,47,95])}, !,
        chars_to_atom(Chars).

chars_to_atom([]) --> [].


chars_to_atom2([Char|Chars]) -->
        [Char],   % QAUZ variables: letters + periods + slashes.
        {(Char > 64, Char < 91) ; (Char > 96, Char < 123) ;
         Char=46 ; Char=47}, !,
        chars_to_atom2(Chars).

chars_to_atom2([]) --> [].



/****************************** CHARS_TO_STRING ****************************/

/*  chars_to_string(Quote, String)
    reads the rest of a string which was opened by  a  Quote  character.
    The  string is expected to end with a Quote as well.  If there isn't
    a matching Quote, the attempt to parse the  string  will  FAIL,  and
    thus the whole parse will FAIL.  I would prefer to give some sort of
    error  message and try to recover but that is application dependent.
    Two adjacent Quotes are taken as one, as they are in Prolog itself.
*/
chars_to_string(Quote, [Quote|String]) -->
        [Quote,Quote], !,
        chars_to_string(Quote, String).

chars_to_string(Quote, []) -->
        [Quote], !.

chars_to_string(Quote, [Char|String]) -->
        [Char], !,
        chars_to_string(Quote, String).
```

```
/******************************** READ_LINE ********************************/

/*  read_line(Chars)
    reads characters up to the next newline or the end of the file, and
    returns them in a list, including the newline or end of file.  When
    you want multiple spaces crushed out, and the newline dropped, that
    is most of the time, call trim_blanks on the result.
*/
read_line(Chars) :- is_newline(NL), read_until([NL], Chars).



/******************************** MAIN ADDITIONS ***************************/

/*REMOVE_NEW_LINE_CHAR: removes the end of input signal (new line = 10). */
remove_new_line_char(X,Y) :- reverse(X,[X2|X2_Rest]), X2=10, nth1(1,X2_Rest,X3),
                             ((X3\==32, Out=X2_Rest);
                              (X3=32, remove_blanks_nl(X2_Rest,Out))),
                             reverse(Out,Y).
remove_new_line_char(X,X).


remove_blanks_nl([H|R],[H|R]) :- H\==32.
remove_blanks_nl([H|R],R2) :- H=32, remove_blanks_nl(R,R2).


/* UPPER_TO_LOWER: used only for the very first letter of the input. */
upper_to_lower(X,Y) :- X > 64, X < 91, Y is X + 32.
upper_to_lower(X,X).


/* CHECK_ENDING: verifies that the last character is one of [. ; ?]
                 and that it is preceeded by a blank (space=32). */
check_ending(Mode,X,Y)
      :- reverse(X,[X1|X_Rest]),
         (((X1=46 ; X1=63 ; X1=59), nth1(1,X_Rest,Sec_Last),
           ((Sec_Last\==32,
             append([32],X_Rest,X_Rest2), Y2=[X1|X_Rest2]);
            (Y2=[X1|X_Rest]))));
          ((Mode=translator, Y=fatal_error);
           ((nl, write('Incomplete end of input: select one of'),
            get_end(X1_new),
            ((X1\==32, append([X1_new],[32],X1_OK),
              append(X1_OK,[X1],X1_OK2),
              append(X1_OK2,X_Rest,Y2));
             append([X1_new],[X1|X_Rest],Y2))))))),
        ((Y\==fatal_error, reverse(Y2,Y)) ; (!, fail)).


/* GET_END: prompts the user to get a legal ending for his input. */
get_end(End)
     :- nl, write('1: end of statement (.) OR'),
        nl, write('2: end of question  (?) OR'),
        nl, write('3: end of phrase    (;).'),
        nl, write('Please, enter 1. or 2. or 3.. '), nl, read(X),
        (((X=1, End=46) ; (X=2, End=63) ; (X=3, End=59)) , get_end(End))



/* MAIN PROCEDURE: READ_IN */

read_in(Mode,String,Words_OK)
     :- ((Mode=translator, String=[Char|Chars2]);
```

```
        read_line([Char|Chars]), !, remove_new_line_char(Chars,Chars2), !
        Chars2=[Second_Char|_],
        ((Second_Char > 64, Second_Char < 91, Char2=Char)
         upper_to_lower(Char,Char2)), !,
        check_ending(Mode,[Char2|Chars2],Chars_OK), !,
        chars_to_words(Chars_OK,Words_OK)

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

```
/* ----------------------------------------------------------------------------
   Author   Sylvain Delisle
   Project  SEQAP parser
   File     lex-ph.pl
   Version  November 4, 1986

   This file contains the LEXICAL ANALYSER for the SEQAP parser
   It contains its own 'micro-interface'

   The reader will find detailed explanations about lexical convention
   and the general design of this phase in what follows
---------------------------------------------------------------------------- - --*/


/*****************************************************************************

   SPECIAL CHARACTERS ALLOWED IN THE INPUT

   Definition  an atom is a sequence of (consecutive) characters where each
               one of them is a  letter  digit, single quote  hyphen  period
               slash, underscore  Atoms are separated by spaces  The lexical
               phase will read in characters and assemble them into atoms,
               according to the following rules


   1- SINGLE QUOTES  to denote a 'logical' variable, that is a name used by
      the user and treated as a variable  A legal variable name is simply an
      atom in single quotes
      EX  'F1', 'file_emp', 'element99', 'F12', 'Sort3',


   2- CAPITAL LETTERS  to denote a proper noun or a Quiz noun
      Proper nouns and Quiz nouns are atoms and stored as such in the lexi-
      con  If a noun contains capital letters or underscores or any other
      characters, the noun with capital letters must reproduce it perfectly to
      refer to the very same noun
      EX  REPORT, YYMMDD, SELECT,


   3- SQUARE BRACKETS  to make explicit pps attachments  A pair of square
      brackets must be put around a complete noun phrase (specifier + noun
      group + prep  phrase) when the prep  phrase modifies the noun group that
      is located to its left instead of the verb
      The default is that a prep  phrase modifies the verb, thus square bra-
      ckets change this default
      EX  Hdi report a numeric item to a subfile ?
          This one doesn't need square brackets because the prep  phrase
          "to a subfile" modifies the verb "report"
      EX  Hdi report [a numeric item with leading zeroes] ?
          This second example needs square brackets because the prep  phrase
          "with leading zeroes" modifies the noun group "numeric item"


   4- DOUBLE QUOTES  to delimit Quiz error messages (or any other Quiz
      'stuff', meaning error messages or code, that the parser won't touch)
       EX  (error message)      "Cannot find NAME OF BRANCHES"
           (Quiz code)      "ACCESS EMPLOYEES"


   5- PERIOD  to terminate a statement
      EX  I have a file 'F1'


   6- SEMICOLON  to terminate a phrase
      EX  the file 'F3',
```

```
   7- QUESTION  MARK: to terminate a question.
      EX: Why do I get the error message " ... " ?


   8- UNDERSCORE: to 'link' compound words.
      Its use is OPTIONAL for common nouns and mass nouns.
      For proper nouns and Quiz nouns it must be used.
      EX: data_file or data file


   9- THE WORD unspec: as requested by Doug, this reserved word will be used
      in certain DM-slots when the specifier is left unspecified.


   10- QAUZ VARIABLES:   these will have their own notation.
       A QAUZ variable starts with a period and all other characters are
       capital letters, slashes and periods.
       EX: .ITEM , .ITEM/.RITEM

*****************************************************************************/

/* Lexical Phase Driver */


/* The lexical phase is composed of four different steps:
   1- check input characters;
   2- build the mini lexicon;
   3- find compound words;
   4- replace canonical synonyms.

   The output of the lexical analyser consists of five elements:
   1- the original input as processed by the read-in.pl procedure;
   2- the canonical input where words having canonical synonyms have
      been replaced by them;
   3- the synonym mapping list where every word having been replaced by its
      canonical synonym will be paired with the latter;
   4- type of input flag (statement, question or phrase);
   5- the status flag indicating whether the lexical phase completed
      normally (Status=error OR success).

   The mini lexicon is a side-effect output of the lexical analysis, refer
   to the BUILD_MINI_LEX procedure below.
*/


/********************************* LEX_PHASE *********************************/

lex_phase(Mode,String,Original_Input,Canonical_Input,
          SynMapList,Input_Type,Status)
      :- clean_up_mini_lex,

         % PROMPTING THE USER (not in 'translator' mode)
         ((Mode\==translator,
          nl, write('Enter your input '),
          write('[question(?) OR statement(.) OR phrase(;)]:'), nl) , true),

         (% LET'S ANALYSE HIS INPUT;
          (% READ IT and CHECK MATCHING OF SQUARE BRACKETS
           read_in(Mode,String,Original_Input),
           check_delimitators(Original_Input),

           % CHECK INPUT CHARACTERS
           check_atoms(Mode,Original_Input,[],Verified_Input1),

           % IF THE INPUT IS NOT ACCEPTABLE THEN STOP RIGHT HERE
           (Verified_Input1=error, Status=error(invalid_character));

           % ELSE THE INPUT IS OK THEN BUILD THE MINI LEXICON
           (build_mini_lex(Mode,Verified_Input1,
                           [],Skipped_Out,[],Replace_Out,
                           Missing_Word_Flag),

            (% IS EVERY WORD FOUND IN THE INPUT KNOWN TO THE LEXICON ??
             (% YES
              var(Missing_Word_Flag),

              % REMOVE WORDS THAT THE USER WANTS TO SKIP
              remove_skipped_words(Skipped_Out,Verified_Input1,Verified_Input2),

              % REMOVE WORDS THAT THE USER HAS REPLACED
              remove_replaced_words(Replace_Out,Verified_Input2,Verified_Input3),

              % FIND COMPOUNDS
              find_compounds(Verified_Input3,[],Verified_Input4),

              % REPLACE CANONICAL SYNONYMS
              find_can_syns(Mode,Verified_Input4,[],Canonical_Input,[],
                            SynMapList,Input_Type,S),
              (% ERROR IN FIND_CAN_SYNS
               (var(S), Status=error(missing_canonical_synonym));
               % LEXICAL PHASE COMPLETED SUCCESSFULLY
               Status=success));

             (% ERROR: AT LEAST ONE UNKNOWN WORD CANNOT BE REPLACED/REMOVED
              Status=error(unknown_word),
              ((Mode\==translator,
               write('If necessary have the missing word added to the lexicon.')
               ,nl) ; true))))));

          (% ERROR: FATAL TYPO FOUND IN THE INPUT
           ((Mode\==translator,
            nl, write('Lexical analysis stopped: fatal typo in the input.'), nl,
            write('Check quotes, square brackets and invalid characters.'),
            nl) ; true), Status=error(typo(quotes,brackets,'./?/;',...)))).


/***************************** REMOVE_SKIPPED_WORDS ***************************/
/* Removes unknown words that the user wants to skip.                         */

remove_skipped_words([],Input,Input).
remove_skipped_words([Head|Rest],Input1,Input2)
     :- delete(Input1,Head,Input12),
        append([pn_qn],[Head],Head2), Head3=..Head2,
        delete(Input12,Head3,Input123),
        remove_skipped_words(Rest,Input123,Input2).


/************************** REMOVE_REPLACED_WORDS *************************/
/* Removes words that the user has replaced by other ones.                */

remove_replaced_words([],Input,Input).
remove_replaced_words([Head|Rest],Input1,Input2)
     :- Head=[Old,New], select(Old,Input1,New,Input12),
        remove_replaced_words(Rest,Input12,Input2).
```

```
/******************************** CHECK_ATOMS ******************************/
/* Verifies input characters.                                              */

check_atoms(Mode,[Head|Rest],Current,Final)
     :- functor(Head,Name,1),
        (% 'NORMAL' TOKEN (WORD OR NUMBER)
         (Name=atom, Head=..Head2, nth1(2,Head2,Atom),
          append([Atom],Current,Current2));
         (% STRING TOKEN
          Name=string, Head=..Head2, nth1(2,Head2,Atom),
          append([quiz_stuff(Atom)],Current,Current2));
         (% 'SPECIAL' TOKEN: VARIABLE OR PROPER_QUIZ_NOUN
          (Name=var ; Name=pn_qn ; Name=qauz_var),
          append([Head],Current,Current2))),
        check_atoms(Mode,Rest,Current2,Final).

check_atoms(Mode,[Head|Rest],Current,Final)
     :- functor(Head,Name,0),
        ((% SQUARE BRACKETS
          (Name='[' ; Name=']'),
          append([Name],Current,Current2),
          check_atoms(Mode,Rest,Current2,Final));
         (% TERMINAL CHARACTERS
          (Name='.' ; Name=';' ; Name='?'), Rest=[],
          append([Name],Current,Current2), reverse(Current2,Final));
         (% ERROR: INVALID CHARACTER
          ((Mode\==translator,
           nl, write('Lexical analysis stopped: '),
           write('the character '), write(Name), write(' is invalid.'),
           nl, write('Please resubmit your corrected input.'), nl) ; true),
          Final=error)).



/****************************** BUILD_MINI_LEX ****************************/
/* Once the input is validated, we build the mini lexicon by fetching the  */
/* corresponding entry in the SEQAP lexicon and asserting it in the current*/
/* environment (side-effect).                                              */

build_mini_lex(_,[X|_],Y,Y,Z,Z,_) :- X='.' ; X=';' ; X='?'

build_mini_lex(Mode,[Head|Rest],
               Skipped_In,Skipped_Out,Replace_In,Replace_Out,
               Missing_Word_Flag)
     :- (((% 'Head' IS AN ATOM (REGULAR TOKEN)
           word(Head), Head\==[', Head\==']', Head\==unspec, Word=Head,
           (mini_lex(Word,_,_,_,_,_) ; get_entries(Word)));
          (% 'Head' is A PROPER_QUIZ_NOUN
           functor(Head,pn_qn,1),
           Head=..Head2, nth1(2,Head2,Word),
           (mini_lex(Word,_,_,_,_,_) ; get_entries(Word))));
         ((% 'Word' IS INDEED IN THE LEXICON, LET'S CONTINUE
           mini_lex(Word,_,_,_,_,_),
           build_mini_lex(Mode,Rest,
                          Skipped_In,Skipped_Out,Replace_In,Replace_Out,
                          Missing_Word_Flag));
          (% PROBLEM: 'Word' IS NOT IN THE LEXICON (UNKNOWN)
           ((Mode\==translator,
            nl, write('Lexical analysis stopped: the word |'),
            write(Word), write('| is not in the lexicon.'), nl,
            get_user_fb(Word,Resume_Flag),
            (% LET'S SEE IF THE USER WANTS TO STOP, SKIP IT OR REPLACE IT
             (% HE WANTS TO SKIP IT
              Resume_Flag=2,
              append([Word],Skipped_In,Skipped_In2),
              build_mini_lex(Mode,Rest,
                             Skipped_In2,Skipped_Out,Replace_In,Replace_Out,
                             Missing_Word_Flag));
             (% HE WANTS TO STOP RIGHT HERE
              Resume_Flag=1, Missing_Word_Flag=true);
             (% HE WANTS TO REPLACE IT
              append([[Word,Resume_Flag]],Replace_In,Replace_In2),
              build_mini_lex(Mode,[Resume_Flag|Rest],
                             Skipped_In,Skipped_Out,Replace_In2,Replace_Out,
                             Missing_Word_Flag))) ; Missing_Word_Flag=true)))
          (% THIS TOKEN IS NOT A WORD, LET'S SKIP IT
           not_word(Head),
           build_mini_lex(Mode,Rest,
                          Skipped_In,Skipped_Out,
                          Replace_In,Replace_Out,
                          Missing_Word_Flag))).



/****************************** GET_USER_FB ******************************/

get_user_fb(W,X)
     :- write('Select one of the following options:'), nl,
        write('1: Stop right here  OR'), nl,
        write('2: Skip the missing word and continue  OR'), nl,
        write('3: Replace the missing word.'), nl,
        write('Please, enter 1. or 2. or 3.: '), nl, read(Y),
        (((Y=1 ; Y=2), X=Y);
         (Y=3, get_new_word(W,X));
         get_user_fb(W,X)).


get_new_word(W,X)
     :- write('Enter you replacement: '), nl, read(Y),
        write('Do you want to replace |'), write(W),
        write('| by |'), write(Y), write('| ? (y. or n.)'), nl, read(Z),
        ((Z=y, X=Y) ; get_new_word(W,X)).



/****************************** FIND_COMPOUNDS ****************************/
/* After having built the mini lexicon we check for compound words.        */
/* EX: " ..data file ..." becomes "... data_file ..."                      */

find_compounds([X|_],Inter_CI_in,Inter_CI_out)
     :- (X='.' ; X=';' ; X='?'),
        append([X],Inter_CI_in,Inter_CI_in2),
        reverse(Inter_CI_in2,Inter_CI_out).

find_compounds([Head|Rest],Inter_CI_in,Inter_CI_out)
     :- word(Head), Head\==[', Head\==']', Head\==unspec,
        mini_lex(Head,Cat,_,_,_,_),
        ((% CHECK FOR COMPOUNDS
          member(Cat,[c_noun,m_noun,reg_adj,v_ger,v_inf,v_past,v_ppart,v_pre]),
          nth1(1,Rest,Next), mini_lex(Next,Cat2,_,_,_,_),
          member(Cat2,[c_noun,m_noun,reg_adj,v_ger,v_inf,v_past,v_ppart,v_pre]),
          compound(Head,Next,Word2), lookup_lex(Word2,Cat2,B2,C2,D2,E2),
          append([Next],Rest2,Rest), append([Word2],Inter_CI_in,Inter_CI_in2),
          Entry=..[mini_lex,Word2,Cat2,B2,C2,D2,E2], assert(Entry),
          find_compounds(Rest2,Inter_CI_in2,Inter_CI_out));
         (% IT'S NOT A COMPOUND
          append([Head],Inter_CI_in,Inter_CI_in2),
          find_compounds(Rest,Inter_CI_in2,Inter_CI_out))).

find_compounds([Head|Rest],Inter_CI_in,Inter_CI_out)
     :- not_word(Head),
        append([Head],Inter_CI_in,Inter_CI_in2),
```

```
        find_compounds(Rest,Inter_CI_in2,Inter_CI_out).


/****************************** FIND_CAN_SYNS ******************************/
/* After having found compound words we replace non-canonical words by their */
/* canonical form. This must be specified in the lexicon, of course.         */

find_can_syns(_,[X|_],CI_in,CI_out,SML_in,SML_out,Input_Type,true)
     :- ((X='.', Input_Type=statement);
         (X=';', Input_Type=phrase);
         (X='?', Input_Type=question)),
        append([X],CI_in,CI_in2), append([X],CI_in3,CI_in2),
        reverse(CI_in3,CI_out), reverse(SML_in,SML_out).

find_can_syns(Mode,[Head|Rest],CI_in,CI_out,SML_in,SML_out,Input_Type,Status)
     :- word(Head), Head\==']', Head\==unspec,
        mini_lex(Head,Cat_CS,_,_,CS,_),
        (% THE WORD 'Head' DOESN'T HAVE A CANONICAL SYNONYM
         (CS=nil, append([Head],CI_in,CI_in2),
         find_can_syns(Mode,Rest,CI_in2,CI_out,SML_in,SML_out,Input_Type,Status))
        (% THE WORD 'Head' DOES HAVE A CANONICAL SYNONYM
         lookup_lex(CS,Cat_CS,B_CS,C_CS,CS_CS,E_CS),
         append([CS],CI_in,CI_in2), append([[Head,CS]],SML_in,SML_in2),
         Entry=..[mini_lex,CS,Cat_CS,B_CS,C_CS,CS_CS,E_CS], assert(Entry),
         find_can_syns(Mode,Rest,CI_in2,CI_out,SML_in2,SML_out,Input_Type,Status)
        (% ERROR: THE CANONICAL SYNONYM OF 'Head' DOESN'T EXIST !!
         ((Mode\==translator,
          nl, write('Lexical analysis stopped:'), nl,
          write('the canonical synonym of |'), write(Head),
          write('| is |'), write(CS), write('|,'),
          write(' but |'), write(CS), write('| is not in the lexicon.'), nl);
          true))).

find_can_syns(Mode,[Head|Rest],CI_in,CI_out,SML_in,SML_out,Input_Type,Status)
     :- not_word(Head),
        append([Head],CI_in,CI_in2),
        find_can_syns(Mode,Rest,CI_in2,CI_out,SML_in,SML_out,Input_Type,Status).



/***************************** Home made utilities *****************************/

/* WORD: verifies whether an atom X is a valid word (starts with a letter). */
word(X) :- atom_num(X,false).


/* NOT_WORD: verifies whether an atom X is not a word (numbers, ...). */
not_word(X) :- functor(X,_,1) ;
               X='[' ; X=']' ;
               X=unspec ; atom_num(X,true).


/* ATOM_NUM: checks whether an atom X is a word or a number. */
atom_num(X,T) :- name(X,X_Chars), nth1(1,X_Chars,First),
                 ((T=true, First > 47, First < 58);  % 48-57=digits
                  (T=false, (First < 48 ; First > 57))).


/* CHECK_DELIMITATOR: verifies the pairing of square brackets. */
check_delimitators(OI) :-  check_pairing(OI,sb,0,0).

check_pairing([H|_],_,Num_left,Num_right)
     :- (H='.' ; H=';' ; H='?'), !, Num_left=Num_right.

check_pairing([H|R],Delimitator,Num_left,Num_right)
     :- ((Delimitator=sb,  % sb=square bracket
          ((H='[', Num_left2 is Num_left+1, Num_right2=Num_right);
           (H=']', Num_right2 is Num_right+1, Num_left2=Num_left)),
          !, check_pairing(R,Delimitator,Num_left2,Num_right2));
         !, check_pairing(R,Delimitator,Num_left,Num_right)).


/* CLEAN_UP_MINI_LEX: before a new input erase old mini lexicon entries. */
clean_up_mini_lex :- retract(mini_lex(_,_,_,_,_,_)), fail.
clean_up_mini_lex.


/* GET_ENTRIES: fetches a specific word entry in the SEQAP lexicon and
                asserts it in the environment (side-effect). */
get_entries(Word) :- lookup_lex(Word,A,B,C,CS,E),
                     Entry=..[mini_lex,Word,A,B,C,CS,E], assert(Entry), fail
get_entries(_).


/* LOOKUP_LEX: fetches a specific word entry in the SEQAP lexicon. */
lookup_lex(Word,Cat,B,C,CS,E) :- Y=..[Word,Cat,B,C,CS,E], call(Y).


/* COMPOUND: concatenate two words with the underscore.
             EX: compound(data,file,data_file) is true. */
compound(X,Y,W) :- name(X,Xchars), name(Y,Ychars),
                   append(Xchars,[95],X2), append(X2,Ychars,W0), name(W,W0).


/* SHOW_MINI_LEX: for verification purposes; prints the content of the
                  current mini lexicon. */
show_mini_lex :- mini_lex(W,A,B,C,CS,E),
                 write(W), write(' : '), write(A), write(', '),
                 write(B), write(', '), write(C), write(', '),
                 write(CS), write(', '), write(E), nl, fail.
show_mini_lex.


/* SHOW_VAR_TABLE: for verification purposes; prints the content of the
                   current variable table (user's variables). */
show_var_table :- var_table(S,A,NT,N,V,P),
                  write(V), write(' : '), write(S), write(', '),
                  write(A), write(', '), write(NT), write(', '),
                  write(N), write(', '), write(P), nl, fail.
show_var_table.

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

```
/* ------------------------------------------------------------------------
   Author : Sylvain Delisle
   Project: SEQAP parser
   File   : syn-ph.pl
   Version: November 11, 1986

   This file contains the SEQAP parser.
   The parser is called for each individual entry, be it a question,
   a statement or a phrase.
   The parsing process is composed of two different steps:
   (Three if we include the post-processing (semantic check or formatting))

   1- the LEXICAL ANALYSIS (see lex-ph.pl for details) during which the
      user input is analysed at the lexical level only. This first phase
      also builds the mini-lexicon that will be used by the second step
      of the parsing. The mini-lexicon contains only the information on
      words found in the input. Thus the parser doesn't have to backtrack
      through the huge lexicon but only through the mini-lexicon.

