Natural Language Interfaces to Databases – An Introduction*

I. Androutsopoulos[†] G.D. Ritchie[†] P. Thanisch[‡]

[†]Department of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, Scotland, U.K.
e-mail: `ion@aisb.ed.ac.uk`, `G.D.Ritchie@ed.ac.uk`

[‡]Department of Computer Science, University of Edinburgh
King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland, U.K.
e-mail: `pt@dcs.ed.ac.uk`

Abstract

This paper is an introduction to natural language interfaces to databases (NLIDBs). A brief overview of the history of NLIDBs is first given. Some advantages and disadvantages of NLIDBs are then discussed, comparing NLIDBs to formal query languages, form-based interfaces, and graphical interfaces. An introduction to some of the linguistic problems NLIDBs have to confront follows, for the benefit of readers less familiar with computational linguistics. The discussion then moves on to NLIDB architectures, portability issues, restricted natural language input systems (including menu-based NLIDBs), and NLIDBs with reasoning capabilities. Some less explored areas of NLIDB research are then presented, namely database updates, meta-knowledge questions, temporal questions, and multi-modal NLIDBs. The paper ends with reflections on the current state of the art.

1 Introduction

A natural language interface to a database (NLIDB) is a system that allows the user to access information stored in a database by typing requests expressed in some natural language (e.g. English). The following example is a dialogue between the user and LOQUI, a commercially available NLIDB [15] (the dialogue is borrowed from [14]; the system's responses are slightly simplified). Throughout this paper messages printed by a NLIDB are shown in `this typeface`; user entries are shown in *this typeface*.

> *Who works on 3 projects?*
`B. Vandecapelle, C. Willems, D. Sedlock, J.L. Binot, L. Debille, ...`
> *Which of them are project leaders?*
`D. Sedlock, J.L. Binot`
> *Documents describing their projects?*
`BIM_LOQUI: "The LOQUI NLIDB", "BIM_LOQUI"`
`MMI2: "Technical Annex"`
> *How many of these projects do not finish before 1994?*

*To appear in the *Journal of Natural Language Engineering*, Cambridge University Press. Also available as Research Paper no. 709, Department of Artificial Intelligence, University of Edinburgh, 1994.

2
BIM_LOQUI, MMI2
> *Are they led by JLB or DS?*
`The former.`

The purpose of this paper is to serve as an introduction to *some* key concepts, problems, methodologies, and lines of research in the area of natural language interfaces to databases. This paper is by no means a complete discussion of all the issues that are relevant to NLIDBs. The reader is also referred to [29] and [79] for alternative surveys of the same field.[1] A detailed description of a natural language front-end that reflects the current state of the art can be found in [2] and [3].

Although the paper contains hints about the capabilities of existing NLIDBs, it does not contain complete descriptions of particular systems, nor is the purpose of this paper to compare particular NLIDBs. This paper is mainly based on information obtained from published documents. The authors do not have personal hands-on experience with most of the NLIDBs that will be mentioned. Whenever a system's feature is mentioned, this means that the documents cited state that the particular system provides this feature, and it is not implied that other systems do not have similar capabilities. Finally, this paper assumes that the user's requests are communicated to the NLIDB by typing on a computer keyboard. Issues related to speech processing are not discussed.

The remainder of this paper is organised as follows: section 2 is a brief overview of the history of NLIDBs[2]; section 3 discusses the advantages and disadvantages of NLIDBs; section 4 presents some of the linguistic problems NLIDBs have to cope with; section 5 describes architectures adopted in existing NLIDBs; section 6 discusses portability issues related to NLIDBs; section 7 introduces NLIDBs that explicitly restrict the set of natural language expressions the user is allowed to input, so that the user can have a clearer view of what sorts of questions the system can understand; section 8 describes NLIDBs with reasoning modules; section 9 highlights some less explored areas of NLIDB research, namely database updates, meta-knowledge questions, temporal questions, and multi-modal interfaces; finally, section 10 summarises the current state of the art.