   2- the SYNTACTIC ANALYSIS takes as its input the output of the first
      phase and produces a representation based on roles (except for cer-
      tain types of input) instead of only a parsing tree.
      Notice though that the parser is not role-driven. It tries to match
      the main verb's attributes (direct object and prepositional phrases)
      to its roles. (Basically, a role is a name for describing the relation
      of a particular attribute to the main verb.)
      Information on roles is kept in a table in which verbs are associated
      with their corresponding set of roles. The latter provides the crite-
      rion by which the SEMANTIC component can determine whether a certain role
      is actually filled or not. This criterion is either positional (direct
      object) or a given preposition.
      Every role is either mandatory or optional.


The elements of the output are:

   *Lexical Phase*
    OI : Original Input as given by the user;
    CI : Canonical Input where compounds and canonical synonyms found
         in OI have been replaced;
    SML: Synonym Mapping List is a list of pairs where a word given
         by the user is paired with its canonical synonym when the
         latter is different from the first one;
    IT : Input Type is question, statement or phrase;
    S1 : Success 1 is success if the lexical phase was OK, else is error;
    RT1: Runtime for lexical phase.

   *Parsing*
    QAS: Question answering system (hdi, qauz, or seqap);
    PO : Parser's Output for CI;
    QS : Quiz Stuff found in OI;
    S2 : Success 2 is success if the parsing was OK, else is error;
    RT2: Runtime for the rest of the parsing.


QUESTION
A question is represented as follows:
q(QT,Pos,MVinf,MVmods,Roles).

where QT is the question type (ex: why_1, hdi, syn_1, ...);
      Pos is the positivity (false if the verb is negated);
      MVinf is the infinitive form of the main verb;
      Roles is the list of prepositions with their role names.


STATEMENT
A statement is represented as follows:
s(ST(Pos,Representation)).

where ST is the statement type (is_adj, is_a, assign, comparison, have,
                                there_is, passive and regular);
      Pos is the positivity (false if the verb is negated);
      Representation is specific to each statement:

      is_adj: <Noun_Phrase|Var> is <Adj>
              Representation=Noun_Phrase|Var,Adj;
      is_a: <Var|Proper_Quiz_Noun> is <Noun_Phrase>
            Representation=Var,Noun_Phrase;
            Note: Proper_Quiz_Noun is a proper-noun or a Quiz-noun.
                  A Quiz-noun (ex: SELECT, DATE, ...) is a special proper-noun;
      assign: the <Noun_Phrase1> of <Var|Noun_Phrase2> is <Noun_Phrase3>
              Representation=(the,Noun_Phrase1),(of,Noun_Phrase2),
                             (is,Noun_Phrase3);
      comparison: the <Noun_Phrase1> is Compar_op than <Noun_Phrase2>
                  Representation=Noun_Pharse1,Compar_op,Noun_Pharse2;
      have: <Noun_Phrase1> have <Noun_Phrase2>
            Representation=Noun_Phrase1,Noun_Phrase2;
      there_is: there is <Noun_Phrase>
                Representation=Noun_Phrase;
      passive: <Noun_Phrase1> is Past-Participle-Verb <Noun_Phrase2> <Pps>
               Representation=MVinf,Roles;
      regular: <Noun_Phrase1> Verb <Noun_Phrase2> <Pps>
               Representation=MVinf,Roles.


PHRASE
A phrase is Represented as follows:
p(PT(Representation)).

where PT is the phrase type (regular_np, gerund_np, vp);
      Representation is np_args(...) for regular_np,
                        Pos,MVinf,Roles for the other ones.


NOUN PHRASE
A noun phrase is represented as follows:
np_args(_,spec(Spec),ng(Adjs,NounType,Noun,var(Var),Pps)).

where the first underscore will be the role name;
      Spec is the specifier (déterminer, quantifier, ...) of the noun-group ng;
      Adjs is the list of adjectives modifying the noun;
      NounType is the noun type (common noun, proper_quiz_noun, ...) or the
               type of what could replace a normal ng (pronoun, number, ...);
      Noun is the noun itself (or an equivalent like pronoun, ...);
      Var is the variable name used by the user for denoting this ng;
      Pps is a list of prepositional phrases modifying the ng.

Also, for denoting the subject and the direct object we have created two
pseudo-prepositions: sub (for subject) and d_o (for direct object).
These will be represented as: np_pp(sub,...) or np_pp(d_o,...).


PREPOSITIONAL PHRASES
A prepositional phrase is represented as follows:
```

```
pp(Prep,np_args(...)).

where Prep is instantiated to the preposition (ex: to, in, at, ...).

-------------------------------------------------------------------------------*/


/********************** Initial call: (TOP LEVEL DRIVER) **********************/

parser(Canonical_Input,Input_Type,                % From: Lexical Phase
       Reset, QAS,                                % From: User Interface
       Parser_Output,Quiz_Stuff,Parser_Status)    % Parser's output

    :- clean_up_quiz_stuff,        % Done for every new input
       clean_up_var_table(Reset), % Done only when new query
       call_grammar_subset(Input_Type,QAS,Canonical_Input,
                           Parser_Output,ERR,Canonical_Input,[]),
       ((var(ERR), get_quiz_stuff(Quiz_Stuff), Parser_Status=success);
        Parser_Status=ERR).
       % QAS is a flag indicating the Question Answering System,
       % at the present time it's HDI, QAUZ or SEQAP.
       % SEQAP means that the output will not be transformed as it is
       % the case for HDI (Lisp form) and QAUZ (simplified output).



/*************************** CALL_GRAMMAR_SUBSET ****************************/
/* Depending on the Input_Type (question, statement or phrase), calls the     */
/* appropriate subset of the grammar.                                         */

call_grammar_subset(question,QAS,CI,Parser_Output2,ERR)
     --> question_Q(QAS,Parser_Output1,ERR), !,
         {transform(QAS,CI,Parser_Output1,Parser_Output2,ERR)}.

call_grammar_subset(statement,QAS,CI,Parser_Output2,ERR)
     --> statement_S(QAS,Parser_Output1,ERR), !,
         {transform(QAS,CI,Parser_Output1,Parser_Output2,ERR)}.

call_grammar_subset(phrase,QAS,CI,Parser_Output2,ERR)
     --> phrase_P(QAS,Parser_Output1,ERR), !,
         {transform(QAS,CI,Parser_Output1,Parser_Output2,ERR)}.



/************************* QUESTION and QUESTION_FORM ************************/
/* Defines acceptable questions.                                              */
/* VARIABLES: QT=Question Type [to identify the type of the question]         */
/*            Pos=Positivity ['no':main verb is negated, otherwise 'yes']     */
/*            MV=Main Verb                                                    */
/*            MVinf=Main Verb infinitive form                                 */
/*            MVmods=list of adverbs (verb modifiers)                         */
/*            MV_form=main verb form (inf, pre, ger, past, ppart)             */
/*            MV_number=for subject-verb number agreement                     */
/*            Roles=list of the Representations of the main verb's roles      */
/*            Nps=list of noun-phrases (noun-phrase is abbreviated as np)     */
/*            Pps=list of prepositional phrases (prep.phrase is pp)           */



question_Q(QAS,q(QT,Pos,Qauz_syntax_element),ERR)
     --> {QAS=qauz}, restricted_question_form(QT,Pos,Qauz_syntax_element,ERR).
         % Stricly for Qauz (very constrained) syntax questions. [q/3]

question_Q(QAS,q(when_is_it_true,Statement_Representation),ERR)
     --> {QAS=qauz}, interrogative(when), verb(is,v_pre,sing3),
         pronoun(it,sing), adj(true), pronoun(that,_),
         statement_S(QAS,Statement_Representation,ERR).
         % Stricly for Qauz "When is it true that S ?" question. [q/2]

question_Q(QAS,q(QT,Pos,MVinf,MVmods,Representation),ERR)
     --> question_form(QT2,MV,Pos,MVmods,[Nps1,Nps2],Pps,ERR),
         {outform(QAS,MVinf,MV,[Nps1,Nps2],Pps,Representation,ERR)},
         {question_code(Pos,MVinf,Nps2,QT2,QT)}.
         % General question form. [q/4]



question_form(is_there,is,Pos,_,[np_pp(sub,Nps),_],_,ERR)
     --> aux_group(be,Pos,Number), demonstr_adv(there),
         {get_number_compat(Number,Number2)},
         np_stat(Nps,Number2,ERR,optional).
         % Special form: for a "is there" question that doesn't contain a vp.
         % EX: "Is there an option to the DEFINE statement ?"

question_form(why_is,is,Pos,_,[np_pp(sub,Nps1),np_pp(d_o,Nps2)],_,ERR)
     --> ['why'], aux_group(be,Pos,Number),
         np_stat(Nps1,Number,ERR,optional), np_stat(Nps2,_,ERR,optional).
         % Special form: "why is np np" (doesn't contain a vp).
         % EX: "Why is 'X' an accessible record item ?"

question_form(QT,MV,Pos,MVmods,[Nps1,Nps2],Pps,ERR)
     --> header(QT,Pos1,MV_form,Nps1,ERR),
         vp(MV,MV_form,_,Pos2,MVmods,Nps2,Pps,ERR,mandatory),
         {and(Pos1,Pos2,Pos)}.
         % General form: header + vp.
         % MV_form: MV must be infinitive, gerund, ...

question_form(_,_,_,_,_,_,ERR) --> error_msg(illegal_question_form,ERR).



/********************************* HEADER ***********************************/
/* Defines acceptable question headers. A question header is the first part  */
/* of a question, located before the verb, which identifies (sometimes par- */
/* tially) the question type.                                                */

header(How_Type,Pos,MV_form,SUB,ERR)
     --> interrogative(how), !,
         header_how(How_Type,Pos,MV_form,SUB,ERR).

header_how(how_do_i,Pos,v_inf,np_pp(sub,pers_pro(Pr)),_)
     --> (['do'], !, pers_pronoun(Pr,Number), {Number\==sing, Number\==sing3}),
         aux_group(do,Pos,Number), pers_pronoun(Pr,Number).
         % EX: "How do I report X to Y ?"

header_how(how_to,Pos,v_inf,np_pp(sub,nil),_)
     --> positivity(Pos), ['to'].
         % EX: "How to suppress X ?"

header_how(how_does,Pos,v_inf,np_pp(sub,Nps),ERR)
     --> (['do'], !, np(Nps,Number,ERR,optional),
          {Number\==sing, Number\==sing3});
         aux_group(do,Pos,Number), np(Nps,Number,ERR,optional).
         % EX: "How does X define Y ?"

header_how(how_can_i,Pos,v_inf,np_pp(sub,pers_pro(Pr)),_)
     --> aux_group(modal,Pos,Number), pers_pronoun(Pr,Number).
         % EX: "How can I establish a column heading ?"

header_how(_,_,_,_,ERR) --> error_msg(illegal_how_question,ERR).
```

```
header(Why_Type,Pos,MV_form,SUB,ERR)
     --> interrogative(why), !,
         header_why(Why_Type,Pos,MV_form,SUB,ERR).

header_why(why_do_i,Pos,v_inf,np_pp(sub,pers_pro(Pr)),_)
     --> (['do'], !, pers_pronoun(Pr,Number), {Number\==sing, Number\==sing3});
         aux_group(do,Pos,Number), pers_pronoun(Pr,Number).
         % EX: "Why do I get X  ?"

header_why(why_am_i,Pos,v_ger,np_pp(sub,pers_pro(Pr)),_)
     --> aux_group(be,Pos,Number), pers_pronoun(Pr,Number).
         % EX: "Why am I getting X ?"

header_why(why_does,Pos,v_inf,np_pp(sub,Nps),ERR)
     --> (['do'], !, np(Nps,Number,ERR,optional),
          {Number\==sing, Number\==sing3});
         aux_group(do,Pos,Number), np(Nps,Number,ERR,optional).
         % EX: "Why does X give Y ?"

header_why(why_is,Pos,_,np_pp(sub,Nps),ERR)
     --> aux_group(be,Pos,Number), np(Nps,Number,ERR,optional).
         % EX: "Why is X  not ... ?"

header_why(_,_,_,_,ERR) --> error_msg(illegal_why_question,ERR).



header(What_Type,Pos,MV_form,SUB,ERR)
     --> interrogative(what), !,
         header_what(What_Type,Pos,MV_form,SUB,ERR).

header_what(what_happens_if,yes,v_inf,np_pp(sub,pers_pro(Pr)),_)
     --> verb(happens,v_pre,sing3), conjunction(if), pers_pronoun(Pr,_).
         % EX: "What happens if I use ... ?"

header_what(what_do_i_do,Pos,v_inf,np_pp(sub,Nps),ERR)
     --> (([do'], !, pers_pronoun(_,Number), {Number\==sing, Number\==sing3});
          (aux_group(do,Pos,Number1), pers_pronoun(_,Number1))),
         ['do'], (conjunction(if) ; conjunction(when)),
         np(Nps,_,ERR,mandatory).
         % EX: "What do I do if|when I get X ?"
         % This is a hypothetical question. (NOTE: if=/=when)

header_what(_,_,_,_,ERR) --> error_msg(illegal_what_question,ERR).



header(is_there,Pos,v_ppart,np_pp(sub,Nps),ERR)
     --> aux_group(be,Pos,Number), demonstr_adv(there),
         {get_number_compat(Number,Number2)},
         np(Nps,Number2,ERR,mandatory).
         % EX: "Is there a message given by the system ?"


header(can_i,Pos,v_inf,np_pp(sub,pers_pro(Pr)),_)
     --> aux_group(modal,Pos,Number), pers_pronoun(Pr,Number).
         % EX: "Can I use X in Y ?"


header(does,Pos,v_inf,np_pp(sub,Nps),ERR)
     --> (['do'], !, np(Nps,Number,ERR,optional),
          {Number\==sing, Number\==sing3});
         aux_group(do,Pos,Number), np(Nps,Number,ERR,optional).
         % EX: "Does X determine Y ?"
```

```
/************************ RESTRICTED QUESTION FORM ************************/
/* NOTE: this section covers only very constrained question forms for the  */
/*       QAUZ system. These had to be included in this fashion because they */
/*       are not used in a general way but only for their surface structure */
/*       where at most one element is variable.                            */
/*       The linguistic coverage of this section is extremely ad hoc.      */
/* NOTE: usage of square brackets is optional for these questions.         */

restricted_question_form(QT,Pos,Qauz_syntax_element,ERR)
     --> ((interrogative(what), !,
           (qauz_syntax_1(QT,Pos,Qauz_syntax_element,ERR)));
          (interrogative(where), !,
           qauz_syntax_2(QT,Pos,Qauz_syntax_element,ERR))).


qauz_syntax_1(SYN_1_2,yes,SE,ERR)
     --> verb(is,v_pre,sing3), ('['; []), specifier(the,sing),
         ((adj(complete), {SYN_1_2=syn2}) ; {SYN_1_2=syn1}),
         c_noun(syntax,_), preposition(of), np(SE,_,ERR,mandatory), (']'; []).
         % SYN1: "What is the syntax of file_declaration ?"
         % SYN2: "What is the complete syntax of file_declaration ?"
         % In SYN1 and SYN2, the 'variable' element is "file_declaration".

qauz_syntax_1(syn3,Pos,SE,ERR)
     --> verb(is,v_pre,sing3), np(SE,_,ERR,mandatory), positivity(Pos),
         specifier(a,sing), c_noun(component,sing), preposition(of).
         % SYN3: "What is file_declaration a component of ?"
         % The 'variable' element is "file_declaration".

qauz_syntax_1(syn5,yes,SE,ERR)
     --> verb(are,v_pre,plur), ('['; []), specifier(the,plur), adj(different),
         c_noun(kinds,plur), preposition(of), np(SE,_,ERR,mandatory), (']'; []).
         % SYN5: "What are the different kinds of X statement ?"
         % The 'variable' element is "X" (X is Report; Select; ...).

qauz_syntax_1(syn6,yes,_,_)
     --> ('['; []), c_noun(kinds,plur), preposition(of),
         c_noun(statements,plur), (']'; []),
         verb(are,v_pre,plur), demonstr_adv(there).
         % SYN6: "What kinds of statements are there ?"
         % This one doesn't contain a 'variable' element.


qauz_syntax_2(syn4,Pos,SE,ERR)
     --> verb(is,v_pre,sing3), np(SE,_,ERR,mandatory),
         positivity(Pos), verb(used,v_ppart,_).
         % SYN4: "Where is file_declaration used ?"
         % The 'variable' element is "file_declaration"



/****************************** QUESTION_CODE ******************************/
/* Table of codes for different types of questions.                        */
/* '?' means that this question type is recognized by the parser but not   */
/* currently handled by one of the answering systems (HDI/QAUZ).           */
/* Note: Qauz restricted question forms (syntax1-6) as well as Qauz when-is- */
/*       true-that-S are not included here;                                */
/*       they are so constrained that their type can be determined by their */
/*       specific surface form only.                                       */

question_code(_,_,_,how_do_i,hdi).
question_code(_,_,_,how_to,hdi).

question_code(_,MVinf,np_pp(d_o,np_args(_,spec(_),
```

```
                                ng(_,NounType,Noun,var(_),_))),Type,err)
    :- (Type=why_do_i ; Type=why_am_i), MVinf=get,
       NounType=c_noun, Noun=error_message.
       % 'Semantical' verification for an error-message question.

question_code(yes,_,_,why_do_i,why).
question_code(yes,_,_,why_am_i,why).
question_code(yes,_,_,why_does,why).
question_code(yes,_,_,why_is,why).

question_code(no,_,_,why_do_i,why_not).
question_code(no,_,_,why_am_i,why_not).
question_code(no,_,_,why_does,why_not).
question_code(no,_,_,why_is,why_not).

question_code(_,_,_,how_can_i,hci).

question_code(_,_,_,what_happens_if,hyp).

question_code(_,_,_,is_there,?).
question_code(_,_,_,how_does,?).
question_code(_,_,_,what_do_i_do,?).
question_code(_,_,_,can_i,?).
question_code(_,_,_,does,?).

question_code(_,_,_,_,error).



/********************************** OUTFORM ***********************************/
/* High-level procedure selecting the appropriate output representation for  */
/* a specific question answering  system.                                    */

outform(qauz,MVinf,MV,Nps,Pps,Representation,ERR)
      :- get_MVinf(MVinf,MV,Nps,Pps,Representation,ERR).

outform(hdi,MVinf,MV,Nps,Pps,Representation,ERR)
      :- get_MVinf(MVinf,MV,Nps,Pps,Representation,ERR).

outform(seqap,MVinf,MV,Nps,Pps,Representation,ERR)
      :- get_roles(MVinf,MV,Nps,Pps,Representation,ERR).



/********************************** GET_MVinf *********************************/
/* For QAUZ and HDI inputs that don't require roles in their representation. */

get_MVinf(MVinf,MV,Nps,Pps,no_roles(Nps,Pps),_)
      :- mini_lex(MV,VT,_,_,_,MVinf),  % Fetch MV's infinitive form
         (VT=v_ger ; VT=v_inf ; VT=v_past ; VT=v_ppart ; VT=v_pre).
         % This test must be done to make sure that MV is not a gerund
         % acting as a noun (a noun instead of a verb).

get_MVinf(_,_,_,_,_,ERR) :- error_msg(invalid_main_verb_inf,ERR,[],_).



/********************************** GET_ROLES *********************************/
/* Once the input has been syntactically understood, the parser produces its */
/* Representation in terms of roles (except if the input is 'special',       */
/* meaning that it does not have roles).                                     */

get_roles(MVinf,MV,Nps,Pps,Roles,ERR)
      :- mini_lex(MV,VT,_,_,_,MVinf),  % Fetch MV's infinitive form
         (VT=v_ger ; VT=v_inf ; VT=v_past ; VT=v_ppart ; VT=v_pre),
         % This test must be done to make sure that MV is not a gerund
         % acting as a noun (a noun instead of a verb).
         (role_post_processing(MVinf,Nps,Pps,Roles,ERR);
          (Roles=no_roles(Nps,Pps),
           error_msg(invalid_role_transformation,ERR,[],_))).



/********************************** STATEMENT *********************************/
/* Defines acceptable forms of statements.                                   */
/* NOTE: the expression ('['; []) or (']'; []) doesn't mean that each square */
/*       bracket is optional. It means that A PAIR of square brackets is     */
/*       optional: the lexical phase verifies that the number of '[' is      */
/*       equal to the number of ']'  If not, it's a fatal typo.              */

statement_S(_,s(Representation),ERR)
     --> ['there'], ((['is'], {Number=sing}) ; ['are'], {Number=plur}), !,
          there_is_statement(Representation,Number,ERR).

statement_S(_,s(Representation),ERR)
     --> ((variable(Object), {Var_Type=reg});
          (qauz_variable(Object), {Var_Type=qauz})),
         (is_adj_statement1(Representation,Object,Var_Type,ERR);
          is_a_statement1(Representation,Object,Var_Type,ERR)).

statement_S(_,s(Representation),ERR)
     --> ('['; []), np_simple(Nps,MV_number,ERR,optional), preposition(of),
         assignment_statement(Representation,Nps,MV_number,ERR).
         % The closing square bracket is in assignment_statement clause

statement_S(_,s(Representation),ERR)
     --> proper_quiz_noun(PQN,MV_number),
         is_a_statement2(Representation,PQN,MV_number,ERR).

statement_S(QAS,s(Representation),ERR)
     --> np_stat(Nps,MV_number,ERR,mandatory),
         (is_adj_statement2(Representation,Nps,MV_number,ERR);
          comparison_statement(Representation,Nps,MV_number,ERR);
          have_statement(Representation,Nps,MV_number,ERR);
          passive_statement(Representation,Nps,MV_number,QAS,ERR);
          regular_statement(Representation,Nps,MV_number,QAS,ERR)).

statement_S(_,_,ERR) --> error_msg(invalid_statement,ERR).



/******************** IS_ADJ, IS_A, ASSIGNMENT, COMPARISON, ******************/
/************** HAVE, THERE_IS, PASSIVE and REGULAR STATEMENTS ***************/

is_adj_statement1(is_adj(Pos,np_pp(sub,var(Object)),
                         np_pp(attribute,Adj)),Object,reg,ERR)
    --> vp_adj(is,_,Pos,Adj,ERR), {nonvar(Adj), Adj\==[]},
        {store_in_var_table(_,Adj,_,_,Object,_)}.
        % Object is a SEQAP variable
        % "is adj": <var> is <adj>

is_adj_statement1(is_adj(Pos,np_pp(sub,qauz_var(Object)),
                         np_pp(attribute,Adj)),Object,qauz,ERR)
    --> vp_adj(is,_,Pos,Adj,ERR), {nonvar(Adj), Adj\==[]}.
        % Object is a Qauz variable (not inserted in the symbol table)
        % "is adj": <qauz_var> is <adj>

is_adj_statement2(is_adj(Pos,np_pp(sub,Nps),
                         np_pp(attribute,Adj)),Nps,MV_number,ERR)
    --> vp_adj(is,MV_number,Pos,Adj,ERR), {nonvar(Adj), Adj\==[]}.
        % "is adj": <np> is <adj>
```

```
is_a_statement1(is_a(Pos,np_pp(identifier,var(Object)),
                np_pp(identified_object,np_args(_,spec(Spec),
                      ng(Adjs,NounType,Noun,var(Object),Pps)))),
                Object,reg,ERR)
    --> vp_assign(is,_,Pos,np_args(_,spec(Spec),
                      ng(Adjs,NounType,Noun,var(Object),Pps)),ERR),
        {store_in_var_table(Spec,Adjs,NounType,Noun,Object,Pps)}.
        % Object is a SEQAR variable
        % "is a": <var> is <np>

is_a_statement1(is_a(Pos,np_pp(identifier,qauz_var(Object)),
                np_pp(identified_object,np_args(_,spec(Spec),
                      ng(Adjs,NounType,Noun,_,Pps)))),
                Object,qauz,ERR)
    --> vp_assign(is,_,Pos,
                  np_args(_,spec(Spec),
                  ng(Adjs,NounType,Noun,_,Pps)),ERR).
        % Object is a QAUZ variable (not inserted in the symbol table)
        % "is a": <qauz_var> is <np>


is_a_statement2(is_a(Pos,np_pp(identifier,PQN),
                np_pp(identified_object,Nps)),PQN,MV_number,ERR)
    --> vp_assign(is,MV_number,Pos,Nps,ERR).
        % "is_a": <proper_quiz_noun> is <np>


assignment_statement(assign(Pos,np_pp(object,Nps1),
                            np_pp(object_modifier,Nps2),
                            np_pp(value,Nps3)),
                     Nps1,MV_number,ERR)
    --> np_stat(Nps2,_,ERR,optional), (']';[]),
        vp_assign(is,MV_number,Pos,Nps3,ERR).
        % "assignment": the <np_simple> of <var|np_conj> is <np_stat>


comparison_statement(comparison(Pos,np_pp(compar_object1,Nps1),Compar_op,
                            np_pp(compar_object2,Nps2)),Nps1,MV_number,ERR)
    --> aux_group(be,Pos,MV_number),
        ((comparative_adj(equal), ['to'], {Compar_op=equal});
         (comparative_adj(Compar_op), conjunction(than))),
        np_stat(Nps2,_,ERR,mandatory).
        % EX: "the width of the detail line is greater than the width of
        %      the print line"


have_statement(have(Pos,np_pp(sub,Nps1),MVmods,np_pp(d_o,Nps2)),
               Nps1,MV_number,ERR)
    --> vp_have(have,MV_number,Pos,MVmods,Nps2,ERR).
        % 'have' statement form.
        % The part before the vp specifies the 'haver'.
        % The part after the 'have' describes the object, with possibly
        % a value, of the 'having'.
        % EX: "I have a file with 100 items"


there_is_statement(there_is(Pos,MVmods,np_pp(object,Nps)),Number,ERR)
    --> adv(MVmod1), positivity(Pos),
        np_stat(Nps,Number,ERR,mandatory), adv(MVmod2),
        {concat([MVmod1,MVmod2],MVmods)}, !,
        {Nps=np_args(_,spec(Spec),_), ((Pos=no, !, Spec=unspec) ; true)}.
        % 'there is' statement form.
        % Final test: determiners cannot be used if negation (Pos=no).
        % EX: "there is no selection condition on records of .SFILE"


there_is_statement(_,_,ERR) --> error_msg(invalid_there_is_stat,ERR).


passive_statement(passive(Pos,MVinf,MVmods,Representation),
                  Nps1,MV_number,QAS,ERR)
    --> aux_group(be,Pos,MV_number),
        vp_passive(passive,MV,v_ppart,MVmods,Pps,ERR),
        {outform(QAS,MVinf,MV,[np_pp(sub,nil),np_pp(d_o,Nps1)],
                 Pps,Representation,ERR)}.
        % Strictly speaking it's a pseudo-passive form: see vp_passive.
        % NOTE: the real subject is unknown (it is probably QUIZ)
        % EX: "nothing is reported"
        %     "a link for .FILE is established"
        %     "the file was not found in the dictionary"
        %     '.TAB is established through an internal procedure"


regular_statement(regular(Pos,MVinf,MVmods,Representation),
                  Nps1,MV_number,QAS,ERR)
    --> vp(MV,_,MV_number,Pos,MVmods,Nps2,Pps,ERR,mandatory),
        {outform(QAS,MVinf,MV,[np_pp(sub,Nps1),Nps2],Pps,Representation,ERR)}.
        % Regular (and natural) form of a statement.



/********************************** PHRASE ***********************************/
/* Defines acceptable forms for phrases.                                     */

phrase_P(QAS,p(Representation),ERR)
    --> gerund_np_form(QAS,Representation,ERR);
        vp_form(QAS,Representation,ERR);
        regular_np_form(Representation,ERR).

phrase_P(_,_,ERR) --> error_msg(invalid_phrase,ERR).


gerund_np_form(QAS,gerund_np(Pos,MVinf,MVmods,Representation),ERR)
    --> vp(MV,v_ger,_,Pos,MVmods,Nps,Pps,ERR,optional),
        {outform(QAS,MVinf,MV,[np_pp(sub,nil),Nps],Pps,Representation,ERR)}
        % EX: "printing an item"
        % We consider this case as a vp.


vp_form(QAS,vp(Pos,MVinf,MVmods,Representation),ERR)
    --> positivity(Pos), ['to'], vp(MV,v_inf,_,_,MVmods,Nps,Pps,ERR,mandatory),
        {outform(QAS,MVinf,MV,[np_pp(sub,nil),Nps],Pps,Representation,ERR)}.
        % Explicit infinitive vp form. EX: "to print a subtotal"

vp_form(QAS,vp(Pos,MVinf,MVmods,Representation),ERR)
    --> vp(MV,_,_,Pos,MVmods,Nps,Pps,ERR,optional),
        {outform(QAS,MVinf,MV,[np_pp(sub,nil),Nps],Pps,Representation,ERR)}
        % General vp form. EX: "report a subtotal to a subfile"


regular_np_form(regular_np(Nps),ERR)
    --> np_stat(Nps,_,ERR,mandatory).
        % Simplest form of input: a np.
        % The prepositional phrase Pps modifies the noun.
        % Usage of square brackets is optional here.



/****************************** NOUN PHRASE ***********************************/
/* Defines a simple np and a conjoined np. It refers to three other impor-   */
/* tant groups of predicates; NP_FORM, NOUN_COMPLEX and NP_EQUIVALENT.       */
```

```
/* Note: a simple np is a np having only one head noun.                    */
/*                                                                         */
/* Format: np_args(_,spec(Spec),ng(Adjs,NounType,Noun,var(Var),Pps)).      */
/*                                                                         */
/* On pps attachment: in order to eliminate ambiguities the parser will    */
/* interpret the square-bracket (sb) notation as follows:                  */
/*  a pp modifies either the main verb or a np;                            */
/*  the np (modified by the pp) cannot be separated from the pp by the verb; */
/*  the default is: the pp modifies the verb;                              */
/*  if a pp modifies a noun then the whole np (with its pp) must be enclosed */
/*  in sbs, unless the context is 'obvious' (ex: in an assignment stat.).  */
/* EX: Hdi report [a file with leading zeroes] ?                           */
/*     Hdi report a numeric item to a subfile ?                            */
/*                                                                         */
/* On conjoined nps: allowed conjunctions for nps are AND and OR,          */
/*                   but one CANNOT mix them (using both AND and OR).      */
/* EX: "the file 'F1' and the file 'F2'" is OK                              */
/*     "the file 'F' and the file 'G' or the file 'H'" is AMBIGUOUS        */
/*     "the file 'F1' and [the file 'F2' with leading zereos]" is OK       */
/*     "[the file 'F1' and the file 'F2' with leading zereos]" is AMBIGUOUS. */
/*                                                                         */
/* NOTE: the expression ('['; []) or (']'; []) doesn't mean that each square */
/*       bracket is optional. It means that A PAIR of square brackets is   */
/*       optional: the lexical phase verifies that the number of '[' is    */
/*       equal to the number of ']'. If not, it's a fatal typo.            */

np(Nps,Number,ERR,M_O)
    --> np_simple(Nps1,Number1,ERR,optional),          % np_simple:simple form
        ((np_conj([Nps1],_,Nps,Number2,ERR,optional),  % np_conj:conjoined form
          {Number=Number2});
         ({Nps=Nps1, Number=Number1})).

np(_,_,ERR,mandatory) --> error_msg(invalid_noun_phrase,ERR).



np_simple(np_args(_,spec(Spec),ng(Adjs,NounType,Noun,var(Var),Pps)),
          Number,ERR,M_O)
    --> ['['],
        np_form(Spec,ng(Adjs,NounType,Noun,Var,Pps),Number,ERR),
        (pps(Pps,ERR);[]),
        [']'].

np_simple(np_args(_,spec(Spec),ng(Adjs,NounType,Noun,var(Var),Pps)),
          Number,ERR,M_O)
    --> np_form(Spec,ng(Adjs,NounType,Noun,Var,Pps),Number,ERR).

np_simple(_,_,ERR,mandatory) --> error_msg(invalid_simple_noun_phrase,ERR).



np_conj(NPSin,Conj,NPSout,Number,ERR,M_O)
    --> ((preposition(Conj), !, fail) ; conjunction(Conj)),
         np_simple(Nps,_,ERR,optional), {append(NPSin,[Nps],NPSnext)},
        np_conj(NPSnext,Conj,NPSout,Number,ERR,mandatory).
        % The first test in this clause, preposition(Conj), is meant to
        % make sure that the next word is not a preposition or a conjunction.
        % (many words can play more than one syntactic role when analysed in
        %  a surface-level fashion. For example, 'after' can be a preposition
        %  or a conjunction.)
        % If it is the case then priority is given to the preposition.

np_conj(NPSin,Conj,NPSout,Number,ERR,M_O)
    --> (nonvar(Conj)), {append([conj,Conj],[NPSin],NPSin2), NPSout=..NPSin2},
        ((Conj=and, !, Number=plur) ; (Conj=or, !, Number=sing) ; true).
        % About Number: this is under the hypothesis that each conjoined
        %               element is itself singular. [and=>plur, or=>sing]

np_conj(_,_,_,_,ERR,mandatory) --> error_msg(invalid_conj_np,ERR).



np_stat(np_args(_,spec(Spec),ng(Adjs,NounType,Noun,var(Var),Pps)),
        Number,ERR,M_O)
    --> (['['] ; []),
        np_form(Spec,ng(Adjs,NounType,Noun,Var,Pps),Number,ERR),
        (pps(Pps,ERR);[]),
        ([']']; []).
        % Special np form used for statements and phrases.
        % (The prepositional phrase Pps modifies the noun group)
        % It is identical to np_simple above except that here square brackets
        % are optional. In statements and phrases, it is often the case that
        % the verb is not important (doesn't have roles); so it is much more
        % convenient not to force the user to put square brackets in obvious
        % situations.

np_stat(_,_,ERR,mandatory) --> error_msg(invalid_stat_noun_phrase,ERR).



/********************************* NP_FORM ***********************************/
/* (Noun Phrase Form): high-level definition of acceptable np forms.       */

np_form(Spec,ng(Adjs,NounType,Noun,Var,_),Number,ERR)
    --> noun_complex(Spec,Adjs,NounType,Noun,Var,_,Number,ERR),
        ((quiz_stuff(Err_or_Code), {store_quiz_stuff(Err_or_Code)}) ; []).
        % Quiz stuff is asserted to simplify parameter passing.

np_form(Spec,ng(Adjs,NounType,Noun,Var,Pps),Number,_)
    --> np_equivalent(Spec,Adjs,NounType,Noun,Var,Pps,Number).
        % A np_equivalent is an element that can replace a normal np:
        % a pronoun, a proper_quiz_noun, a number, ...
        % This element plays approximately the same role.



/******************************** NOUN_COMPLEX ******************************/
/* Two basic forms of a noun group.                                        */

noun_complex(Spec,Adjs,NounType,Noun,Var,_,Number,ERR)
    --> specifier(Spec,Number), adj_group(Adjs,ERR),
        noun(NounType,Noun,Number),
        ((variable(Var), {store_in_var_table(Spec,Adjs,NounType,Noun,Var,_)}),
         []).
        % The head noun is either a common noun or a quiz_proper_noun

noun_complex(Spec,Adjs,m_noun,MN,Var,_,Number,ERR)
    --> ((specifier(Spec,Number), !,  % Only certain specifiers are acceptable
          ({Spec=the; Spec=all; Spec=some; Spec=unspec; Spec=her;
            Spec=his; Spec=its; Spec=my; Spec=our; Spec=their; Spec=your}));
         []),
        adj_group(Adjs,ERR), m_noun(MN,Number),
        ((variable(Var), {store_in_var_table(Spec,Adjs,m_noun,Noun,Var,_)}),
         []).
        % The head noun is a mass noun (m_noun)



adj_group(Adjs,ERR)
    --> adjp(Adjs1,ERR),
        ((proper_quiz_noun(Adjs2,Number),
          (((Adjs1\==[], append(Adjs1,[Adjs2],Adjs)));
```

```
          ({Adjs=[Adjs2]}))));
        {Adjs=Adjs1}).


noun(c_noun,CN,Number)     --> c_noun(CN,Number).
noun(proper_quiz_noun,QN,Number) --> proper_quiz_noun(QN,Number)


/****************************** NP_EQUIVALENT ******************************/
/* Defines forms which stricly speaking are not (pure) noun phrases but     */
/* equivalents. (an equivalent can replace a normal np)                     */

np_equivalent(_,_,qauz_var,_,Var,_,_)
     --> qauz_variable(Var).
         % A qauz variable is a np_equivalent by itself.
         % We don't want to look it up in the symbol table or the lexicon.

np_equivalent(Spec,Adjs,NounType,Noun,Var,Pps,_)
     --> variable(Var),
         {var_table(spec(Spec),Adjs,NounType,Noun,var(Var),Pps)}.
         % Refering to a variable introduced earlier.
         % See last clause of 'noun_complex' above.

np_equivalent(_,_,pers_pro,P_PR,_,_,Number) --> pers_pronoun(P_PR,Number).

np_equivalent(_,_,pronoun,PR,_,_,Number) --> pronoun(PR,Number).

np_equivalent(_,_,proper_quiz_noun,PQN,_,_,Number)
     --> proper_quiz_noun(PQN,Number).
         % EX: "Date format is YYMMDD"

np_equivalent(Spec,_,gerund,CN,_,_,Number)
     --> specifier(Spec,Number), verb(CN,v_ger,_),
         {mini_lex(Adj,c_noun,Number,_,_,_)}.
         % The gerund as a noun. EX: "... at the beginning of a ...

np_equivalent(_,_,quiz_code_errmessage,_,_,_,_)
     --> quiz_stuff(Err_or_Code), {store_quiz_stuff(Err_or_Code)}.

np_equivalent(_,_,number,Num,_,_,_)
     --> [Num], {number1(Num)}.
         % Num is an integer or real number.



/***************************** ADJECTIVE-PHRASES ****************************/
/* Simple and 'ANDed' adjectives.                                           */

adjp(ADJS,ERR) --> adjp1([],ADJS,ERR).

adjp1(ADJSin,ADJSout,ERR)
     --> adj(Adj), {append(ADJSin,[Adj],ADJSnext)},
         adjp1(ADJSnext,ADJSout,ERR).

adjp1(ADJSin,ADJSout,ERR)
     --> adj(Adj),
         conjunction(Conj),
         ({Conj=and} ; error_msg(invalid_conj_adjs(Conj),ERR)),
         {append(ADJSin,[Adj],ADJSnext)}, adjp1(ADJSnext,ADJSout,ERR).


adjp1(ADJS,ADJS,_)
     --> [].
         % "ANDed" adjectives will be recognized, as shown
         %  in the second clause of 'adjp1' above.
         %  EX: " ... a permanent and important file ... "


/************************** PREPOSITIONAL PHRASE ****************************/
/* Simple and conjoined pps.                                                */

pps(Pps,ERR)
     --> pps_simple(pp(Prep,Nps),ERR),                  % simple pp
         (pps_conj(Prep,_,[Nps],Pps,ERR);               % conjoined pps
          pps_consec([pp(Prep,Nps)],Pps,ERR);           % consecutive pps
          {Pps=pp(Prep,Nps)}).


pps_simple(pp(Prep,Nps),ERR)
     --> preposition(Prep), np(Nps,_,ERR,mandatory).
         % If Nps contains a set of conjoined nps it means that
         % the preposition is distributed over the set of nps.
         % EX: "to X and Y and Z" = "to X and to Y and to Z"


pps_consec(Pps_in,Pps_in,ERR) --> [].
         % Pps_Consec is the case where we have consecutive prepositions;
         % EX: "Hdi report 'X' to 'Y' at 'Z' ?"

pps_consec(Pps_in,Pps_out,ERR)
     --> preposition(Prep), np(Nps,_,ERR,mandatory),
         {append([pp(Prep,Nps)],Pps_in,Pps_in2)},
         pps_consec(Pps_in2,Pps_out,ERR).


pps_conj(Prep,Conj,PPSin,PPSout,ERR)
     --> conjunction(Conj), preposition(Prep), np_simple(Nps,_,ERR,mandatory),
         {append(PPSin,[Nps],PPSnext)},
         pps_conj(Prep,Conj,PPSnext,PPSout,ERR).

pps_conj(Prep,Conj,PPSin,PPSout,ERR)
     --> {nonvar(Conj)},
         {PPSout=[pp(Prep,conj(Conj,PPSin))]}.
         % pps_conj is the explicit case where we have "to X and to Y"
         % 'Explicit' means non-elliptical ("to" is repeated).


/****************** VP_ADJ, VP_ASSIGN, VP_HAVE and VP_PASSIVE ***************/
/* Special verb phrase forms for statements and phrases.                    */

vp_adj(is,MV_number,Pos,ADJ,ERR)
     --> {get_number_compat(MV_number2,MV_number)},
         aux_group(be,Pos,MV_number2), adjp(ADJ,ERR), {nonvar(ADJ), ADJ\==[]}
         % "is adj" statement.
         % EX: "records of .FILE are optional"


vp_assign(is,MV_number,Pos,Nps,ERR)
     --> aux_group(be,Pos,MV_number), np_stat(Nps,MV_number,ERR,optional).
         % "is a" statement. The 'object' is set to a value.
         % EX: "the type of 'elem' is DATE"
```

```
vp_have(have,MV_number,Pos,MVmods,Nps,ERR)
    --> ((aux_group(do,Pos,MV_number), verb(have,v_inf,_));
         aux_group(have,Pos,MV_number)),
        adv(MVmod1), np(Nps,_,ERR,mandatory), adv(MVmod2),
        {concat([MVmod1,MVmod2],MVmods)}.
        % 'have' statement: do (not) have.
        % EX: ".DFILE does not have an alias"


vp_passive(passive,MV,v_ppart,MVmods,Pps,ERR)
    --> adv(MVmod1), verb(MV,v_ppart,_), adv(MVmod2),
        {concat([MVmod1,MVmod2],MVmods)}, (pps(Pps,ERR) ; []).
        % In our form of passive statement, we don't really have a vp.
        % What we have is a verb followed by nothing or a pp.
        % Strictly speaking it's a pseudo-passive form.
        % EX: "the value of .OPT is set to no"
        % The prepositional phrase Pps modifies the verb.



/******************************** VERB PHRASE ********************************/
/* 'General' verb phrase forms.                                               */

vp(MV,MV_form,MV_number,Pos,MVmods,Nps,Pps,ERR,M_O)
    --> {nonvar(MV_form)}, positivity(Pos), adv(MVmod1),
        (({MV_form=v_inf}, verb(MV,v_inf,_));
         ({MV_form=v_ger}, verb(MV,v_ger,_))),
        adv(MVmod2),
        (vp_compl_def(MVmod3,Nps,Pps,ERR) ; vp_compl_reg(MVmod3,Nps,Pps,ERR)),
        {concat([MVmod1,MVmod2,MVmod3],MVmods)}.
        % Infinitive and Gerund: no auxiliaries.

vp(MV,MV_form,MV_number,Pos,MVmods,Nps,Pps,ERR,M_O)
    --> {MV_form\==v_inf, MV_form\==v_ger},
        aux_group(Aux,Pos,MV_number), adv(MVmod1),
        (({MV_form\==v_pre}, verb(MV,MV_form,_)); verb(MV,MV_form,MV_number)),
        adv(MVmod2),
        (vp_compl_def(MVmod3,Nps,Pps,ERR) ; vp_compl_reg(MVmod3,Nps,Pps,ERR)),
        {concat([MVmod1,MVmod2,MVmod3],MVmods)}.

vp(_,_,_,_,_,_,_,ERR,mandatory) --> error_msg(invalid_vp,ERR).



vp_compl_def([],np_pp(d_o,nil),Pps,ERR)
    --> (pps(Pps,ERR) ; []).
        % Secondary case: verb (+ pp).
        % EX: "Hdi subtotal using 'D' ?"
        %     "Hdi print in Column1 ?"
        %     "Hdi subtotal into 'X' ?"
        %     "Hdi multiply ?"
        %     "Why does the code not work ?"
        % The prepositional phrase Pps modifies the verb.

vp_compl_reg(MVmod3,np_pp(d_o,Nps),Pps,ERR)
    --> np(Nps,_,ERR,mandatory), adv(MVmod3), (pps(Pps,ERR) ; []).
        % General case: verb + np (+ pp).
        % EX: " ... report a numeric item"
        % EX: " ... report a subtotal to a file"
        % The prepositional phrase Pps modifies the verb.



/****************************** AUX_GROUP ********************************/
/* Defines the auxiliary group.                                           */

/* The following is obviously a simplification: auxiliaries and modals are  */
/* only recognized (not parsed).                                            */

aux_group(be,Pos,Number)
    --> [Aux],
        (({var(Number)}, {mini_lex(Aux,_,Number,_,_,be)});
         (({Number2=Number} ; {get_number_compat(Number2,Number)}),
          {mini_lex(Aux,_,Number2,_,_,be)})),
        positivity(Pos).
        % is, is not, are, are not, am, am not.

aux_group(do,Pos,Number)
    --> [Aux],
        (({var(Number)}, {mini_lex(Aux,_,Number,_,_,do)});
         (({Number2=Number} ; {get_number_compat(Number2,Number)}),
          {mini_lex(Aux,_,Number2,_,_,do)},
          {((Aux=do, !, (Number=sing1;Number=sing2;Number=plur)) ; true)}))),
        positivity(Pos).
        % does, does not, do, do not.

aux_group(have,Pos,Number)
    --> [Aux],
        (({var(Number)}, {mini_lex(Aux,_,Number,_,_,have)});
         (({Number2=Number} ; {get_number_compat(Number2,Number)}),
          {mini_lex(Aux,_,Number2,_,_,have)},
          {((Aux=have, !, (Number=sing1;Number=sing2;Number=plur)) ; true)}))),
        positivity(Pos).
        % have, have not, has, has not, had, had not.

aux_group(modal,Pos,_)
    --> [Modal], {mini_lex(Modal,v_aux,_,_,_,_)},
        ({(Modal=cannot, Pos=no)} ; positivity(Pos)),
        [Aux], {(X=be ; X=do ; X=have), mini_lex(Aux,v_inf,_,_,_,_,X)}.
        % can do, can be, can have, should do, ...

aux_group(modal,Pos,_)
    --> [Modal], {mini_lex(Modal,v_aux,_,_,_,_)},
        ({(Modal=cannot, Pos=no)} ;  positivity(Pos)).
        % can, should, must, will, ...

aux_group(X,Pos,_)
    --> {var(X)}, positivity(Pos).
        % When a sentence does not contain an auxiliary.
        % var(X) makes sure that, for example, aux_group(modal,Pos,_)
        % will not be successfull after failing appropriate clauses.



/************************** High level lexical rules **************************/


/* ADJECTIVE */
adj(ADJ) --> [ADJ],
             {(mini_lex(ADJ,reg_adj,_,_,_,_) ; mini_lex(ADJ,rel_adj,_,_,_,_))}.


/* ADVERB */
adv(ADV) --> [ADV], {mini_lex(ADV,reg_adv,_,_,_,_)}.
adv([]) --> [].