2 Some history

Prototype NLIDBs had already appeared in the late sixties and early seventies. The best-known NLIDB of that period is LUNAR [106], a natural language interface to a database containing chemical analyses of moon rocks. LUNAR and other early natural language interfaces were each built having a particular database in mind, and thus could not be easily modified to be used with different databases. (Although the internal representation methods used in LUNAR were argued to facilitate independence between the database and other modules [105], the way that these were used was somewhat specific to that project's needs. Portability issues are discussed in section 6.)

By the late seventies several more NLIDBs had appeared. RENDEZVOUS [27] engaged the user in dialogues to help him/her formulate his/her queries. LADDER [58] could be used with large databases, and it could be configured to interface to different underlying database management systems (DBMSs). LADDER used semantic grammars (discussed in section 5.3),

[1]Sections 4 and 5 were greatly influenced by [79].
[2]Section 2 is largely based on information from [29], [79], and [89].

a technique that interleaves syntactic and semantic processing. Although semantic grammars helped to implement systems with impressive characteristics, the resulting systems proved difficult to port to different application domains. Indeed, a different grammar had to be developed whenever LADDER was configured for a new application. As researchers started to focus on portable NLIDBs, semantic grammars were gradually abandoned. PLANES [100] and PHILIQA1 [82] were some of the other NLIDBs that appeared in the late seventies.

CHAT-80 [101] is one of the best-known NLIDBs of the early eighties. CHAT-80 was implemented entirely in Prolog. It transformed English questions into Prolog expressions, which were evaluated against the Prolog database. The code of CHAT-80 was circulated widely, and formed the basis of several other experimental NLIDBs (e.g. MASQUE [6] [7] [5]).

In the mid-eighties NLIDBs were a very popular area of research, and numerous prototype systems were being implemented. A large part of the research of that time was devoted to portability issues. For example, TEAM [47] [48] [73] was designed to be easily configurable by database administrators with no knowledge of NLIDBs.

ASK [92] [93] allowed end-users to teach the system new words and concepts at any point during the interaction. ASK was actually a complete information management system, providing its own built-in database, and the ability to interact with multiple external databases, electronic mail programs, and other computer applications. All the applications connected to ASK were accessible to the end-user through natural language requests. The user stated his/her requests in English, and ASK transparently generated suitable requests to the appropriate underlying systems.

JANUS [59] [81] [102] [17] had similar abilities to interface to multiple underlying systems (databases, expert systems, graphics devices, etc). All the underlying systems could participate in the evaluation of a natural language request, without the user ever becoming aware of the heterogeneity of the overall system. JANUS is also one of the few systems to support temporal questions (discussed in section 9.3).

Systems that also appeared in the mid-eighties were DATALOG[3] [50] [52], EUFID [89], LDC [9] [10], TQA [30] [31], TELI [8], and many others.

Although some of the numerous NLIDBs developed in the mid-eighties demonstrated impressive characteristics in certain application areas, NLIDBs did not gain the expected rapid and wide commercial acceptance. For example, in 1985 Ovum Ltd. [63] (p.14) was foreseeing that *"By 1987 a natural language interface should be a standard option for users of DBMS and 'Information Centre' type software, and there will be a reasonable choice of alternatives."*.[4] Since then, several commercially available NLIDBs have appeared, and some of them are claimed to be commercially successful. However, NLIDBs are still treated as research or exotic systems, rather than a standard option for interfacing to databases, and their use is certainly not wide-spread. The development of successful alternatives to NLIDBs, like graphical and form-based interfaces, and the intrinsic problems of NLIDBs (both discussed in the following section) are probably the main reasons for the lack of acceptance of NLIDBs.

In recent years there has been a significant decrease in the number of papers on NLIDBs published per year. Still, NLIDBs continue to evolve, adopting advances in the general natural language processing field (e.g. discourse theories – section 8.3), exploring architectures that transform NLIDBs into reasoning agents (section 8), and integrating language and graphics to exploit the advantages of both modalities (section 9.4), to name some of the lines of current

[3] This NLIDB has nothing to do with the subset of Prolog described in [98], which is also called DATALOG.