/* AUXILIARY */
aux --> [AUX], {mini_lex(AUX,v_aux,_,_,_,_)}.

/* COMMON NOUN */
```

```
c_noun(CN,Number) --> [CN], {mini_lex(CN,c_noun,Number,_,_,_)}.


/* COMPARATIVE ADJECTIVE */
comparative_adj(Cadj) --> [Cadj], {mini_lex(Cadj,comp_adj,_,_,_,_)}.


/* CONJUNCTION */
conjunction(Conj) --> [Conj], {mini_lex(Conj,conj,_,_,_,_)}.


/* DEMONSTRATIVE ADVERB */
demonstr_adv(Dadv) --> [Dadv], {mini_lex(Dadv,demonstr_adv,_,_,_,_)}.


/* INTERROGATIVE: starts a question. */
interrogative(Interr) --> [Interr],
                          {(mini_lex(Interr,inter_pro,_,_,_,_);
                            mini_lex(Interr,inter_adv,_,_,_,_))}.


/* MASS NOUN */
m_noun(MN,Number) --> [MN], {mini_lex(MN,m_noun,Number,_,_,_)}.


/* PARTICLE */
particle(Part) --> [Part], {mini_lex(Part,particle,_,_,_,_)}.


/* PERSONAL PRONOUN */
pers_pronoun(Ppro,Number) --> [Ppro], {mini_lex(Ppro,pers_pro,Number,_,_,_)}.


/* REGULAR PRONOUN */
pronoun(Pro,Number) --> [Pro], {mini_lex(Pro,reg_pro,Number,_,_,_)}.


/* POSSESSIVE ADJECTIVE */
possess_adj(Padj) --> [Padj], {mini_lex(Padj,poss_adj,_,_,_,_)}.


/* PREPOSITION: notice that 'using' IS NOT a preposition. It is used here
                as an ad hoc preposition simply because its behaves like
                one in certain types of sentences that we cover. */
preposition(Prep)  --> [Prep], {mini_lex(Prep,prep,_,_,_,_)}.
preposition(using) --> ['using'].


/* PRETERITE */
preterite(Pre) --> [Pre], {mini_lex(Pre,preterite,_,_,_,_)}.


/* PROPER_QUIZ_NOUN */
proper_quiz_noun(PQN,Number) --> [PQN1],
                                 {PQN1=..PQN2, nth1(1,PQN2,pn_qn),
                                  nth1(2,PQN2,PQN),
                                  (mini_lex(PQN,quiz_noun,Number,_,_,_);
                                   mini_lex(PQN,p_noun,Number,_,_,_))}.
% Note: a proper-quiz-noun is a proper-noun or a Quiz-noun.
%       A Quiz-noun is a special kind of proper-noun.


/* QUIZ STUFF */
quiz_stuff(QS) --> [X], {X=..X2, nth1(1,X2,quiz_stuff), nth1(2,X2,QS)}.
```

```
/* SPECIFIER: the following should not be taken as 'the' way to go about
              quantifiers and determiners (called specifiers).
              Several cases are indeed quite ad hoc. */
specifier(Spec,Number) --> [Spec],
                           {(mini_lex(Spec,article,Number,_,_,_);
                             mini_lex(Spec,num_adj,Number,_,_,_);
                             mini_lex(Spec,quant_adj,Number,_,_,_))}

specifier(PA,_) --> possess_adj(PA).

specifier(one,sing)  --> [Spec], {integer(Spec), Spec=1}.
specifier(Spec,plur) --> [Spec], {integer(Spec), Spec > 1}.

specifier(int_or_more(Spec),plur)  --> [Spec], {integer(Spec)},
                                        ['or'], ['more'].
specifier(zero_or_more,plur) --> ['0m'].
specifier(zero_or_more,plur) --> (['0'] , ['zero']), ['or'], ['more'].
specifier(one_or_more,plur)  --> ['1m'].
specifier(one_or_more,plur)  --> (['1'] , ['one']), ['or'], ['more'].

specifier(range(Int1,Int2),plur) --> [Spec],
                                     {name(Spec,Chars),
                                      get_integer(Chars,[],Chars2,Int1),
                                      get_integer(Chars2,[],_,Int2)}.

specifier(unspec,_) --> ['unspec'].
specifier(unspec,_) --> [].


/* VARIABLE */
variable(VAR)      --> [X], {X=..X2, nth1(1,X2,var), nth1(2,X2,VAR)}.

qauz_variable(VAR) --> [X], {X=..X2, nth1(1,X2,qauz_var), nth1(2,X2,VAR)}.


/* VERB */
verb(MV,MV_form,MV_number)
    --> {nonvar(MV_number), MV_number=sing}, !,
        verb2(MV,MV_form,sing3), (particle(_) ; preterite(_) ; []).
        % Third singular person case.
        % A verb can be followed by a particle (up, down, ...) or
        % a preterite ('rid' in "get rid of ...").

verb(MV,MV_form,MV_number)
    --> {MV_number\==sing}, !,
        verb2(MV,MV_form,MV_number), (particle(_) ; preterite(_) ; [])
        % For MV_number \== sing.


verb2(MV,MV_form,MV_number)
    --> {nonvar(MV_number), MV_number=sing3},
        [MV], {mini_lex(MV,MV_form,MV_number2,_,_,Inf), Inf\==nil},
        {nonvar(MV_number2), MV_number2=sing3}.

verb2(MV,MV_form,MV_number)
    --> {MV_number\==sing3},
        [MV], {mini_lex(MV,MV_form,MV_number,_,_,Inf), Inf\==nil}.

/* <><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>*/


/****************************** Home Made Utilities **************************/


/* CLEAN_UP_QUIZ_STUFF and CLEAN_UP_VAR_TABLE: removes assertions from the   */
```

```
/* global environment. These come in two kinds:                          */
/* 1- Quiz 'stuff' found in the input [error message, Quiz code, ...]    */
/* 2- Symbol table (variable table) for user's variables.                */
clean_up_quiz_stuff :- retract(quiz_stuff_elem(_)), fail.
clean_up_quiz_stuff.

clean_up_var_table(true) :- retract(var_table(_,_,_,_,_,_)), fail.
clean_up_var_table(_).


/* STORE_IN_VAR_TABLE: maintains a symbol table of the user's variables.  */
store_in_var_table(S,A,NT,N,V,P) :- (var_table(_,_,_,_,var(V),_), !);
                                    assert(var_table(spec(S),A,NT,N,var(V),P)).


/* GET_QUIZ_STUFF: once the parsing has been completed, we check whether some
                   Quiz stuff was found to return it as an individual para-
                   meter. This Quiz stuff is kept globally to simplify para-
                   meter passing in complicated situations.               */
get_quiz_stuff(QS) :- quiz_stuff_elem(QS) ; QS=[].


/* STORE_QUIZ_STUFF: keep Quiz stuff in the global environment.          */
store_quiz_stuff(QS) :- (quiz_stuff_elem(QS), !) ; assert(quiz_stuff_elem(QS)).


/* ERROR_MSG: takes care of the error message, when an error is found.   */
error_msg(MSG,ERR) --> ([var(ERR)], get_rest_input(X), {ERR=error(MSG,X)});
                       {true}.


/* GET_REST_INPUT: eats up the rest of the input (when an error occurs)  */
get_rest_input([H|R]) --> [H], get_rest_input(R).
get_rest_input([]) --> [].


/* AND: logical and */
and(yes,yes,yes):- !.
and(_,_,no).


/* POSITIVITY: a negation flag */
positivity(no) --> ['not'].
positivity(no) --> ['no'].
positivity(yes)  --> [].


/* GET_NUMBER_COMPAT: verifies the number compatibility between an entry X  */
/*                    and the 'standard' form contained in the lexicon.     */
/*                    It is used when the subject preceeds the verb for     */
/*                    number agreement. EX: the file (sing) is (sing3) ...  */
get_number_compat(sing3,sing).  get_number_compat(sing,sing3).
get_number_compat(sing2,sing).  get_number_compat(sing,sing2).
get_number_compat(sing1,sing).  get_number_compat(sing,sing1).
get_number_compat(sing,sing).
get_number_compat(plur,plur).


/* NUMBER1: verifies whether X is an integer or a real number */
number1(X) :- nonvar(X), name(X,X_Chars), number2(X_Chars).
number2([H|R]) :- is_digit(H), number2(R).
number2([H|R]) :- is_period(H), number3(R).
number2([]).
number3([H|R]) :- is_digit(H), number3(R).
number3([]).


/* GET_INTEGER: for a special kind of specifier; a range. EX: 5-9, 1-7, ... */
get_integer(X,In,Out,Int) :- (X=[] ; (X=[45|Out])),
                             reverse(In,In2), name(Int,In2), !, integer(Int).
get_integer([H|R],In,Out,Int) :- append([H],In,In2),
                                 get_integer(R,In2,Out,Int).


/* CONCAT: concatenates a list of empty lists and atoms (adverbs are either */
/*         an atom or an empty list) to produce a single list.               */
concat([],[]).
concat([H|R],List) :- H\==[], append([H],List2,List), concat(R,List2).
concat([H|R],List) :- H=[], concat(R,List).

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

```
/* ------------------------------------------------------------------------
   Author : Sylvain Delisle
   Project: SEQAP parser
   File   : transform.pl
   Version: November 11, 1986

   This file contains two transformation procedures;
   one for the QAUZ answering system (which simplifies the SEQAP output),
   one for the HDI answering system (which translates the SEQAP output into
   a Lisp expression). It should be mentioned that these little 'translators'
   do not cover 100% of SEQAP's capabilities (see descriptions below)
   For QAUZ: conjoined noun phrases and conjoined prepositional phrases
             are not translated. Also, the "is_adj" statement is translated
             only for single adjectives (not lists of adjectives)
   For HDI: "have" statement, "passive" statement, QAUZ syntax questions and a
            few other QAUZ specific questions, and phrases are not translated

------------------------------------------------------------------------*/

transform(_,_,In,In,ERR) :- nonvar(ERR). % Error during the parsing; stop

transform(seqap,_,In,In,_). % Original SEQAP Output.


/***************** DESCRIPTION OF THE OUTPUT FORMATS FOR QAUZ *****************

Notation: MVinf=Main Verb infinitive form
          Pos=Positivity (false if MV negated, otherwise true)
          MVmods=Main Verb modifiers (Adverbs)
          CI=Canonical Input
          QT=Question Type
          Sub=Subject (of the MV)
          D_O=Direct Object (of the MV)
          Pps=Prepositional Phrases (modifying the MV)
          Adj=Adjective


'standard' question: qauz(CI,QT,MVinf,Pos,MVmods,Sub,D_O,Pps).

'syntax 1-5' question: qauz(CI,QT,nil,Pos,nil,nil,D_O,nil).

'syntax 6' question: qauz(CI,syn6,nil,Pos,nil,nil,nil,nil).

'when_is_it_true' question:
qauz(CI,when_is_it_true(Stat_Type),MVinf,Pos,MVmods,Sub,D_O,Pps),
where Stat_Type is the type of S in: "When is it true that S ?".

'is_adj' statement: qauz(CI,is_adj,nil,Pos,nil,Sub,d_o(Adj),nil).

'is_a' statement: qauz(CI,is_a,nil,Pos,nil,Sub,D_O,nil).

'assign' statement: qauz(CI,assign,nil,Pos,nil,Sub,D_O,nil)

'comparison' statement: qauz(CI,comparison,nil,Pos,nil,Sub,D_O,Compar_op),
where Compar_op is placed in the Pps parameter for convenience.

'have' statement: qauz(CI,have,nil,Pos,MVmods,Sub,D_O,nil).

'there_is' statement: qauz(CI,there_is,nil,Pos,MVmods,nil,D_O,nil).

'passive' statement: qauz(CI,passive,MVinf,Pos,MVmods,nil,D_O,Pps).

'regular' statement: qauz(CI,regular,MVinf,Pos,MVmods,Sub,D_O,Pps).

'regular_np' phrase: qauz(CI,regular_np,nil,nil,nil,Sub,nil,nil).

'gerund_np' phrase: qauz(CI,gerund_np,MVinf,Pos,MVmods,nil,D_O,Pps).

'vp' phrase: qauz(CI,vp,MVinf,Pos,MVmods,nil,D_O,Pps).


LOW-LEVEL REPRESENTATION OF Sub AND D_O:

Sub=sub(HN,HN_Type,HN_Var,HN_Adjs,HN_Pps).
% The passive object is placed in the subject parameter for
% convenience, it is not the 'real' subject

D_O=d_o(HN,HN_Type,HN_Var,HN_Adjs,HN_Pps).

Notation: HN=Head Noun
          HN_Type=Head Noun Type
          HN_Var=Head Noun Variable
          HN_Adjs=Head Noun Adjectives
          HN_Pps=Head Noun Prepositions (Pps modifying the Head Noun).

*****************************************************************************/


/****************************** QAUZ  TRANSFORM ******************************/

transform(qauz,CI,In,Out,_)  % q/4
    :- In=q(QT,Pos,MVinf,MVmods,no_roles(Nps,Pps)),
       simplify_Np(Nps,Sub,D_O), simplify_Pp(Pps,[],Pps2),
       ((MVmods=[], MVmods2=nil) ; MVmods2=MVmods),
       Out=qauz(CI,QT,MVinf,Pos,MVmods2,Sub,D_O,Pps2).

transform(qauz,CI,In,Out,_)` % q/3:syn1-5
    :- In=q(QT,Pos,Qauz_syntax_element),
       simplify_Np(np_pp(d_o,Qauz_syntax_element),nil,D_O),
       Out=qauz(CI,QT,nil,Pos,nil,nil,D_O,nil).

tranform(qauz,CI,In,Out,_)  % q/3:syn6
    :- In=q(syn6,Pos,_),
       Out=qauz(CI,syn6,nil,Pos,nil,nil,nil,nil).

transform(qauz,CI,In,Out,_)  % q/2
     - In=q(when_is_it_true,Statement_Representation),
       transform(qauz,_,Statement_Representation,Out2,_),
       Out2=qauz(_,Stat_Type,MVinf,Pos,MVmods,Sub,D_O,Pps),
       ((MVmods=[], MVmods2=nil) ; MVmods2=MVmods),
       Out=qauz(CI,when_is_it_true(Stat_Type),MVinf,Pos,MVmods2,Sub,D_O,Pps)

transform(qauz,CI,In,Out,_)  % is_adj statement
    :- In=s(is_adj(Pos,Np1,np_pp(attribute,Adj))),
       simplify_Np(Np1,Sub,nil),
       Out=qauz(CI,is_adj,nil,Pos,nil,Sub,d_o(Adj),nil).

transform(qauz,CI,In,Out,_)  % is_a statement
    :- In=s(is_a(Pos,Np1,Np2)),
       simplify_Np([Np1,Np2],Sub,D_O),
       Out=qauz(CI,is_a,nil,Pos,nil,Sub,D_O,nil).

transform(qauz,CI,In,Out,_)  % assign statement
    :- In=s(assign(Pos,Np1,Np2,Np3)),
       Np1=np_pp(object,Np11), Np2=np_pp(object_modifier,Np22),
       Np3=np_pp(value,Np33),
       Np11=np_args(_,_,ng(Adjs,NT,N,Var,_)),
       Np111=np_args(_,_,ng(Adjs,NT,N,Var,[pp(of,Np22)])),
       simplify_Np([np_pp(sub,Np111),np_pp(d_o,Np33)],Sub,D_O),
```

```
       Out=qauz(CI,assign,nil,Pos,nil,Sub,D_O,nil).

transform(qauz,CI,In,Out,_)  % comparison statement
    :- In=s(comparison(Pos,Np1,Compar_op,Np2)),
       simplify_Np([Np1,Np2],Sub,D_O),
       Out=qauz(CI,comparison,nil,Pos,nil,Sub,D_O,Compar_op).
       % The Compar_op is placed in the Pps parameter for convenience.

transform(qauz,CI,In,Out,_)  % have statement
    :- In=s(have(Pos,Np1,MVmods,Np2)),
       simplify_Np([Np1,Np2],Sub,D_O),
       ((MVmods=[], MVmods2=nil) ; MVmods2=MVmods),
       Out=qauz(CI,have,nil,Pos,MVmods2,Sub,D_O,nil).

transform(qauz,CI,In,Out,_)  % there_is statement
    :- In=s(there_is(Pos,MVmods,Np1)),
       simplify_Np(Np1,nil,D_O),
       ((MVmods=[], MVmods2=nil) ; MVmods2=MVmods),
       Out=qauz(CI,there_is,nil,Pos,MVmods2,nil,D_O,nil).

transform(qauz,CI,In,Out,_)  % passive statement
    :- In=s(passive(Pos,MVinf,MVmods,no_roles(Nps,Pps))),
       simplify_Np(Nps,nil,D_O), simplify_Pp(Pps,[],Pps2),
       ((MVmods=[], MVmods2=nil) ; MVmods2=MVmods),
       Out=qauz(CI,passive,MVinf,Pos,MVmods2,nil,D_O,Pps2).
       % Note: passive form is transformed in active;
       %       what seems to be the subject is in fact the direct object.

transform(qauz,CI,In,Out,_)  % regular statement
    :- In=s(regular(Pos,MVinf,MVmods,no_roles(Nps,Pps))),
       simplify_Np(Nps,Sub,D_O), simplify_Pp(Pps,[],Pps2),
       ((MVmods=[], MVmods2=nil) ; MVmods2=MVmods),
       Out=qauz(CI,regular,MVinf,Pos,MVmods2,Sub,D_O,Pps2).

transform(qauz,CI,In,Out,_)  % regular_np phrase
    :- In=p(regular_np(Np)),
       simplify_Np(np_pp(sub,Np),Sub,nil),
       Out=qauz(CI,regular_np,nil,nil,nil,Sub,nil,nil).

transform(qauz,CI,In,Out,_)  % gerund_np phrase
    :- In=p(gerund_np(Pos,MVinf,MVmods,no_roles(Nps,Pps))),
       simplify_Np(Nps,nil,D_O), simplify_Pp(Pps,[],Pps2),
       ((MVmods=[], MVmods2=nil) ; MVmods2=MVmods),
       Out=qauz(CI,gerund_np,MVinf,Pos,MVmods2,nil,D_O,Pps2).

transform(qauz,CI,In,Out,_)  % vp phrase
    :- In=p(vp(Pos,MVinf,MVmods,no_roles(Nps,Pps))),
       simplify_Np(Nps,nil,D_O), simplify_Pp(Pps,[],Pps2),
       ((MVmods=[], MVmods2=nil) ; MVmods2=MVmods),
       Out=qauz(CI,vp,MVinf,Pos,MVmods2,nil,D_O,Pps2).


/******************************* SIMPLIFY_NP *********************************/

simplify_Np(In,Sub,D_O)
    :- In=[np_pp(X,Np1),np_pp(Y,Np2)],
       check_identifier(X), check_identifier(Y),
       ((Np1\==nil,
        simple_np_rep(Np1,HN,HN_Type,HN_Var,HN_Adjs,HN_Pps),
        Sub=sub(HN,HN_Type,HN_Var,HN_Adjs,HN_Pps)) ; Sub=nil),
       ((Np2\==nil,
        simple_np_rep(Np2,HN2,HN_Type2,HN_Var2,HN_Adjs2,HN_Pps2),
        D_O=d_o(HN2,HN_Type2,HN_Var2,HN_Adjs2,HN_Pps2)) ; D_O=nil).

simplify_Np(In,Sub,nil)
    :- Sub\==nil,
       (In=np_pp(X,Np) ; In=[np_pp(X,Np)]), check_identifier(X),
       simple_np_rep(Np,HN,HN_Type,HN_Var,HN_Adjs,HN_Pps),
       Sub=sub(HN,HN_Type,HN_Var,HN_Adjs,HN_Pps).

simplify_Np(In,nil,D_O)
    :- D_O\==nil,
       (In=np_pp(X,Np) ; In=[np_pp(X,Np)]),
       simple_np_rep(Np,HN,HN_Type,HN_Var,HN_Adjs,HN_Pps),
       D_O=d_o(HN,HN_Type,HN_Var,HN_Adjs,HN_Pps).


check_identifier(X) :- member(X,[sub,d_o,object,identifier,identified_object,
                                 compar_object1,compar_object2]).


/******************************* SIMPLIFY_PP *********************************/

simplify_Pp(X,[],nil) :- var(X).

simplify_Pp([],In,In).

simplify_Pp([H|R],In,Out)
    :- H=pp(Prep,Np),
       simple_np_rep(Np,HN,HN_Type,HN_Var,HN_Adjs,HN_Pps),
       append([pp(Prep,HN,HN_Type,HN_Var,HN_Adjs,HN_Pps)],In,In2),
       simplify_Pp(R,In2,Out).



/******************************* SIMPLE_NP_REP *******************************/

simple_np_rep(nil,nil,nil,nil,nil).

simple_np_rep(In,_,variable,nil,nil)
    :- In=..In2, nth1(1,In2,var).

simple_np_rep(In,HN,HN_Type,HN_Var,HN_Adjs,HN_Pps)
    :- (In=np_args(_,_,ng(HN_Adjs2,HN_Type2,HN2,HN_Var2,Pps2));
        (In=..In2, nth1(1,In2,HN_Type), nth1(2,In2,HN))),
       (((var(HN_Adjs2) ; HN_Adjs2=[]), HN_Adjs=nil) ; HN_Adjs=HN_Adjs2),
       ((var(HN_Type2), HN_Type=nil) ; HN_Type=HN_Type2),
       ((var(HN2), HN=nil) ; HN=HN2),
       (((var(HN_Var2) ; (HN_Var2=..X, nth1(2,X,Var), var(Var))), HN_Var=nil);
        HN_Var=HN_Var2),
       ((var(Pps2), HN_Pps=nil);
        simplify_Pp(Pps2,[],HN_Pps)).



/*****************************************************************************/



/***************************** HDI SPECIFICATIONS ****************************

        NOTE: these specifications have been provided by the HDI team.

        Contents: a specification for the representation supplied to the
                  HDI component by the parser (or the UI) as an S-expression

        Notes:
```

```
        - The use of the symbol '#' following a nonterminal should
          be interpreted as meaning "something of this syntactic
          category or LISP nil".
        - the "have" and "passive" statement forms have been omitted.
        - SD's "phrase" forms have been omitted.


<query> ::= (query (<statement>*) <hdi.question>) (this will not be supplied
            by SD)

<statement> ::= <is_adj.statement>
              | <is_a.statement>
              | <assign.statement>
              | <comparison.statement>
              | <there_is.statement>
              | <regular.statement>

<is_adj.statement> ::= (is_adj <truth.value> <np.equivalent> (<adjective>*))

<truth.value> ::= yes | no

<np.equivalent> ::=   <np.simple>
                    | (and <np.simple> <np.simple>+)

<np.simple>     ::=   <nounphrase>
                    | <variable>
                    | <propernoun>
                    | <pronoun>
                    | <personal.pronoun>
                    | <number>

<nounphrase> ::= (count_nounphrase <specifier> (<adjective>*) <count.noun>
                      <variable># (<prepositional.phrase>*))
               | (mass_nounphrase <specifier># (<adjective>*) <mass.noun>
                      <variable># (<prepositional.phrase>*))

<specifier> ::= the
              | all
              | <non.negative.integer>
              | (range <non.negative.integer># <non.negative.integer>#)

<variable> ::= (variable <string>)

<prepositional.phrase> ::= (<preposition> <np.equivalent>)

<propernoun> ::= (propernoun <atom>)

<pronoun> ::= (pronoun <atom>)

<personal.pronoun> ::= (personal_pronoun <atom>)

<number> ::= (number <integer or real number>)

<is_a.statement> ::= (is_a <truth.value> <var.or.pn> <nounphrase>)

<assign.statement> ::= (assign <truth.value> <nounphrase> <np.equivalent>
                               <np.equivalent>)

<comparison.statement> ::= (comparison <truth.value> <np.equivalent> <comparator>
                               <np.equivalent>)

<there_is.statement> ::= (there_is <truth.value> (<adverb>*) <nounphrase>)

<regular.statement> ::= (regular <truth.value> (<adverb>*) <infinitive.verb>
                               (<role>*))

<role> ::= (<role.name> <preposition> <np.equivalent>)
Note: for the time being, <role.name> will be 'nil'.

<hdi.question> ::= (hdi <truth.value> (<adverb>*) <infinitive.verb> (<role>*))

*******************************************************************************/


/****************************** HDI TRANSFORM ******************************/

transform(hdi,_,In,Out,_)  % q/4
    :- In=q(hdi,Pos,MVinf,MVmods,no_roles(Nps,Pps)),
       name(hdi,S_c), name(Pos,Pos_c), name(MVinf,MVinf_c),
       ((MVmods=[], MVmods_c=[40,41]) ; plist_llist(MVmods,[],MVmods_c)),
       roles_hdi(Nps,Pps,Roles_c),
       append_name([[40],S_c,[32],Pos_c,[32],MVmods_c,[32],MVinf_c,
                    [32],[40],Roles_c,[41],[41]],Out).
       % 40='(', 41=')'
       % [110,105,108]=nil


transform(hdi,_,In,Out,_)  % is_adj statement
    :- In=s(is_adj(Pos,np_pp(sub,Np),np_pp(attribute,Adjs))),
       name(is_adj,S_c), name(Pos,Pos_c),
       np_equivalent_hdi(Np,Np_c,_),
       plist_llist(Adjs,[],Adjs_c),
       append_name([[40],S_c,[32],Pos_c,[32],Np_c,
                    [32],Adjs_c,[41]],Out).


transform(hdi,_,In,Out,_)  % is_a statement
    :- In=s(is_a(Pos,np_pp(identifier,Np1),np_pp(identified_object,Np2))),
       name(is_a,S_c), name(Pos,Pos_c),
       ((Np1=var(X), variable_hdi(Np1,Sub_c,_)) ;
        (name(propernoun,P_c), name(Np1,N_c),
         append_name2([[40],P_c,[32],N_c,[41]],[],Sub_c))),
       nounphrase_hdi(Np2,D_O_c,_),
       append_name([[40],S_c,[32],Pos_c,[32],Sub_c,[32],D_O_c,[41]],Out).


transform(hdi,_,In,Out,_)  % assign statement
    :- In=s(assign(Pos,Np1,Np2,Np3)),
       name(assign,S_c), name(Pos,Pos_c),
       Np1=np_pp(object,Np11), Np2=np_pp(object_modifier,Np22),
       Np3=np_pp(value,Np33),
       nounphrase_hdi(Np11,Np11_c,_),
       np_equivalent_hdi(Np22,Np22_c,_),
       np_equivalent_hdi(Np33,Np33_c,_),
       append_name([[40],S_c,[32],Pos_c,[32],Np11_c,[32],Np22_c,[32],
                    Np33_c,[41]],Out).


transform(hdi,_,In,Out,_)  % comparison statement
    :- In=s(comparison(Pos,np_pp(compar_object1,Np1),
                       Compar_op,np_pp(compar_object2,Np2))),
       name(comparison,S_c), name(Pos,Pos_c), name(Compar_op,Compar_op_c),
       np_equivalent_hdi(Np1,Np1_c,_),
       np_equivalent_hdi(Np2,Np2_c,_),
       append_name([[40],S_c,[32],Pos_c,[32],Np1_c,[32],Compar_op_c,[32],
                    Np2_c,[41]],Out).


transform(hdi,_,In,Out,_)  % there_is statement
    :- In=s(there_is(Pos,MVmods,np_pp(object,Np))),
```

```
        name(there_is,S_c), name(Pos,Pos_c),
        ((MVmods=[], MVmods_c=[40,41]) ; plist_llist(MVmods,[],MVmods_c)),
        nounphrase_hdi(Np,Np_c,_),
        append_name([[40],S_c,[32],Pos_c,[32],MVmods_c,[32],Np_c,[41]],Out)


transform(hdi,_,In,Out,_)  % regular statement
    :- In=s(regular(Pos,MVinf,MVmods,no_roles(Nps,Pps))),
       name(regular,S_c), name(Pos,Pos_c), name(MVinf,MVinf_c),
       ((MVmods=[], MVmods_c=[40,41]) ; plist_llist(MVmods,[],MVmods_c)),
       roles_hdi(Nps,Pps,Roles_c),
       append_name([[40],S_c,[32],Pos_c,[32],MVmods_c,[32],MVinf_c,[32],
                    [40],Roles_c,[41],[41]],Out).


/*************************** NP and NP_EQUIVALENT ***************************/

np_equivalent_hdi(Np,Np_c,Role_c)
    :- np_conj_hdi(Np,Np_c,Role_c);
       np_simple_hdi(Np,Np_c,Role_c).



np_conj_hdi(Nps,Nps_c,Role_c)
    :- Nps=conj(CONJUNCTION,Simple_Nps), name(CONJUNCTION,S_c),
       process_conj_nps_hdi(Simple_Nps,Role_c,[],Simple_Nps_c),
       append_name2([[40],S_c,[32],Simple_Nps_c,[41]],[],Nps_c).


process_conj_nps_hdi([],_,In,Out) :- append_name2(In,[],Out).

process_conj_nps_hdi([H|R],Role_c,In,Out)
    :- np_simple_hdi(H,Np_c,Role_c),
       append([[32],Np_c],In,In2),
       process_conj_nps_hdi(R,Role_c,In2,Out).


np_simple_hdi(nil,[110,105,108],[110,105,108]).

np_simple_hdi(Np,Np_c,Role_c)
    :- variable_hdi(Np,Np_c,Role_c);
       proper_noun_hdi(Np,Np_c,Role_c);
       pronoun_hdi(Np,Np_c,Role_c);
       personal_pronoun_hdi(Np,Np_c,Role_c);
       number_hdi(Np,Np_c,Role_c);
       nounphrase_hdi(Np,Np_c,Role_c).


variable_hdi(Np,Np_c,Role_c)
    :- (Np=np_args(Role,SPEC,ng(_,HN_Type,HN,var(X),_)) ; Np=var(X)),
       nonvar(X), var(SPEC), var(HN_Type), var(HN),
       ((var(Role), Role_c=[110,105,108]) ; name(Role,Role_c)),
       name(variable,S_c), name(X,X_c),
       append_name2([[40],S_c,[32],X_c,[41]],[],Np_c).


proper_noun_hdi(Np,Np_c,Role_c)
    :- Np=np_args(Role,_,ng(_,proper_quiz_noun,X,_,_)),
       ((var(Role), Role_c=[110,105,108]) ; name(Role,Role_c)),
       name(propernoun,S_c), name(X,X_c),
       append_name2([[40],S_c,[32],X_c,[41]],[],Np_c).


pronoun_hdi(Np,Np_c,Role_c)
    :- Np=np_args(Role,_,ng(_,pronoun,X,_,_)),
       ((var(Role), Role_c=[110,105,108]) , name(Role,Role_c)),
       name(pronoun,S_c), name(X,X_c),
       append_name2([[40],S_c,[32],X_c,[41]],[],Np_c).


personal_pronoun_hdi(Np,Np_c,Role_c)
    :- (Np=np_args(Role,_,ng(_,pers_pro,X,_,_)) ; Np=pers_pro(X)), nonvar(X),
       ((var(Role), Role_c=[110,105,108]) ; name(Role,Role_c)),
       name(personal_pronoun,S_c), name(X,X_c),
       append_name2([[40],S_c,[32],X_c,[41]],[],Np_c).


number_hdi(Np,Np_c,Role_c)
    :- Np=np_args(Role,_,ng(_,number,X,_,_)),
       ((var(Role), Role_c=[110,105,108]) ; name(Role,Role_c)),
       name(number,S_c), name(X,X_c),
       append_name2([[40],S_c,[32],X_c,[41]],[],Np_c).


nounphrase_hdi(Np,Np_c,Role_c)
    :- Np=np_args(Role,Spec,ng(Adjs,c_noun,HN,Var,Pps)),
       ((var(Role), Role_c=[110,105,108]) ; name(Role,Role_c)),
       name(count_nounphrase,S_c), spec_hdi(Spec,Spec_c),
       ((Adjs=[], Adjs_c=[40,41]) ; plist_llist(Adjs,[],Adjs_c)),
       name(HN,HN_c),
       ((nonvar(Var), Var=var(X), nonvar(X), name(X,X_c), name(variable,V_c),
         append_name2([[40],V_c,[32],X_c,[41]],[],Var_c));
        Var_c=[110,105,108]),
       ((nonvar(Pps), pp_hdi(Pps,[],Pps_c)) ; Pps_c=[32]),
       append_name2([[40],S_c,[32],Spec_c,[32],Adjs_c,[32],
                     HN_c,[32],Var_c,[32],[40],Pps_c,[41],[41]],[],Np_c)

nounphrase_hdi(Np,Np_c,Role_c)
    :- Np=np_args(Role,Spec,ng(Adjs,m_noun,HN,Var,Pps)),
       ((var(Role), Role_c=[110,105,108]) ; name(Role,Role_c)),
       name(mass_nounphrase,S_c),
       ((var(Spec), Spec_c=[110,105,108]) ; spec_hdi(Spec,Spec_c)),
       ((Adjs=[], Adjs_c=[40,41]) ; plist_llist(Adjs,[],Adjs_c)),
       name(HN,HN_c),
       ((nonvar(Var), Var=var(X), nonvar(X), name(X,X_c), name(variable,V_c),
         append_name2([[40],V_c,[32],X_c,[41]],[],Var_c));
        Var_c=[110,105,108]),
       ((nonvar(Pps), pp_hdi(Pps,[],Pps_c)) ; Pps_c=[32]),
       append_name2([[40],S_o,[32],Spec_c,[32],Adjs_c,[32],
                     HN_c,[32],Var_c,[32],[40],Pps_c,[41],[41]],[],Np_c)



spec_hdi(Spec_in,Spec_out)
    :- Spec_in=spec(X), name(range,S_c), name(nil,N_c),
       ((X=int_or_more(Y), name(Y,Y_c),
         append_name2([[40],S_c,[32],Y_c,[32],N_c,[41]],[],Spec_out)),
        (X=zero_or_more, name(0,Z_c),
         append_name2([[40],S_c,[32],Z_c,[32],N_c,[41]],[],Spec_out)),
        (X=one_or_more, name(1,O_c),
         append_name2([[40],S_c,[32],O_c,[32],N_c,[41]],[],Spec_out));
        (X=range(Y,Z), name(Y,Y_c), name(Z,Z_c),
         append_name2([[40],S_c,[32],Y_c,[32],Z_c,[41]],[],Spec_out)),
        (X=some, name(1,O_c),
         append_name2([[40],S_c,[32],O_c,[32],N_c,[41]],[],Spec_out));
        (X=unspec, Spec_out=N_c);
        ((X=each ; X=every ; X=tsoa), name(all,Spec_out));
        ((X=a ; X=an), name(1,Spec_out));
        (X=eight, name(8,Spec_out));
        (X=five, name(5,Spec_out));
        (X=four, name(4,Spec_out));
```

```
        (X=nine, name(9,Spec_out));
        (X=one, name(1,Spec_out));
        (X=seven, name(7,Spec_out));
        (X=six, name(6,Spec_out));
        (X=ten, name(10,Spec_out));
        (X=three, name(3,Spec_out));
        (X=two, name(2,Spec_out));
        (X=zero, name(0,Spec_out));
        name(X,Spec_out)).


/****************************** PP_ROLES and PP ******************************/

pp_roles_hdi(X,[],[40,41]) :- var(X).

pp_roles_hdi([],In,Out) :- append_name2(In,[],Out).

pp_roles_hdi([H|R],In,Out)
    :- H=pp(Prep,Np), name(Prep,Prep_c),
       (np_conj_hdi(Np,Np_c,Np_role_c) ; np_simple_hdi(Np,Np_c,Np_role_c)),
       append([[32],[40],Np_role_c,[32],Prep_c,[32],Np_c,[41]],In,In2),
       pp_roles_hdi(R,In2,Out).


pp_hdi(X,[],[40,41]) :- var(X).

pp_hdi([],In,Out) :- append_name2(In,[],Out).

pp_hdi([H|R],In,Out)
    :- H=pp(Prep,Np), name(Prep,Prep_c),
       (np_conj_hdi(Np,Np_c,_) ; np_simple_hdi(Np,Np_c,_)),
       append([[32],[40],Prep_c,[32],Np_c,[41]],In,In2),
       pp_hdi(R,In2,Out).


/********************************** ROLES ***********************************/

roles_hdi(Nps,Pps,Roles_c)
    :- Nps=[np_pp(sub,Np1),np_pp(d_o,Np2)],
       np_equivalent_hdi(Np1,Np1_c,Np1_role_c), name(sub,S_c),
       np_equivalent_hdi(Np2,Np2_c,Np2_role_c), name(d_o,D_O_c),
       pp_roles_hdi(Pps,[],Pps_c),
       append([[40],Np1_role_c,[32],S_c,[32],Np1_c,[41]],[],Roles_c1),
       append(Roles_c1,[[32],[40],Np2_role_c,[32],D_O_c,[32],
                        Np2_c,[41],[32]],Roles_c2),
       append(Roles_c2,[Pps_c],Roles_c3), append_name2(Roles_c3,[],Roles_c).


/****************************** Home made Utilities ***************************/

% PLIST_LLIST: transforms a Prolog list (of characters) into a Lisp list
plist_llist([],L,X_c) :- append(LR,[32],L), append([40],LR,L2),
                         reverse(L2,L3), append([41],L3,L4), reverse(L4,X_c).

plist_llist([H|R],L,X_c) :- name(H,H_c),
                            append([32],L,L2), append(H_c,L2,L3),
                            plist_llist(R,L3,X_c).


% APPEND_NAME: appends lists of characters and 'name' the resulting list.
append_name(In,Out) :- append_name2(In,[],Out_c), name(Out,Out_c).

append_name2([],L,L).
append_name2([X|R],L,Out) :- append(X,L2,Out), append_name2(R,L,L2).


/*****************************************************************************/

% If it failed before, let it go through !
transform(_,_,In,In,ERR)
     :- error_msg(invalid_qauz_hdi_transformation,ERR,[], ).

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

```
/* ------------------------------------------------------------------------
   Author : Sylvain Delisle
   Project: SEQAP parser
   File   : hierarchy.pl
   Version: November 4, 1986

This file contains a simplified hierarchy of QUIZ objects. The latter is used
for a simple post-processing verification where the acceptability of head
nouns is verified. This post-processing is a simple semantic check.

------------------------------------------------------------------------*/


/********************************* TOP-LEVEL *********************************/

controlbreak_reportobject(X) :- control_break(X) ; report_object(X).

item_file(X) :- item(X) ; file(X).

item_file_report(X) :- item(X) , file(X) ; report(X).

item_file_reportobject(X) :- item(X) ; file(X) ; report_object(X).

item_number(X) :- item(X) ; number9(X).

item_report_reportobject(X) :- item(X) ; report(X) ; report_object(X).

item_reportobject(X) :- item(X) ; report_object(X).

quiz_report(X) :- quiz(X) ; report(X).

quizstat_report(X) :- quiz_statement(X) ; report(X).

quizstat_quizoption(X) :- quiz_statement(X) , quiz_option(X).


/********************************* HIERARCHY *********************************/


device(X) :- member(X,[device,disc,disk,printer,screen,terminal]).


file(X) :- member(X,[data_file,display_file,file,subfile]).


item(X) :- member(X,[item,record_item,sort_key]) ; summary_item(X).

summary_item(X) :- member(X,[average,count,maximum,minimum,
                             percentage,subtotal,summary_item]).


report_object(X) :- member(X,[line,page,report_column,
                              report_group,report_object]) ;
                    footing(X) ; heading(X) , skip1(X).

footing(X) :- X=footing ; X=page_footing.

heading(X) :- X=heading ; X=page_heading.

skip1(X) :- member(X,[line_skip,page_skip,skip]).


quiz_statement(X) :- member(X,['ACCESS','CHOOSE','DEFINE','DISPLAY',
                               'EXECUTE','FINAL_FOOTING',
                               'FOOTING','HEADING','INITIAL_HEADING',
                               'PAGE_FOOTING','PAGE_HEADING','REPORT',
                               'SAVE','SELECT','SET','SORT','SORTED']).

quiz_option(X) :- member(X,['LINK']).


number9(X):- X=number ; number1(X).


control_break(control_break).

object(_).  % For the time being, it could be anything.

order(order).

quiz('QUIZ').

report(report).

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

```
/* ------------------------------------------------------------------------
   Author : Sylvain Delisle
   Project: SEQAP parser
   File   : role-table.pl
   Version: November 4, 1986

This file contains the SEQAP role table where each verb, in the set of 34 verbs
considered for P1, is associated with its legal prepositions,
its mandatory/optional flag, its class of potential fillers, and its role name.

The format is as follows: (where 'nomi' means nominal group)
role_table(MVinf,roles([nomi(Prep1,M/O,Filler_Type,Role_Name1),
                        nomi(Prep2, ...                      ),
                        ...])).

------------------------------------------------------------------------*/


/******************************** ROLE TABLE ********************************/

role_table(access,
           roles([nomi(d_o,mandatory,file,source_file),
                  nomi(with,optional,quiz_statement,instrument),
                  nomi(using,optional,quiz_statement,instrument)])).

role_table(add,
           roles([nomi(d_o,mandatory,quiz_option,added_object),
                  nomi(to,optional,quiz_statement,recipient_object)])).

role_table(add,
           roles([nomi(d_o,mandatory,item_number,argument1),
                  nomi(to,optional,item_number,argument2),
                  nomi(after,optional,controlbreak_reportobject,location),
                  nomi(before,optional,controlbreak_reportobject,location)])).

role_table(average,
           roles([nomi(d_o,mandatory,item,item),
                  nomi(in,optional,report_object,location),
                  nomi(at,optional,controlbreak_reportobject,location)])).

role_table(choose,
           roles([nomi(d_o,mandatory,item_file,object)])).

role_table(compare,
           roles([nomi(d_o,mandatory,item_number,item1),
                  nomi(to,optional,item_number,item2),
                  nomi(with,optional,item_number,item2)])).

role_table(count,
           roles([nomi(d_o,mandatory,item,item),
                  nomi(at,optional,controlbreak_reportobject,location)])).

role_table(create,
           roles([nomi(d_o,mandatory,item_file_reportobject,object)])).

role_table(define,
           roles([nomi(d_o,mandatory,item,item),
                  nomi(in,optional,report_object,location),
                  nomi(with,optional,quiz_statement,instrument),
                  nomi(using,optional,quiz_statement,instrument)])).

role_table(direct,
           roles([nomi(d_o,mandatory,report,report),
                  nomi(to,optional,device,device)])).

role_table(display,
           roles([nomi(d_o,mandatory,item_reportobject,object),
                  nomi(on,optional,device,device)])).

role_table(divide,
           roles([nomi(d_o,mandatory,item_number,argument1),
                  nomi(by,optional,item_number,argument2)])).

role_table(execute,
           roles([nomi(d_o,mandatory,quizstat_quizoption,object)])).

role_table(find,
           roles([nomi(d_o,mandatory,item,item),
                  nomi(in,optional,file,location),
                  nomi(with,optional,quiz_statement,instrument)])).

role_table(format,
           roles([nomi(d_o,mandatory,item_report_reportobject,object),
                  nomi(with,optional,quizstat_quizoption,instrument),
                  nomi(using,optional,quizstat_quizoption,instrument)])).

role_table(get,
           roles([nomi(d_o,mandatory,object,object)])).

role_table(link,
           roles([nomi(d_o,mandatory,file,co_source_file),
                  nomi(to,optional,file,co_source_file),
                  nomi(with,optional,file,co_source_file)])).

role_table(multiply,
           roles([nomi(d_o,mandatory,item_number,argument1),
                  nomi(by,optional,item_number,argument2)])).

role_table(print,
           roles([nomi(d_o,mandatory,item_file_reportobject,object),
                  nomi(on,optional,device,device),
                  nomi(at,optional,controlbreak_reportobject,location),
                  nomi(in,optional,report_object,location),
                  nomi(on,optional,report_object,location),
                  nomi(under,optional,report_object,location)]))

role_table(read,
           roles([nomi(d_o,mandatory,file,file)])).

role_table(report,
           roles([nomi(d_o,mandatory,item,item),
                  nomi(to,optional,file,destination_file),
                  nomi(for,optional,item,condition),
                  nomi(from,optional,file,source_file),
                  nomi(in,optional,order,order),
                  nomi(on,optional,report_object,location),
                  nomi(at,optional,controlbreak_reportobject,location),
                  nomi(in,optional,report_object,location)])).

role_table(reset,
           roles([nomi(d_o,mandatory,item,item),
                  nomi(at,optional,controlbreak_reportobject,location),
                  nomi(to,optional,item_number,value),
                  nomi(with,optional,item_number,value),
                  nomi(using,optional,item_number,value)])).

role_table(right_justify,
           roles([nomi(d_o,mandatory,item,item)])).

role_table(run,
           roles([nomi(d_o,mandatory,report,report)])).
```

```
role_table(save,
          roles([nomi(d_o,mandatory,quizstat_report,object),
                 nomi(in,optional,file,location)])).

role_table(select,
          roles([nomi(d_o,mandatory,item,item),
                 nomi(from,optional,file,file),
                 nomi(on,optional,item,condition)])).

role_table(set,
          roles([nomi(d_o,mandatory,item,item),
                 nomi(to,optional,item_number,value),
                 nomi(with,optional,item_number,value),
                 nomi(using,optional,item_number,value)])).

role_table(skip,
          roles([nomi(d_o,mandatory,report_object,object),
                 nomi(to,optional,report_object,direct_location),
                 nomi(after,optional,report_object,relative_location),
                 nomi(before,optional,report_object,relative_location)])).

role_table(sort,
          roles([nomi(d_o,mandatory,item,item),
                 nomi(in,optional,order,order),
                 nomi(on,optional,item,sort_field)])).

role_table(specify,
          roles([nomi(d_o,mandatory,item,item),
                 nomi(in,optional,quizstat_report,location)])).

role_table(start,
          roles([nomi(d_o,mandatory,quiz_report,object)])).

role_table(store,
          roles([nomi(d_o,mandatory,object,object),
                 nomi(in,optional,file,location)])).

role_table(subtotal,
          roles([nomi(d_o,mandatory,item,item),
                 nomi(in,optional,report_object,location),
                 nomi(for,optional,item,condition),
                 nomi(at,optional,controlbreak_reportobject,location)])).

role_table(total,
          roles([nomi(d_o,mandatory,item,item),
                 nomi(in,optional,report_object,location),
                 nomi(for,optional,item,condition),
                 nomi(at,optional,controlbreak_reportobject,location)]))

role_table(use,
          roles([nomi(d_o,mandatory,quizstat_quizoption,object),
                 nomi(at,optional,controlbreak_reportobject,location)]))
                c
/*****************************************************************************/
```

```
/* ---------------------------------------------------------------------------
   Author : Sylvain Delisle
   Project: SEQAP parser
   File   : role-system.pl
   Version: November 11, 1986

This file contains the role system which perfoms a simple semantic verification
on the main verb's nominal groups (direct object, prepositional phrases).
In each of the latter, we check whether the head noun (HN) belongs to the
filler type found in the role table. No verification is done for prepositional
phrases that modify nouns

The filler types considered for our prototype system are organized in a
hierarchy of QUIZ objects presented in "hierarchy.pl".
Thus, the role system is composed of this file, the role table "role-table.pl",
and the hierarchy of QUIZ objects "hierarchy pl".

---------------------------------------------------------------------------*/


/***************************** ROLE_POST_PROCESSING ***************************/

role_post_processing(MVinf,Nps,Pps,Roles,ERR)
    :- get_all_possible_meanings(MVinf,ERR),
       ((var(ERR),
         try_find_intended_meaning(MVinf,Nps,Pps,Roles,ERR));
        (Roles=no_roles(Nps,Pps),
         error_msg(error_during_role_postprocessing,ERR,[],_))),
       clean_up_main_verb_meanings.
       % At the beginning all possible meanings are fetched and kept
       % globally in "main_verb_meaning(X)". This must be done that
       % way because of the error treatment which does not allow for
       % backtracking (ERR variable).



/************************** GET_ALL_POSSIBLE_MEANINGS *************************/

get_all_possible_meanings(MVinf,_)
    :- role_table(MVinf,Meaning),
       assert(main_verb_meaning(Meaning)),
       fail.

get_all_possible_meanings(MVinf,ERR)
    :- main_verb_meaning(Meaning);
       error_msg(warning__no_roles_for_this_verb(MVinf),ERR,[],_).



/************************** TRY_FIND_INTENDED_MEANING *************************/

try_find_intended_meaning(MVinf,Nps,Pps,Roles,ERR)
    ,-  % MR=Mandatory Roles (list)
        retract(main_verb_meaning(roles(Meaning))),
        get_list_mandatory_roles(Meaning,[],MR),
        process_Nps(MVinf,Meaning,Nps,Nps_out,[],MR_found1,ERR2),
        process_Pps(MVinf,Meaning,Pps,[],Pps_out,MR_found1,MR_found2,ERR2),
        append(Nps_out,Pps_out,Roles2),
        verify_mandatory_roles(MR,MR_found2,ERR2),
        ((var(ERR2), Roles=roles(Roles2));  % Success
         (main_verb_meaning(X),             % Try next possible meaning
          try_find_intended_meaning(MVinf,Nps,Pps,Roles,ERR));
         (Roles=no_roles(Roles2),           % Input was not meaningful
          ERR=ERR2)).
```

```
/************************** GET_LIST_MANDATORY_ROLES ************************/

get_list_mandatory_roles([],MR_in,MR_in).

get_list_mandatory_roles([H|R],MR_in,MR_out)
    :- H=nomi(Prep,M_O,_,Role_Name),
       ((M_O=mandatory,
         PREP=..[Prep,Role_Name], append([PREP],MR_in,MR_in2));
        MR_in2=MR_in),
       get_list_mandatory_roles(R,MR_in2,MR_out).


/********************************* PROCESS_NPS ******************************/

process_Nps(MVinf,Role_Slots,Nps_in,Nps_out,MR_in,MR_out,ERR)
    :- Nps_in=[Nps1,Nps2],
       Nps1=np_pp(Prep1,Np1), Nps2=np_pp(Prep2,Np2),
       get_role_rep(MVinf,Role_Slots,Prep1,Np1,Np1_out,MR_in,MR_in2,ERR),
       get_role_rep(MVinf,Role_Slots,Prep2,Np2,Np2_out,MR_in2,MR_out,ERR),
       Nps_out=[nomi(Prep1,Np1_out),nomi(Prep2,Np2_out)].

process_Nps(_,_,Nps_in,Nps_in,MR_in,MR_in,ERR)
    :- error_msg(error_in_process_Nps(Nps_in),ERR,[],_).


/********************************* GET_ROLE_REP *****************************/

get_role_rep(_,_,sub,Np_in,Np_in,MR_in,MR_in,_).

get_role_rep(MVinf,Role_Slots,Prep,nil,Np_out,MR_in,MR_in,ERR)
    :- role_check(Role_Slots,nil,Prep,Role_Name,_,_,ERR),
       Np_out=np_args(Role_Name,nil).

get_role_rep(MVinf,Role_Slots,Prep,Np_in,Np_out,MR_in,MR_out,ERR)
    :- Np_in=np_args(_,SPEC,NG),
       role_check(Role_Slots,Np_in,Prep,Role_Name,MR_in,MR_out,ERR),
       Np_out=np_args(Role_Name,SPEC,NG).

get_role_rep(MVinf,Role_Slots,Prep,Np_in,Np_out,MR_in,MR_out,ERR)
    :- Np_in=conj(CONJUNCTION,List_Nps),
       process_conj_Nps(MVinf,Role_Slots,List_Nps,[],List_Nps_out,
                        Prep,MR_in,MR_out,ERR),
       Np_out=conj(CONJUNCTION,List_Nps_out).

get_role_rep(_,_,Prep,Np_in,Np_in,MR_in,MR_in,ERR)
    :- error_msg(error_in_get_role_rep(Prep,Np_in),ERR,[],_).


/************************** PROCESS_CONJ_NPS ********************************/

process_conj_Nps(_,_,[],List_Nps_in,List_Nps_in,_,MR_in,MR_in,_).

process_conj_Nps(MVinf,Role_Slots,[H|R],List_Nps_in,List_Nps_out,
                 Prep,MR_in,MR_out,ERR)
    :- get_role_rep(MVinf,Role_Slots,Prep,H,H2,MR_in,MR_in2,ERR),
       append(List_Nps_in,[H2],List_Nps_in2),
       process_conj_Nps(MVinf,Role_Slots,R,List_Nps_in2,List_Nps_out,
                        Prep,MR_in2,MR_out,ERR).

process_conj_Nps(_,_,L,List_Nps_in,List_Nps_in,Prep,MR_in,MR_in,ERR)
    :- error_msg(error_in_process_conj_Nps(Prep,L),ERR,[],_).
```

```
/********************************* PROCESS_PPS ******************************/

process_Pps(_,_,[],Pps_in,Pps_in,MR_in,MR_in,_).

process_Pps(MVinf,Role_Slots,[H|R],Pps_in,Pps_out,MR_in,MR_out,ERR)
     - H=pp(Prep,Np),
       get_role_rep(MVinf,Role_Slots,Prep,Np,Np_out,MR_in,MR_in2,ERR),
       H2=nomi(Prep,Np_out),
       append(Pps_in,[H2],Pps_in2),
       process_Pps(MVinf,Role_Slots,R,Pps_in2,Pps_out,MR_in2,MR_out,ERR)

process_Pps(MVinf,Role_Slots,[H|R],Pps_in,Pps_out,MR_in,MR_out,ERR)
    :- H=pp(Prep,conj(CONJUNCTION,List_Pps)),
       process_Pps(MVinf,Role_Slots,List_Pps,[],List_Pps_out,MR_in,MR_in2,ERR),
       H2=nomi(Prep,conj(CONJUNCTION,List_Pps_out)),
       append(Pps_in,[H2],Pps_in2),
       process_Pps(MVinf,Role_Slots,R,Pps_in2,Pps_out,MR_in2,MR_out,ERR)

process_Pps(_,_,L,Pps_in,Pps_in,MR_in,MR_in,ERR)
    :- error_msg(error_in_process_Pps(L),ERR,[],_).


/********************************* ROLE_CHECK *******************************/

role_check(_,nil,Prep,Prep,MR_in,MR_in,_).

role_check(Role_Slots,Np,Prep,Role_Name,MR_in,MR_out,ERR)
    :- fetch_role_slots(Role_Slots,Prep,[],Slots,ERR),
       check_potential_role(Slots,Np,Prep,Role_Name,MR_in,MR_out,ERR).

role_check(_,Np,Prep,_,MR_in,MR_in,ERR)
    :- error_msg(error_in_role_check(Prep,Np),ERR,[],_).


/************************** CHECK_POTENTIAL_ROLE ****************************/

check_potential_role([],_,_,_,_,_,ERR).

check_potential_role([H|R],Np,Prep,Role_Name,MR_in,MR_out,ERR)
    :- H=nomi(Prep,M_O,Filler_Type,Role_Name2),
       ((M_O=mandatory,
         PREP=..[Prep,Role_Name2], append([PREP],MR_in,MR_in2));
        MR_in2=MR_in),
       validate_filler_type(Np,Filler_Type,ERR2),
       ((var(ERR2), Role_Name=Role_Name2, MR_out=MR_in2),
        (R=[], MR_out=MR_in2, ERR=ERR2),
        check_potential_role(R,Np,Prep,Role_Name,MR_in2,MR_out,ERR)),


/********************************* FETCH_ROLE_SLOTS *************************/

fetch_role_slots([],Wanted_Slot,[],[],ERR)
    :- error_msg(illegal_preposition(Wanted_Slot),ERR,[],_).

fetch_role_slots([],_,WS_in,WS_in,_).

fetch_role_slots([H|R],Wanted_Slot,WS_in,WS_out,ERR)
    :- ((H=nomi(Wanted_Slot,M_O,Filler_Type,Role_Name),
         append([H],WS_in,WS_in2));
        WS_in2=WS_in),
       fetch_role_slots(R,Wanted_Slot,WS_in2,WS_out,ERR).
```

```
/*************************** VALIDATE_FILLER_TYPE ***************************/

validate_filler_type(Np,Filler_Type,ERR)
    :- member(Filler_Type,[quiz_option,quiz_statement,
                           quizstat_quizoption,quizstat_report]),
       ((Np=np_args(_,_,ng(_,HN_Type,_,_,_)), HN_Type=quiz_code_errmessage);
        (Np=np_args(_,_,ng([Adj],_,HN,_,_)), (HN=statement ; HN=option))),
       (var(Adj);
        (Verif=..[Filler_Type,Adj],
         (call(Verif);
          ((HN=statement,
            error_msg(semantic_invalid_quiz_statement(Adj),ERR,[],_));
           error_msg(semantic_invalid_quiz_option(Adj),ERR,[],_))))).

validate_filler_type(Np,Filler_Type,ERR)
    :- Np=np_args(_,_,ng(_,_,HN,_,_)),
       name(HN,HN_Chars), reverse(HN_Chars,[First|Rest]),
       ((First=115, reverse(Rest,Rest2), name(HN2,Rest2)) ; HN2=HN),
       Verif=..[Filler_Type,HN2],
       (call(Verif) ; error_msg(semantic_invalid_noun(HN2),ERR,[],_)).

validate_filler_type(Np,Filler_Type,ERR)
    :- error_msg(error_in_validate_filler_type(Np,Filler_Type),ERR,[],_).


/************************** VERIFY_MANDATORY_ROLES **************************/

verify_mandatory_roles([],_,_).

verify_mandatory_roles([H|R],MR_found,ERR)
    :- (member(H,MR_found), verify_mandatory_roles(R,MR_found,ERR));
       error_msg(missing_mandatory_role(H),ERR,[],_).


/********************************* UTILITIES ********************************/

clean_up_main_verb_meanings :- retract(main_verb_meaning(_)), fail.
clean_up_main_verb_meanings.

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

```
/* Author : Sylvain Delisle
   Project: SEQAP parser
   File   : driver.pl
   Version: November 4, 1986

   This file contains the high-level driver for the SEQAP parser.
*/


/************************ USING THE PARSER INTERACTIVELY ********************/
/* To use the parser interactively:                                         */
/* parse(X,Y) where X=hdi, qauz, or seqap  AND  Y=true or false             */
/* When X=hdi or qauz, the semantic check is not done. Instead, a tranforma- */
/* tion on the output form is perfomed. For HDI, the output is translated   */
/* into a Lisp expression. For QAUZ, the output (Prolog form) is simplified. */
/* When X=seqap the semantic check is done and the output is augmented with  */
/* role names.                                                              */


parse(X,_)         :- var(X), print_err_mess.
parse(hdi,Reset)   :- (Reset=true ; Reset=false), !, go(hdi,Reset).
parse(qauz,Reset)  :- (Reset=true ; Reset=false), !, go(qauz,Reset).
parse(seqap,Reset):- (Reset=true ; Reset=false), !, go(seqap,Reset).
parse(_,_)         :- print_err_mess.

print_err_mess :- nl, write('Invalid arguments: use "parse(QAS,Reset)".'),
                  nl, write('where QAS=hdi/qauz and Reset=true/false.'), nl



go(QAS,Reset)
        :- nl, write('SEQAP PARSER;'), !,
           statistics(runtime,_),
           lex_phase(nil,_,OI,CI,SML,IT,S1),
           statistics(runtime,RT1),
           nl, write('* Lexical Phase *'), nl,
           write('OI : '), writeq(OI), nl,     % OI: Original Input
           write('CI : '), writeq(CI), nl,     % CI: Canonical Input
           write('SML: '), writeq(SML), nl,    % SML: Synonym Mapping List
           write('IT : '), writeq(IT), nl,     % IT: Input Type
           write('S1 : '), writeq(S1), nl,     % S1: Success Flag
           write('RT1: '), writeq(RT1), nl, !,% RT1: Execution time (lex_phase)

           ((S1=success,
             nl, write('* Parsing *'), nl,
             statistics(runtime,_),
             parser(CI,IT,Reset,QAS,PO,QS,S2),
             statistics(runtime,RT2),          % RT2: Execution time (rest)
             write('QAS: '), writeq(QAS), nl, % QAS: seqap, hdi, or qauz
             write('PO : '), writeq(PO), nl,  % PO: parser's output
             write('QS . '), writeq(QS), nl,  % QS: Quiz stuff
             write('S2 : '), writeq(S2), nl,  % S2: Success Flag
             write('RT2: '), writeq(RT2), nl, nl);

            (write('Parsing stopped: error found during the lexical phase '),
             nl, nl)).


/************************ USING THE PARSER AS A TRANSLATOR ******************/
/* To use the parser as a translator:                                       */
/* trans(String,QAS,Reset,OI,CI,SML,IT,S1,PO,QS,S2,RT).                     */
/* (See above for the meaning of the variables)                             */
/*                                                                          */
/* 'Bound' variables before the call:                                       */
/*  String must be instantiated to the input to be parsed,                  */
```

```
/*  example: String="Why am I getting an error message ?";                  */
/*  QAS must be instantiated to either hdi or qauz.                          */
/*  The third parameter (Reset) is a boolean flag for indicating whether     */
/*  you want the symbol table (for user variables) to be deleted for each    */
/*  new input that you want to translate. If set to 'true' the parser will   */
/*  not recognized variables introduced in other inputs than the current one.*/
/*                                                                           */
/*  All the other variables will be instantiated after execution,            */
/*  see previous section for their meaning.                                  */
/*  Note: RT is runtime (CPU time in millisec.) used for complete processing.*/


trans(String,QAS,Reset,OI,CI,SML,IT,S1,PO,QS,S2,RT)
     :- statistics(runtime,_),
        lex_phase(translator,String,OI,CI,SML,IT,S1), !,
        ((S1=success,
          parser(CI,IT,Reset,QAS,PO,QS,S2), !, statistics(runtime,RT));
         (PO=nil, QS=nil, S2=nil, RT=nil)).

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

APPENDIX E : QAUZ SYNTAX-QUESTION GRAMMAR RULES

QAUZ SYNTAX-QUESTION GRAMMAR RULES

qauz_syntax_question --> "what is the syntax of", noun_phrase.

(e.g. What is the syntax of the REPORT statement ?)

qauz_syntax_question --> "what is the complete syntax of", noun_pharse.

(e.g. What is the complete syntax of ACCESS ?)

qauz_syntax_question --> "what is", noun_phrase, "a component of".

(e.g. What is LINK a component of ?)

qauz_syntax_question --> "what are the different kinds of", noun_phrase, "statement".

(e.g. What are the different kinds of ACCESS statement ?)

qauz_syntax_question --> "where is", noun_phrase, "used".

(e.g. Where is LINK used ?)

qauz_syntax_question --> "what kinds of statements are there".

APPENDIX F : EXAMPLES OF EXECUTION

Appendix F: Elements of the Output

Here is a brief description on the output provided by SEQAP:

a-) Lexical Phase

CI : Canonical Input is the original input in which compound words and synonyms have been replaced;

SML: Synonym Mapping List is a list of pairs in which a word given by the user is paired with its canonical synonym when necessary;

IT : Input Type is either *question*, *statement*, or *phrase*;

S1 : Success 1 is *success* if the lexical phase was successful. Otherwise, Success 1 is an error message;

RT1: Runtime (in ms) for lexical phase. RT1 is presented as the list [A,B] where A is the total CPU time used so far and B is the CPU time used for the lexical phase only.

b-) Parsing (Syntactic and Post-processing Phases)

QAS: Question Answering System is either *hdi*, *qauz*, or *seqap*;

QS : QUIZ 'Stuff' contains a QUIZ error message or a fragment of QUIZ code found in the input;

S2 : Success 2 is *success* if the parsing (after the lexical phase) was successful. Otherwise, Success 2 is an error message;

RT2: Same as RT1 except that the CPU time is for the parsing only (after the lexical phase);

PO : Parser's Output.

```
Script started on Mon Nov 24 19 51 51 1986
uotcsib# seqap-t

yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase(,)]
| 'employee' is a record item of a file 'F'

* Lexical Phase *
CI  [var(employee),is,a,record_item,of,a,file,var('F')]
SML []
IT  statement
S1  success
RT1 [183,150]

* Parsing *
QAS seqap
QS  []
S2  success
RT2 [433,200]
PO
      s
        is_a
          yes
          np_pp
            identifier
            var
              employee
          np_pp
            identified_object
            np_args
              nil
              spec
                a
              ng
                nil
                c_noun
                record_item
                var
                  employee
                [
                  pp
                    of
                    np_args
                      nil
                      spec
                        a
                      ng
                        nil
                        c_noun
                        file
                        var
                          'F'
                        nil


yes
| ?- parse(seqap,false)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
| 'cb' is a control break of a file 'F2'

* Lexical Phase *
CI  [var(cb),is,a,control_break,of,a,file,var('F2')]
SML []
IT  statement
S1  success
RT1 [933,183]

* Parsing *
QAS seqap
QS  []
S2  success
RT2 [1183,183]
PO
      s
        is_a
          yes
          np_pp
            identifier
            var
              cb
          np_pp
            identified_object
            np_args
              nil
              spec
                a
              ng
                nil
                c_noun
                control_break
                var
                  cb
                [
                  pp
                    of
                    np_args
                      nil
                      spec
                        a
                      ng
                        nil
                        c_noun
                        file
                        var
                          'F2
                        nil
                ]


yes
| ?- parse(seqap,false)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase(,)]
| How do I report 'employee' at 'cb' to a subfile ?

* Lexical Phase *
CI  [how,do,i,report,var(employee),at,var(cb),to,a,subfile]
SML []
IT  question
S1  success
RT1 [1733,200]

* Parsing *
QAS seqap
QS  []
S2  success
RT2: [1933,183]
```

```
PO
      q
        hdi
        yes
        report
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               item
               spec
                 a
               ng
                 nil
                 c_noun
                 record_item
                 var
                   employee
                  [
                   pp
                     of
                     np_args
                       nil
                       spec
                         a
                       ng
                         nil
                         c_noun
                         file
                         var
                           F
                         nil
                ]
           nomi
             to
             np_args
               destination_file
               spec
                 a
               ng
                 nil
                 c_noun
                 subfile
                 var
                   nil
                 nil
           nomi
             at
             np_args
               location
               spec
                 a
               ng
                 nil
                 c_noun
                 control_break
                 var
                   cb
                 [
                   pp
                     of
                     np_args
                       nil
                       spec
                         a
                       ng
                         nil
                         c_noun
                         file
                         var
                           'F2
                         nil
                ]
         ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input (question(?) OR statement( ) OR phrase( ))
|  How do I write [a numeric item with leading zeroes]?

* Lexical Phase *
CI   [how,do,i,report,'[',a,numeric item,with leading zeroes ']']
SML  [{write,report}]
IT   question
S1   success
RT1  [2916,283]

* Parsing *
QAS  seqap
QS   []
S2 : success
RT2: [3066,100]
PO
      q
        hdi
        yes
        report
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               item
               spec
                 a
               ng
                [
                  numeric
                ]
                 c_noun
                 item
                 var
                   nil
                [
                  pp
                    with
                    np_args
                      nil
                      spec
                        unspec
                      ng
```

```
                    [
                      leading
                    ]
                    c_noun
                    zeroes
                    var
                      nil
                    nil
                ]
        ]


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|    I1 is a QUIZ item

Lexical analysis stopped  fatal typo in the input
Check quotes, square brackets and invalid characters

* Lexical Phase *
CI    _1757
SML   _94
IT    _1758
S1    error(typo(quotes,brackets,' /?/  ,   ))
RT1   [3383,33]
Parsing stopped  error found during the lexical phase


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )],
|    I1   s a QUIZ item

* Lexical Phase *
CI    [var( I1'),is,a pn_qn( QUIZ'),item]
SML   []
IT    statement
S1    success
RT1   [3550,84]

* Parsing *
QAS   seqap
QS    []
S2    success
RT2   [3716,150]
PO
      s
        is_a
          yes
          np_pp
            identifier
            var
              'I1'
          np_pp
            identified_object
            np_args
              nil
              spec
                a
              ng
                [
                  'QUIZ'
                ]
                c_noun
                item
                var
                  'I1'
                nil


yes
| ?- parse(seqap,false)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  'I2' is a new QUIZ item

* Lexical Phase *
CI    [var('I2'),is,a,new,pn_qn('QUIZ'),item]
SML   []
IT    statement
S1    success
RT1   [4016,133]

* Parsing *
QAS   seqap
QS    []
S2    success
RT2   [4233,150]
PO
      s
        is_a
          yes
          np_pp
            identifier
            var
              'I2
          np_pp
            identified_object
            np_args
              nil
              spec
                a
              ng
                [
                  new
                  'QUIZ'
                ]
                c_noun
                item
                var
                   I2
                nil


yes
| ?- parse(seqap,false)

SEQAP PARSER;
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How do I report 'I1' and  I2' on a line ?

* Lexical Phase *
CI    [how,do,i,report,var('I1 ),and,var('I2'),on,a,line]
SML   []
IT    question
S1    success
RT1   [4583,200]
```

F.3 1

```
* Parsing *
QAS: seqap
QS : []
S2 : success
RT2  [4850,184]
PO :
      q
        hdi
        yes
        report
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             conj
               and
               [
                 np_args
                   item
                   spec
                     a
                   ng
                    [
                      'QUIZ'
                    ]
                     c_noun
                     item
                     var
                       'I1'
                     nil
                 np_args
                   item
                   spec
                     a
                   ng
                    [
                      new
                      'QUIZ'
                    ]
                     c_noun
                     item
                     var
                       'I2'
                     nil
               ]
           nomi
             on
             np_args
               location
               spec
                 a
               ng
                 nil
                 c_noun
                 line
                 var
                   nil
                 nil
        ]
```

```
yes
| ?- parse(seqap,true).

SEQAP PARSER;
Enter your input (question(?) OR statement( ) OR phrase(.))
|: 'X' is a subtotal of an item

Incomplete end of input  select one of
1: end of statement ( ) OR
2: end of question  (?) OR
3: end of phrase    (;)
Please, enter 1. or 2  or 3
| 1.

* Lexical Phase *
CI   [var('X'),is,a,subtotal,of,an,item]
SML: []
IT   statement
S1   success
RT1  [5433,183]

* Parsing *
QAS: seqap
QS   []
S2   success
RT2  [5666,183]
PQ .
       s
         is_a
           yes
           np_pp
             identifier
             var
               'X'
           np_pp
             identified_object
             np_args
               nil
               spec
                 a
               ng
                 nil
                 c_noun
                 subtotal
                 var
                   'X'
                [
                  pp
                    of
                    np_args
                      nil
                      spec
                        an
                      ng
                        nil
                        c_noun
                        item
                        var
                          nil
                        nil
                ]
```

```
yes
| ?- parse(seqap,false)

SEQAP PARSER;
```

```
Enter your input (question(?) OR statement( ) OR phrase( ))
|. How do I repot 'X' to a subfile at a control break ?

Lexical analysis stopped  the word |repot| is not in the lexicon
Select one of the following options
1  Stop right here  OR
2  Skip the missing word and continue  OR
3  Replace the missing word
Please, enter 1  or 2  or 3
| 3
Enter you replacement
| report
Do you want to replace |repot| by |report| ? (y  or n )
| y

* Lexical Phase *
CI   [how,do,i,report,var('X'),to,a,subfile,at,a,control_break]
SML  []
IT   question
S1   success
RT1  [6333,333]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [6550,134]
PO
      q
        hdi
        yes
        report
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               item
               spec
                 a
               ng
                 nil
                 c_noun
                 subtotal
                 var
                   'X'
                 [
                   pp
                     of
                     np_args
                       nil
                       spec
                         an
                       ng
                         nil
                         c_noun
                         item
                         var
                           nil
                         nil
                 ]
           nomi
             at
             np_args
               location
               spec
                 a
               ng
                 nil
                 c_noun
                 control_break
                 var
                   nil
                 nil
           nomi
             to
             np_args
               destination_file
               spec
                 a
               ng
                 nil
                 c_noun
                 subfile
                 var
                   nil
                 nil
         ]


yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input (question(?) OR statement( ) OR phrase( ))
|  Is there an option to the ACCESS statement ?

* Lexical Phase *
CI  . [is,there,an,option,to,the,pn_qn( ACCESS'),statement]
SML  []
IT   question
S1   success
RT1  [7216,183]

* Parsing *
QAS  seqap
QS   _2281
S2   error(warning__no_roles_for_this_verb(be),[])
RT2  [7333,67]
PO
      q
        ?
        yes
        be
        nil
        no_roles
         [
           np_pp
             sub
             np_args
               nil
               spec
                 an
               ng
                 nil
                 c_noun
                 option
                 var
                   nil
                 [
```

F.5

```
                pp
                  to
                  np_args
                    nil
                    spec
                      the
                    ng
                     [
                       'ACCESS'
                     ]
                     c_noun
                     statement
                     var
                       nil
                     nil
              ]
            nil
          ]
          nil

yes
| ?- parse(seqap,true)

SEQAP PARSER.
Enter your input [question(?) OR statement( ) OR phrase(  ]
|  I have a report item 'x'

* Lexical Phase *
CI    [i,have,a,report_item,var(x)]
SML   []
TT    statement
S1    success
RT1.  [7750,117]

* Parsing *
QAS   seqap
QS    []
S2    success
RT2   [7933,117]
PO
       s
         have
           yes
           np_pp
             sub
             np_args
               nil
               spec
                 nil
               ng
                 nil
                 pers_pro
                 i
                 var
                   nil
                 nil
           nil
           np_pp
             d_o
             np_args
               nil
               spec
                 a
               ng
                 nil
                 c_noun
                 report_item
                 var
                   x
                 nil
                  .

yes
| ?- parse(seqap,false)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  the type of 'x' is numeric

* Lexical Phase *
CI    [the,type,of,var(x),is,numeric]
SML   []
IT    statement
S1    success
RT1   [8283,133]

* Parsing *
QAS   seqap
QS    []
S2    success
RT2   [8583,200]
PO
       s
         is_adj
           yes
           np_pp
             sub
             np_args
               nil
               spec
                 the
               ng
                 nil
                 c_noun
                 type
                 var
                   nil
                 [
                   pp
                     of
                     np_args
                       nil
                       spec
                         a
                       ng
                         nil
                         c_noun
                         report_item
                         var
                           x
                         nil
                 ]
           np_pp
             attribute
             [
               numeric
             ]

yes
| ?- parse(seqap,true)
```

Fig. 6 - 1

```
SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  'x222' is an item

* Lexical Phase *
CI   [var(x222),is,an,item]
SML  []
IT   statement
S1   success
RT1  [9016,50]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [9216,166]
PO
      s
        is_a
          yes
          np_pp
            identifier
            var
              x222
          np_pp
            identified_object
            np_args
              nil
              spec
                an
              ng
                nil
                c_noun
                item
                var
                  x222
                nil


yes
| ?- parse(seqap,false)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  Why is 'x222' an accessible record item ?

* Lexical Phase *
CI   [why,is,var(x222),an,accessible,record_item]
SML: []
IT   question
S1   success
RT1  [9633,150]

* Parsing *
QAS  seqap
QS   _2087
S2   error(warning__no_roles_for_this_verb(be),[])
RT2  [9750,84]
PO
      q
        why
        yes
        be
        nil
        no_roles
          [
            np_pp
              sub
              np_args
                nil
                spec
                  an
                ng
                  nil
                  c_noun
                  item
                  var
                    xZ22
                  nil
            np_pp
              d_o
              np_args
                nil
                spec
                  an
                ng
                  [
                    accessible
                  ]
                  c_noun
                  record_item
                  var
                    nil
                  nil
          ]
          nil


yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How does I report an item to a file and a subfile ?

* Lexical Phase *
CI   [how,does,i,report,an,item,to,a,file,and,a,subfile]
SML  []
IT   question
S1   success
RT1  [10333,300]

* Parsing *
QAS  seqap
QS   _2922
S2   error(illegal_how_question,
           [does,i,report,an,item,to,a,file,and,a,subfile])
RT2  [10716,283]
PO
      q
        hdi
        yes
        do
        nil
        no_roles
          [
            nil
            nil
          ]
          nil


yes
```

```
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase(.)]
|  How do I report an item to a file and a subfile ?

* Lexical Phase *
CI   [how,do,i,report,an,item,to,a,file,and,a,subfile]
SML  []
IT   question
S1   success
RT1  [11183,317]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [11350,117]
PO
      q
        hdi
        yes
        report
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               item
               spec
                 an
               ng
                 nil
                 c_noun
                 item
                 var
                   nil
                 nil
           nomi
             to
             conj
               and
              (
                np_args
                  destination_file
                  spec
                    a
                  ng
                    nil
                    c_noun
                    file
                    var
                      nil
                    nil
                np_args
                  destination_file
                  spec
                    a
                  ng
                    nil
                    c_noun
                    subfile
                    var
                      nil
                    nil
                ]
         ]


yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How do I report an item to a file and to a subfile?

* Lexical Phase *
CI   [how,do,i,report,an,item,to,a,file,and,to,a,subfile]
SML  []
IT   question
S1   success
RT1  [12016,350]

* Parsing *
QAS  seqap
QS   []
S2.  success
RT2  [12250,184]
PO
      q
        hdi
        yes
        report
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               item
               spec
                 an
               ng
                 nil
                 c_noun
                 item
                 var
                   nil
                 nil
           nomi
             to
             conj
               and
              (
                np_args
                  destination_file
                  spec
                    a
                  ng
                    nil
                    c_noun
                    file
                    var
                      nil
                    nil
                np_args
```

```
                destination_file
                spec
                  a
                ng
                  nil
                  c_noun
                  subfile
                  var
                    nil
                  nil
              ]
          ]


yes
| ?- parse(seqap,true)

SEQAP PARSER;
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How to report an average at a control break to a permanent QUIZ subfile

* Lexical Phase *
CI    [how,to,report,an,average,at,a,control_break,to a,permanent,pn_qn('QUIZ'),subfil
SML   []
IT    question
S1    success
RT1   [13133,367]

* Parsing *
QAS   seqap
QS    []
S2    success
RT2.  [13366,150]
PO
      q
        hdi
        yes
        report
        nil
        roles
         [
           nomi
             sub
             nil
           nomi
             d_o
             np_args
               item
               spec
                 an
               ng
                 nil
                 c_noun
                 average
                 var
                   nil
                 nil
           nomi
             to
             np_args
               destination_file
               spec
                 a
               ng
                [
                  permanent
                  'QUIZ'
                ]
                c_noun
                subfile
                var
                  nil
                nil
          nomi
            at
            np_args
              location
              spec
                a
              ng
                nil
                c_noun
                control_break
                var
                  nil
                nil
      ]


yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How does QUIZ store a defined numeric item ?

* Lexical Phase *
CI    [how,does,pn_qn('QUIZ'),store,a,defined,numeric,item]
SML   []
IT .  question
S1    success
RT1   [13866,200]

* Parsing *
QAS   seqap
QS    []
S2    success
RT2   [14033,150]
PO .
      q
        ,
        yes
        store
        nil
        roles
         [
           nomi
             sub
             np_args
               nil
               spec
                 unspec
               ng
                 nil
                 proper_quiz_noun
                 'QUIZ'
                 var
                   nil
                 nil
           nomi
             d_o
             np_args
               object
               spec
```

Fig. 9

```
              a
            ng
             [
               defined
               numeric
             ]
              c_noun
              item
              var
                nil
              nil
        ]


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How can I execute the REPORT statement ?

* Lexical Phase *
CI   [how,can,i,execute,the,pn_qn('REPORT'),statement]
SML  []
IT   question
S1   success
RT1  [14466,150]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [14650,100]
PO
      q
        hci
        yes
        execute
        nil
        roles
         [
           nom1
             sub
             pers_pro
               I
           nom1
             d_o
             np_args
               object
               spec
                 the
               ng
                [
                  'REPORT
                ]
                 c_noun
                 statement
                 var
                   nil
                 nil
        ]


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase(,)]
|  Why did I get only [a small number of report complexes?

Lexical analysis stopped  fatal typo in the input
Check quotes, square brackets and invalid characters

* Lexical Phase *
CI   _1757
SML  _94
IT   _1758
S1   error(typo(quotes,brackets ' /?/, ,   ))
RT1  [14950,84]
Parsing stopped  error found during the lexical phase


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  Why did I get only [a small number of report complexes]?

* Lexical Phase *
CI   [why,did,i,get,only, [ ,a,small,number,of,report_complexes,']',
SML  []
IT   question
S1   success
RT1  [15350,300]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [15550,134]
PO
      q
        why
        yes
        get
       [
         only
       ]
        roles
         [
           nom1
             sub
             pers_pro
               I
           nom1
             d_o
             np_args
               object
               spec
                 a
               ng
                [
                  small
                ]
                 c_noun
                 number
                 var
                   nil
                 [
                   pp
                     of
                     np_args
                       nil
```

```
                    spec
                      unspec
                    ng
                      nil
                      c_noun
                      report_complexes
                      var
                        nil
                      nil
                ]
        ]


yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  Why am I getting an error message ?

* Lexical Phase *
CI   [why,am,i,getting,an,error_message]
SML  []
IT   question
S1   success
RT1  [16016,183]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [16433,350]
PO
      q
        err
        yes
        get
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               object
               spec
                 an
               ng
                 nil
                 c_noun
                 error_message
                 var
                   nil
                 nil
         ]


yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  Why am I not getting an error message ?

* Lexical Phase *
CI : [why,am,i,not,getting,an,error_message]
SML: []
IT . question
S1 . success
RT1  [16833,183]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [17233,350]
PO
      q
        err
        no
        get
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               object
               spec
                 an
               ng
                 nil
                 c_noun
                 error_message
                 var
                   nil
                 nil
         ]


yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase(.)]
|  Why did I get the error message "Expected  file key"?

* Lexical Phase *
CI   [why,did,i,get,the,error_message,quiz_stuff('Expected  file key')]
SML  []
IT   question
S1   success
RT1  [17633,200]

* Parsing *
QAS  seqap
QS   'Expected. file key'
S2   success
RT2  [17833,100]
PO
      q
        err
        yes
        get
        nil
        roles
```

```
        [
          nomi
            sub.
            pers_pro
              i
          nomi
            d_o
            np_args
              object
              spec
                the
              ng
                nil
                c_noun
                error_message
                var
                  nil
                nil
        ]
```

```
yes
| ?- parse(seqap,true).

SEQAP PARSER;
Enter your input [question(?) OR statement( ) OR phrase(.)].
|  Why does QUIZ count one item only ?

* Lexical Phase *
CI : [why,does,pn_qn('QUIZ'),count,one,item,only]
SML: []
IT : question
S1 : success
RT1: [18250,200]

* Parsing *
QAS: seqap
QS : []
S2 : success
RT2: [18516,183]
PO :
      q
        why
        yes
        count
        [
          only
        ]
        roles
          [
            nomi
              sub
              np_args
                nil
                spec
                  unspec
                ng
                  nil
                  proper_quiz_noun
                  'QUIZ'
                  var
                    nil
                  nil
            nomi
              d_o
              np_args
                item
                spec
                  one
                ng
                  nil
                  c_noun
                  item
                  var
                    nil
                  nil
          ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER;
Enter your input [question(?) OR statement( ) OR phrase(.)]
|: What happens if I use 'ACCESS BRANCHES  REPORT NAME OF BRANCH'

* Lexical Phase *
CI : [what,happens,if,i,use,
      quiz_stuff('ACCESS BRANCHES, REPORT NAME OF BRANCH')]
SML: []
IT : question
S1 : success
RT1: [18933,183]

* Parsing *
QAS: seqap
QS : 'ACCESS BRANCHES; REPORT NAME OF BRANCH'
S2 : success
RT2: [19083,83]
PO :
      q
        hyp
        yes
        use
        nil
        roles
          [
            nomi
              sub
              pers_pro
                i
            nomi
              d_o
              np_args
                object
                spec
                  nil
                ng
                  nil
                  quiz_code_errmessage
                  nil
                  var
                    nil
                  nil
          ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER;
Enter your input [question(?) OR statement( ) OR phrase(.)]
|: What do I do if I get only one record complex ?
```

```
* Lexical Phase *
CI   [what,do,i,do,if,i,get,only,one,record_complex]
SML  []
IT   question
S1   success
RT1  [19516,283]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [19716 150]
PO
      q
        ?
        yes
        get
        [
          only
        ]
        roles
          [
            nomi
              sub
              np_args
                nil
                spec
                  nil
                ng
                  nil
                  pers_pro
                  i
                  var
                    nil
                  nil
            nomi
              d_o
              np_args
                object
                spec
                  one
                ng
                  nil
                  c_noun
                  record_complex
                  var
                    nil
                  nil
          ]


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input (question(?) OR statement( ) OR phrase( ))
|  Can I display my report groups on the screen ?

* Lexical Phase *
CI   [can,i,display,my report_groups,on the,screen]
SML  []
IT   question
S1   success
RT1  [20250,217]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [20433,117]
PO
      q
        ?
        yes
        display
        nil
        roles
          [
            nomi
              sub
              pers_pro
                i
              nomi
                d_o
                np_args
                  object
                  spec
                    my
                  ng
                    nil
                    c_noun
                    report_groups
                    var
                      nil
                    nil
              nomi
                on
                np_args
                  device
                  spec
                    the
                  ng
                    nil
                    c_noun
                    screen
                    var
                      nil
                    nil


yes
| ?- parse(seqap true)

SEQAP PARSER
Enter your input (question(?) OR statement( ) OR phrase( ))
|  Does QUIZ compare numeric items with defined items ?

* Lexical Phase *
CI   [does,pn_qn('QUIZ ) compare numeric items with defined items]
SML  []
IT   question
S1   success
RT1  [21066,233]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [21333,167]
PO
      q
        ?
```

```
yes
compare
nil
roles
 [
   nomi
     sub
     np_args
       nil
       spec
         unspec
       ng
         nil
         proper_quiz_noun
         'QUIZ'
         var
           nil
         nil
   nomi
     d_o
     np_args
       item1
       spec
         unspec
       ng
        [
          numeric
        ]
         c_noun
         items
         var
           nil
         nil
   nomi
     with
     np_args
       item2
       spec
         unspec
       ng
        [
          defined
        ]
         c_noun
         items
         var
           nil
         nil
 ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How do I report an items ?

* Lexical Phase *
CI    [how,do,i,report,an,items]
SML   []
IT    question
S1    success
RT1   [21833,150]

* Parsing *
QAS   seqap
```

```
QS  _1716
S2   error(invalid_noun_phrase,[an,items])
RT2  [21950,100]
PO
      q
        hdi
        yes
        report
        nil
        no_roles
         [
           np_pp
             sub
             pers_pro
               i
           np_pp
             d_o
             nil
         ]
          nil
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How does you report an item?

* Lexical Phase *
CI    [how,does,you,report,an,item]
SML   []
IT    question
S1    success
RT1.  [22683,167]

* Parsing *
QAS   seqap
QS    _1643
S2    error(illegal_how_question,[does,you,report,an,item])
RT2   [22950,217]
PO
       q
         hdi
         yes
         do
         nil
         no_roles
          [
            nil
            nil
          ]
           nil
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER;
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How do I report one or more items ?

* Lexical Phase *
CI    [how,do,i,report,one,or,more,items]
SML.  []
IT    question
```

```
S1 : success
RT1: [23266,183]

* Parsing *
QAS: seqap
QS : []
S2   success
RT2: [23450,100]
PO
      q
        hdi
        yes
        report
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               item
               spec
                 one_or_more
               ng
                 nil
                 c_noun
                 items
                 var
                   nil
                 nil
         ]


yes
| ?- parse(seqap,true).

SEQAP PARSER;
Enter your input [question(?) OR statement(.) OR phrase(;)]
|: How do I report 2-100 items ?

* Lexical Phase *
CI   [how,do,i,report,'2-100',items]
SML: []
IT : question
S1 : success
RT1: [23933,167]

* Parsing *
QAS: seqap
QS : []
S2 : success
RT2: [24083,67]
PO
      q
        hdi
        yes
        report
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               item
               spec
                 range
                   2
                   100
               ng
                 nil
                 c_noun
                 items
                 var
                   nil
                 nil
         ]


yes
| ?- parse(seqap,true)

SEQAP PARSER;
Enter your input [question(?) OR statement(.) OR phrase(;)]:
|: How do I print on the printer ?

* Lexical Phase *
CI   [how,do,i,print,on,the,printer]
SML: []
IT   question
S1   success
RT1: [24566,150]

* Parsing *
QAS: seqap
QS : _2243
S2 : error(missing_mandatory_role(d_o(object)),[])
RT2  [24716,83]
PO
      q
        hdi
        yes
        print
        nil
        no_roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               d_o
               nil
           nomi
             on
             np_args
               device
               spec
                 the
               ng
                 nil
                 c_noun
                 printer
                 var
                   nil
                 nil
```

```
        ]


yes
| ?- parse(seqap,true)

SEQAP PARSER;
Enter your input (question(?) OR statement( ) OR phrase( ))
|  How do I print report groups on the printer ?

* lexical Phase *
CI  [how,do,i,print,report_groups,on,the,printer]
SML  []
IT   question
S1   success
RT1  [25183,250]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [25383,133]
PO
      q
        hdi
        yes
        print
        nil
        roles
         [
           nomi
             sub
             pers_pro
               1
           nomi
             d_o
             np_args
               object
               spec
                 unspec
               ng
                 nil
                 c_noun
                 report_groups
                 var
                   nil
                 nil
           nomi
             on
             np_args
               device
               spec
                 the
               ng
                 nil
                 c_noun
                 printer
                 var
                   nil
                 nil
         ]


yes
| ?- parse(seqap,true)
```

```
SEQAP PARSER,
Enter your input (question(?) OR statement( ) OR phrase(.))
|  How do I print a printer ?

* Lexical Phase *
CI  [how,do,i,print,a,printer]
SML  []
IT   question
S1   success
RT1  [25950,134]

* Parsing *
QAS: seqap
QS   _1911
S2   error(semantic_invalid noun(printer),[])
RT2  [26033,33]
PO
      q
        hdi
        yes
        print
        nil
        no_roles
         [
           nomi
             sub
             pers_pro
               1
           nomi
             d_o
             np_args
               nil
               spec
                 a
               ng
                 nil
                 c_noun
                 printer
                 var
                   nil
                 nil
         ]


yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input (question(?) OR statement( ) OR phrase( ))
|  How do I print an item from the printer ?

* Lexical Phase *
CI  [how,do,i,print,an,item,from,the,printer]
SML: []
IT   question
S1   success
RT1  [26566,233]

* Parsing *
QAS  seqap
QS   _2779
S2   error(illegal_preposition(from),[])
RT2  [26733,117]
PO
      q
        hdi
        yes
```

```
print
nil
no_roles
 [
   nomi
     sub
     pers_pro
       i
   nomi
     d_o
     np_args
       object
       spec
         an
       ng
         nil
         c_noun
         item
         var
           nil
         nil
   nomi
     from
     np_args
       nil
       spec
         the
       ng
         nil
         c_noun
         printer
         var
           nil
         nil
 ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input (question(?) OR statement( ) OR phrase( ))
|  How do I print an item on a new page ?

* Lexical Phase *
CI   (how,do,i,print,an,item,on,a,new,page)
SML  ()
IT   question
S1   success
RT1  (27250,234)

* Parsing *
QAS. seqap
QS   ()
S2   success
RT2  (27416,100)
PO
```

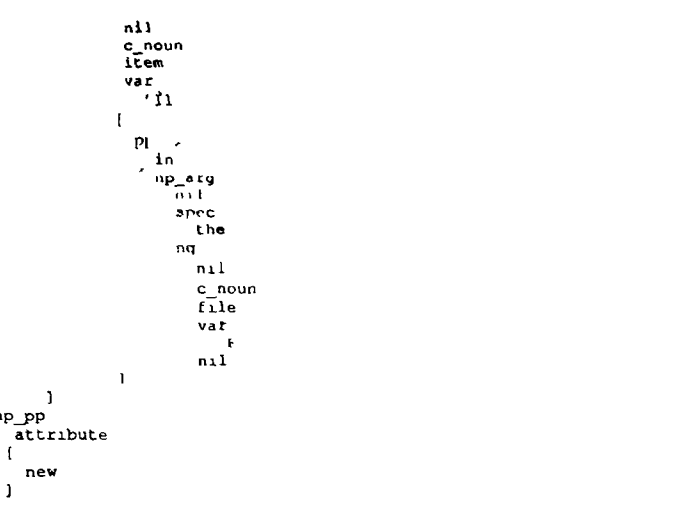
```
 q
   hdi
   yes
   print
   nil
   roles
    [
      nomi
        sub
        pers_pro
          i
      nomi
        d_o
        np_args
          object
          spec
            an
          ng
            nil
            c_noun
            item
            var
              nil
            nil
      nomi
        on
        np_args
          location
          spec
            a
          ng
           [
             new
           ]
            c_noun
            page
            var
              nil
            nil
 ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input (question(?) OR statement( ) OR phrase( ))
|  the file 'FS%^' is primary

Lexical analysis stopped  fatal typo in the input
Check quotes, square brackets and invalid characters

* Lexical Phase *
CI   _1757
SML. _94
IT   _1758
S1   error(typo(quotes,brackets,' /?/ ',   ))
RT1  (27666,33)
Parsing stopped  error found during the lexical phase


yes
| ?- parse(seqap,true)

SEQAP PARSER;
Enter your input (question(?) OR statement( ) OR phrase(:))
|: the.file 'F' is primary.

* Lexical Phase *
CI : (the,file,var('F'),is,primary)
SML  ()
IT   statement
S1   success
RT1. (27866,100)

* Parsing *
QAS: seqap
```

```
QS : []
S2 : success
RT2: [28000,67]
PO :
      s
        is_adj
          yes
          np_pp
            sub
            np_args
              nil
              spec
                the
              ng
                nil
                c_noun
                file
                var
                  'F'
                nil
          np_pp
            attribute
            [
              primary
            ]
```

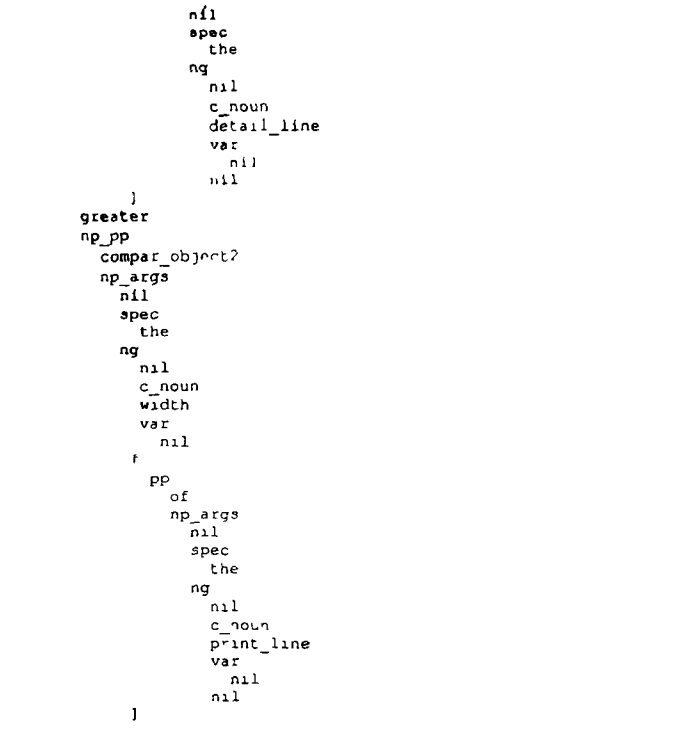
```
yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  the value of the item 'XXX234' is 3509 99

* Lexical Phase *
CI   [the,value,of,the,item,var( XXX234'),is,3509 99]
SML: []
IT   statement
S1   success
RT1: [28350,134]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [28483,100]
PO :
      s
        assign
          yes
          np_pp
            object
            np_args
              nil
              spec
                the
              ng
                nil
                c_noun
                value
                var
                  nil
                nil
          np_pp
            object_modifier
            np_args
              nil
              spec
                the
              ng
                nil
                c_noun
                item
                var
                  'XXX234'
                nil
          np_pp
            value
            np_args
              nil
              spec
                nil
              ng
                nil
                number
                3509 99
                var
                  nil
                nil
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  the name of [the item 'I1' in [the file  f ]] is new

* Lexical Phase *
CI   [the,name,of,'[',the,item,var('I1 '),in, [ ,the,file,var('F'),']',']',is,new]
SML  []
IT   statement
S1   success
RT1  [29016,283]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [30083,967]
PO
      s
        is_adj
          yes
          np_pp
            sub
            np_args
              nil
              spec
                the
              ng
                nil
                c_noun
                name
                var
                  nil
                [
                  pp
                    of
                    np_args
                      nil
                      spec
                        the
                      ng
```



```
                    nil
                    c_noun
                    item
                    var
                      'Il
                    [
                      pl
                        in
                        np_arg
                          nil
                          spec
                            the
                          ng
                            nil
                            c_noun
                            file
                            var
                              f
                            nil
                    ]
              ]
          np_pp
            attribute
            [
              new
            ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )].
| the width of the detail line is greater than the width of the print line

* Lexical Phase *
CI   [the,width,of,the,detail_line,is,greater,than,the width,of,the,print_line]
SML  []
IT   statement
S1   success
RT1  [30833,350]

* Parsing *
QAS  seqap
QS   []
S2 · success
RT2  [31266,350]
PO
      s
        comparison
          yes
          np_pp
            compar_object1
            np_args
              nil
              spec
                the
              ng
                nil
                c_noun
                width
                var
                  nil
                [
                  pp
                    of
                    np_args
```



```
                    nil
                    spec
                      the
                    ng
                      nil
                      c_noun
                      detail_line
                      var
                        nil
                      nil
              ]
          greater
          np_pp
            compar_object2
            np_args
              nil
              spec
                the
              ng
                nil
                c_noun
                width
                var
                  nil
                f
                  pp
                    of
                    np_args
                      nil
                      spec
                        the
                      ng
                        nil
                        c_noun
                        print_line
                        var
                          nil
                        nil
          ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
| there is no selection condition on records of SFILE

* Lexical Phase *
CI · [there,is,no,selection_condition,on records,of,qaur_var(' SFILE')]
SML  []
IT   statement
S1 . success
RT1  [32033,217]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [32233,83]
PO
      s
        there_is
          no
          nil
          np_pp
            object
```

```
        np_args
          nil
          spec
            unspec
          ng
            nil
            [illegible]
            selection_condition
            var
              nil
            [
              pp
                of
                np_args
                  nil
                  spec
                    nil
                  ng
                    [illegible]
                    qauz_var
                    nil
                    var
                      ' SFILE
                    nil
              pp
                on
                np_args
                  nil
                  spec
                    unspec
                  ng
                    nil
                    c_noun
                    records
                    var
                      nil
                    nil
        ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  a link for  FILE is established

* Lexical Phase *
CI   [a,link,for,qauz_var(  FILE ),is,established]
SML  []
IT   statement
S1   success
RT1  [32683 150]

* Parsing *
QAS: seqap
QS : _1668
S2   error(warning__no_roles_for_this_verb(establish),[])
RT2  [32866,133]
PO
      s
        passive
          yes
          establish
          nil
          no_roles
            [
```

```
            np_pp
              sub
              nil
            np_pp
              d_o
              np_args
                nil
                spec
                  a
                ng
                  nil
                  c_noun
                  link
                  var
                    nil

                  pp
                    for
                    np_args
                      nil
                      spec
                        nil
                      ng
                        nil
                        qauz_var
                        nil
                        var
                            FILE
                        nil
          ]
        ]
        nil
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  I am printing an item 'I999'

* Lexical Phase *
CI   [i,am,printing,an,item,var('I999 )]
SML  []
IT   statement
S1   success
RT1  [33383,133]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [33566,133]
PO
      s
        regular
          yes
          print
          nil
          roles
            [
              nomi
                sub
                np_args
                  nil
                  spec
                    nil
```

```
            ng
              nil
              pers_pro
              i
              var
                nil
              nil
          nomi
            d_o
            np_args
              object
              spec
                an
              ng
                nil
                c_noun
                item
                var
                  '1999'
                nil
        ]
```

```
yes
| ?- parse(seqap,true).

SEQAP PARSER
Enter your input (question(?) OR statement( ) OR phrase( ))
|  skipping a line after each report group

* Lexical Phase *
CI    [skipping,a,line,after,each,report_group]
SML   []
IT    phrase
S1    success
RT1   [33916,183]

* Parsing *
QAS   seqap
QS    []
S2    success
RT2   [34033,83]
PO
      F
        gerund_np
          yes
          skip
          nil
          roles
           [
             nomi
               sub
               nil
             nomi
               d_o
               np_args
                 object
                 spec
                   a
                 ng
                   nil
                   c_noun
                   line
                   var
                     nil
                   nil
             nomi
               after
               np_args
                 relative_location
                 spec
                   each
                 ng
                   nil
                   c_noun
                   report_group
                   var
                     nil
                   nil
           ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input (question(?) OR statement( ) OR phrase( ))
|  accessing a file with the ACCESS statement

* Lexical Phase *
CI    [accessing,a,file,with,the,pn_qn('ACCESS'),statement]
SML   []
IT    phrase
S1    success
RT1   [34466,183]

* Parsing *
QAS   seqap
QS    []
S2    success
RT2   [34616,100]
PO
      F
        gerund_np
          yes
          access
          nil
          roles
           [
             nomi
               sub
               nil
             nomi
               d_o
               np_args
                 source_file
                 spec
                   a
                 ng
                   nil
                   c_noun
                   file
                   var
                     nil
                   nil
             nomi
               with
               np_args
                 instrument
                 spec
                   the
                 ng
                  [
                    'ACCESS'
```

```
            ]
             c_noun
             statement
             var
               nil
             nil
          ]


yes
| ?- parse(seqap,true)

SEQAP PARSER;
Enter your input [question(?) OR statement( ) OR phrase(,)]
|: adding the LINK option to the ACCESS statement;

* Lexical Phase *
CI : [adding,the,pn_qn('LINK'),option,to,the,pn_qn('ACCESS'),statement]
SML: []
IT : phrase
S1 : success
RT1: [34933,183]

* Parsing *
QAS: seqap
QS : []
S2 : success
RT2: [35116,100]
PO :
      p
        gerund_np
          yes
          add
          nil
          roles
           [
             nom1
               sub
               nil
             nom1
               d_o
               np_args
                 added_object
                 spec
                   the
                 ng
                  [
                    'LINK'
                  ]
                   c_noun
                   option
                   var
                     nil
                   nil
          - nom1
             to
             np_args
               recipient_object
               spec
                 the
               ng
                [
                  'ACCESS'
                ]
                 c_noun
                 statement
                 var
                   nil
                 nil
          ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER;
Enter your input [question(?) OR statement( ) OR phrase(,)]
|: adding numbers after a control break

* Lexical Phase *
CI   [adding,numbers,after,a,control_break]
SML: []
IT : phrase
S1   success
RT1  [35633,183]

* Parsing *
QAS: seqap
QS : []
S2 : success
RT2: [35766,83]
PO :
      p
        gerund_np
          yes
          add
          nil
          roles
           [
             nom1
               sub
               nil
             nom1
               d_o
               np_args
                 argument1
                 spec
                   unspec
                 ng
                   nil
                   c_noun
                   numbers
                   var
                     nil
                   nil
             nom1
               after
               np_args
                 location
                 spec
                   a
                 ng
                   nil
                   c_noun
                   control_break
                   var
                     nil
                   nil
           ]


yes
| ?- parse(seqap,true).
```

```
SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase(?)]
|: to subtotal an item at a control breaks

* Lexical Phase *
CI   [to,subtotal,an,item,at,a control_breaks]
SML  []
IT   phrase
S1 : success
RT1  [36216,183]

* Parsing *
QAS  seqap
QS   _2365
S2   error(invalid_noun_phrase,[a,control_breaks])
RT2  [36366,100]
PO
      p
        vp
          yes
          subtotal
          nil
          no_roles
           [
             np_pp
               sub
               nil
             np_pp
               d_o
               np_args
                 nil
                 spec
                   an
                 ng
                   nil
                   c_noun
                   item
                   var
                     nil
                   nil
           ]
           [
             pp
               at
               nil
           ]


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
[  to subtotal an item at a control break

* Lexical Phase *
CI   [to,subtotal,an,item at,a,control_break]
SML  []
IT   phrase
S1   success
RT1  [36816,200]

* Parsing *
QAS  seqap
QS   []
S2   success
```

```
RT2: [36950,100]
PO :
      p
        vp
          yes
          subtotal
          nil
          roles
           [
             nomi
               sub
               nil
             nomi
               d_o
               np_args
                 item
                 spec
                   an
                 ng
                   nil
                   c_noun
                   item
                   var
                     nil
                   nil
             nomi
               at
               np_args
                 location
                 spec
                   a
                 ng
                   nil
                   c_noun
                   control_break
                   var
                     nil
                   nil
           ]


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  to define an item at a

* Lexical Phase *
CI   [to,define,an,item at a]
SML  []
IT   phrase
S1   success
RT1  [37366,133]

* Parsing *
QAS  seqap
QS   _1695
S2   error(invalid_noun_phrase,[a])
RT2  [37483,50]
PO
      p
        vp
          yes
          define
          nil
          no_roles
```

```
[
  np_pp
    sub
    nil
  np_pp
    d_o
    np_args
      nil
      spec
        an
      ng
        nil
        c_noun
        item
        var
          nil
        nil
]
[
  pp
    at
    nil
]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  to direct a report to the printttttttttttttter

Lexical analysis stopped  the word |printttttttttttttter| is not in the lexicon
Select one of the following options
1  Stop right here  OR
2  Skip the missing word and continue  OR
3  Replace the missing word
Please, enter 1  or 2  or 3
|  1
If necessary have the missing word added to the lexicon

* Lexical Phase *
CI   _3431
SML  _1146
IT   _3432
S1   error(unknown_word)
RT1  [37950,184]
Parsing stopped  error found during the lexical phase


yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  to direct a report to thhhe the printer

Lexical analysis stopped  the word |thhhe| is not in the lexicon
Select one of the following options
1  Stop right here  OR
2  Skip the missing word and continue  OR
3  Replace the missing word
Please, enter 1  or 2  or 3
|  2

* Lexical Phase *
CI   [to,direct,a,report,to,the,printer]
```

```
SML  []
IT   phrase
S1   success
RT1  [38166,166]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [38266,50]
PO
      p
        vp
          yes
          direct
          nil
          roles
           [
             nom1
               sub
               nil
             nom1
               d_o
               np_args
                 report
                 spec
                   a
                 ng
                   nil
                   c_noun
                   report
                   var
                     nil
                   nil
             nom1
               to
               np_args
                 device
                 spec
                   the
                 ng
                   nil
                   c_noun
                   printer
                   var
                     nil
                   nil
           ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  to divide a numeric item by a number
  *
Incomplete end of input  select one of
1  end of statement  ( ) OR
2  end of question   (?) OR
3  end of phrase     ( )
Please, enter 1  or 2  or 3
|  3

* Lexical Phase *
CI   [to,divide,a,numeric,item,by,a,number]
SML  []
```

```
IT : phrase
S1 : success
RT1: [38733,167]

* Parsing *
QAS: seqap
QS   []
S2 : success
RT2: [38866,83]
PO :
      p
        vp
          yes
          divide
          nil
          roles
           [
             nomi
               sub
               nil
             nomi
               d_o
               np_args
                 argument1
                 spec
                   a
                 ng
                  [
                    numeric
                  ]
                   c_noun
                   item
                   var
                     nil
                   nil
             nomi
               by
               np_args
                 argument2
                 spec
                   a
                 ng
                   nil
                   c_noun
                   number
                   var
                     nil
                   nil
           ]
```

```
yes
| ?- parse(seqap,true).

SEQAP PARSER;
Enter your input [question(?) OR statement(.) OR phrase(.)]
|: link a file 'F777' to a file 'F7878';

* Lexical Phase *
CI : [link,a,file,var('F777'),to,a,file,var('F7878')]
SML: []
IT : phrase
S1 : success
RT1: [39466,116]

* Parsing *
QAS: seqap
QS   []
S2 : success
RT2: [39616,100]
PO :
      p
        vp
          yes
          link
          nil
          roles
           [
             nomi
               sub
               nil
             nomi
               d_o
               np_args
                 co_source_file
                 spec
                   a
                 ng
                   nil
                   c_noun
                   file
                   var
                     'F777'
                   nil
             nomi
               to
               np_args
                 co_source_file
                 spec
                   a
                 ng
                   nil
                   c_noun
                   file
                   var
                     'F7878'
                   nil
           ]
```

```
yes
| ?- parse(seqap,true).

SEQAP PARSER;
Enter your input [question(?) OR statement(.) OR phrase(.)]
|: multiply numbers;

* Lexical Phase *
CI : [multiply,numbers]
SML: []
IT : phrase
S1 : success
RT1: [40000,67]

* Parsing *
QAS: seqap
QS : []
S2 : success
RT2: [40083,67]
PO :
      p
        vp
          yes
          multiply
```

```
        nil
        roles
         [
           nomi
             sub
             nil
           nomi
             d_o
             np_args
               argument1
               spec
                 unspec
               ng
                 nil
                 c_noun
                 numbers
                 var
                   nil
                 nil
         ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  reset an item to a number.

* Lexical Phase *
CI   [reset,an,item,to,a,number]
SML  []
IT   phrase
S1 : success
RT1  [40466,150]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2  [40650,117]
PO
      p
        vp
          yes
          reset
          nil
          roles
           [
             nomi
               sub
               nil
             nomi
               d_o
               np_args
                 item
                 spec
                   an
                 ng
                   nil
                   c_noun
                   item
                   var
                     nil
                   nil
             nomi
               to
               np_args
                 value
                 spec
                   a
                 ng
                   nil
                   c_noun
                   number
                   var
                     nil
                   nil
           ]
```

```
yes
| ?- parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
'  set an item to 34567

* Lexical Phase *
CI   [set,an,item,to,34567]
SML  []
IT   phrase
S1   success
RT1  [41033,117]

* Parsing *
QAS  seqap
QS   []
S2   success
RT2. [41233,117]
PO
      p
        vp
          yes
          set
          nil
          roles
           [
             nomi
               sub
               nil
             nomi
               d_o
               np_args
                 item
                 spec
                   an
                 ng
                   nil
                   c_noun
                   item
                   var
                     nil
                   nil
             nomi
               to
               np_args
                 value
                 spec
                   nil
                 ng
                   nil
                   number
                   34567
```

```
                var
                  nil
                nil
          ]


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How do I sort in ascending order ?

* Lexical Phase *
CI   [how,do,i,sort in,ascending,order]
SML  []
IT   question
S1   success
RT1  [41750,167]

* Parsing *
QAS  seqap
QS   _2164
S2   error(missing_mandatory_role(d_o(item)),[])
RT2  [41883,67]
PO
       q
         hdi
         yes
         sort
         nil
         no_roles
          [
            nomi
              sub
              pers_pro
                i
            nomi
              d_o
              np_args
                d_o
                nil
            nomi
              in
              np_args
                order
                spec
                  unspec
                ng
                 [
                   ascending
                 ]
                 c_noun
                 order
                 var
                   nil
                 nil
          ]


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How do I sort items from a QUIZ data file ?

* Lexical Phase *
CI   [how,do,i,sort,items,from,a,pn_qn('QUIZ'),data file]
SML  []
IT   question
S1   success
RT1  [42333,217]

* Parsing *
QAS  seqap
QS   _2863
S2   error(illegal_preposition(from) [])
RT2  [42466,100]
PO
       q
         hdi
         yes
         sort
         nil
         no_roles
          [
            nomi
              sub
              pers_pro
                i
            nomi
              d_o
              np_args
                item
                spec
                  unspec
                ng
                  nil
                  c_noun
                  items
                  var
                    nil
                  nil
            nomi
              from
              np_args
                nil
                spec
                  a
                ng
                 [
                   'QUIZ'
                 ]
                  c_noun
                  data_file
                  var
                    nil
                  nil
          ]


yes
| ?- parse(seqap,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase(,)]
|  How do I sort a file in ascending order ?

* Lexical Phase *
CI   [how,do,i,sort,a,file,in,ascending,order]
SML  []
IT   question
```

```
S1 : success
RT1  [43133,217]

* Parsing *
QAS  seqap
QS   _2735
S2 : error(semantic_invalid_noun(file),[])
RT2  [43316,150]
PO
      q
        hdi
        yes
        sort
        nil
        no_roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               nil
               spec
                 a
               ng
                 nil
                 c_noun
                 file
                 var
                   nil
                 nil
         nomi
           in
           np_args
             order
             spec
               unspec
             ng
              [
                ascending
              ]
              c_noun
              order
              var
                nil
              nil
       ]


    parse(seqap,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase(,)]
|: How do I sort my items in ascending order ?

* Lexical Phase *
CI  : [how,do,i,sort,my,items,in,ascending,order]
SML : []
IT   question
S1 : success
RT1  [43733,217]

* Parsing *
QAS  seqap
```

```
QS  : []
S2  : success
RT2  [43916,150]
PO
      q
        hdi
        yes
        sort
        nil
        roles
         [
           nomi
             sub
             pers_pro
               i
           nomi
             d_o
             np_args
               item
               spec
                 my
               ng
                 nil
                 c_noun
                 items
                 var
                   nil
                 nil
         nomi
           in
           np_args
             order
             spec
               unspec
             ng
              [
                ascending
              ]
              c_noun
              order
              var
                nil
              nil
       ]


yes
| ?- ^D
[ End of Prolog execution ]
uotcs1b# ^D
script done on Mon Nov 24 20 49.46 1986
```

APPENDIX G : EXAMPLES OF POST-PROCESSING FORMATTING

Appendix G: Examples of Post-Processing Formatting

This appendix presents simple examples of the execution of SEQAP where the default output format, as shown in appendix F, is transformed into a format more appropriate for HDI and QAUZ subsystems (see 3.2.4). As mentioned in chapter 3, the semantic verification is not performed when such a transformation is done.

For the description of the elements of SEQAP's output, see page F.0 (in appendix F).

```
Script started on Mon Nov 24 15 51 26 1986
uotèsib# seqap-t

yes
| ?- parse(hdi,true)

SEQAP PARSER,
Enter your input [question(?) OR statement( ) OR phrase( )]
|  'F' is a new QUIZ file

* Lexical Phase *
CI    [var('F'),is a,new,pn_qn('QUIZ'),file]
SML   []
IT    statement
S1    success
RT1   [133,117]

* Parsing *
QAS   hdi
QS    []
S2    success
RT2   [383,200]
PO
       '(is_a yes (variable F)
          (count_nounphrase 1 (QUIZ new) file (variable F) ( )))'



yes
| ?- parse(hdi,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How do I report an item at a control break to a file ?

* Lexical Phase *
CI    [how,do,i,report,an,item,at,a,control_break,to,a,file]
SML   []
IT    question
S1    success
RT1   [766,316]

* Parsing *
QAS   hdi
QS    []
S2    success
RT2   [1050,200]
PO
       '(hdi yes () report
          ((nil sub (personal_pronoun 1))
           (nil d_o (count_nounphrase 1 () item nil ( )))
           (nil at (count_nounphrase 1 () control_break nil ( )))
           (nil to (count_nounphrase 1 () file nil ( )))))'



yes                          |
| ?- parse(hdi,true)         |
                             |
SEQAP PARSER;
Enter your input [question(?) OR statement( ) OR phrase( )]
|  How do I write [a numeric item with leading zeroes]?

* Lexical Phase *
CI    [how,do,i,repor' ','[',a,numeric,item,with,leading,zeroes,']']
SML   [[write,report]]
IT    question
S1    success
RT1   [1416,333]

* Parsing *
QAS   hdi
QS    []
S2    success
RT2   [1633,150]
PO
       '(hdi yes () report
          ((nil sub (personal_pronoun 1))
           (nil d_o (count_nounphrase 1 (numeric) item nil
             ( (with (count_nounphrase nil (leading) zeroes nil ( ))))))))'



yes
| ?- parse(qauz,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  When is it true that a field is truncated ?

* Lexical Phase *
CI    [when,is,it,true,that,a,field,is,truncated]
SML   []
IT    question
S1    success
RT1   [1966,283]

* Parsing *
QAS.  qauz
QS    []
S2    success
RT2   [2233,200]
PO
       qauz
        [
          when
          is
          it
          true
          that
          a
          field
          is
          truncated
        ]
        when_is_it_true
          passive
        truncate
        yes
        nil
        nil
        d_o
          field
          c_noun
          nil
          nil
          nil
        nil



yes
| ?- parse(qauz,true)

SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
```

```
|: 'FILE' is a QUIZ file

* Lexical Phase *
CI    [var('FILE'),is,a,pn_qn('QUIZ'),file]
SML   []
IT    statement
S1    success
RT1   [2533,100]

* Parsing *
QAS   qauz
QS    []
S2    success
RT2   [2766,133]
PO
       qauz
        [
          var
            'FILE'
          is
          a
          pn_qn
            'QUIZ'
          file
        ]
         is_a
         nil
         yes
         nil
         sub
           'FILE'
           var
           nil
           nil
           nil
         d_o
           file
           c_noun
           var
             'FILE'
           [
             'QUIZ'
           ]
           nil
         nil


yes
| ?- parse(qauz,true,
                                    '
SEQAP PARSER
Enter your input [question(?) OR statement( ) OR phrase( )]
|  Why am I getting the error message 'Expected   filename"?

* Lexical Phase *
CI    [why,am,i,getting,the,error_message,quiz_stuff('Expected   filename')]
SML   []
IT    question
S1    success
RT1   [3150,200]

* Parsing *
QAS   qauz
QS    'Expected   filename'
S2    success
RT2:  [3616,383]
PO
       qauz
        [
          why
          am
          i
          getting
          the
          error_message
          quiz_stuff
            'Expected   filename'
        ]
         err
         get
         yes
         nil
         sub
           i
           pers_pro
           nil
           nil
           nil
         d_o
           error_message
           c_noun
           nil
           nil
           nil
         nil


yes
| ?- ^D
[ End of Prolog execution ]
uotcsib# ^D
script done on Mon Nov 24 15 54 54 1986
```