[4] A more recent report on natural language markets by the same company is [77].

research. Generic linguistic front-ends have also appeared. These are general-purpose systems that map natural language input to expressions of a logical language (e.g. the CLE system [2] – see also [3]). These generic front-ends can be turned into NLIDBs, by attaching additional modules that evaluate the logic expressions against a database (section 5). The following are *some* of the commercially available NLIDBs:

- INTELLECT [56] from Trinzic (formed by the merger of AICorp and Aion). This system is based on experience from ROBOT [53] [54] [55].
- BBN's PARLANCE [12], built on experience from the development of the RUS [16] and IRUS [11] systems.
- IBM's LANGUAGEACCESS [76]. This system stopped being commercially available in October 1992.
- Q&A from Symantec.
- NATURAL LANGUAGE from Natural Language Inc. According to [29], this system was previously known as DATATALKER, it is described in [72], and it is derived from the system described in [46].
- LOQUI [15] from BIM.
- ENGLISH WIZARD from Linguistic Technology Corporation. The company was founded by the author of AICorp's original INTELLECT.

Some aspects of the linguistic capabilities of INTELLECT, Q&A, and NATURAL LANGUAGE are reviewed in [85].

It should be noted that when some researchers use the term "database", they often just mean "a lot of data". In this paper, we mean quite a lot more than that. Most importantly, the data is assumed to represent some coherent attempt to model a part of the world. Furthermore, the database is structured according to some model of data, and the NLIDB is designed to work with that data model. Database systems have also evolved a lot during the last decades. The term "database system" now denotes (at least in computer science) much more complex and principled systems than it used to denote in the past. Many of the underlying "database systems" of early NLIDBs would not deserve to be called database systems with today's standards.

In the early days of database systems, there was no concept of naive end-users accessing the data directly; this was done by an expert programmer writing a special computer program. The reason for this was the 'navigational' nature of the data model used by these early database systems. Not only did the user need to know about the structure of the data in the application. He/she also needed to know many programming tricks to get at the data. The development of the relational model of data in the 1970's (Codd [26]) had a major impact on database systems. In the relational model, the only storage structure is the table, and this was something that even naive users could understand. Relatively simple declarative query languages, such as SQL (see section 3), were developed for this class of user.

Currently, there are two major developments in database technology that will have an impact on NLIDBs. The first is the growing importance of object-oriented database systems, and the second is the trend in relational database technology towards more complex storage structures to facilitate advanced data modelling. We note that both of these trends could make it harder to produce an NLIDB. They both reflect a tendency to concentrate on new complex database application areas, such as network management and computer-aided design, where the user is anything but naive, and the immediate access to the database will often be carried out by a layer of application software.

3 Advantages & disadvantages

This section discusses some of the advantages and disadvantages of NLIDBs, comparing them to formal query languages, form-based interfaces, and graphical interfaces.

Access to the information stored in a database has traditionally been achieved using formal query languages, such as SQL (see [98] for an introduction to SQL). Let us assume, for example, that the following tables are stored in a relational database:

employees_table

employee	department	phone
Thompson	sales	2317
Richardson	accounting	2554
...	...	...

departments_table

department	manager	city
sales	Ferguson	London
accounting	Richardson	Bristol
...	...	...

The first table, employees_table, shows the department and phone number of each employee. The second table, departments_table, shows the manager of each department, and the city where each department is located. A list showing each employee's manager can be created using the SQL query:

```
SELECT employees_table.employee, departments_table.manager
FROM employees_table, departments_table
WHERE employees_table.department = departments_table.department
```

The SQL query above instructs the database system to report all possible pairs consisting of an employee value and a manager value, such that the employee value comes from a row in employees_table, the manager value comes from a row in departments_table, and the two rows have the same department value.

In form-based interfaces (e.g. [80]), pre-defined forms consisting of fields are used. The user fills the information which is already known in the corresponding fields, and the system completes the remaining fields by querying the underlying database. A user wishing to learn the manager of an employee named *"Thompson"* could fill in the pre-defined form "`Employee Information Form`" as shown below on the left. The system would respond by filling the remaining fields as shown below on the right.

`Employee Information Form`	
`Employee:`	*Thompson*
`Department:`	
`Phone:`	
`Manager:`	

`Employee Information Form`	
`Employee:`	`Thompson`
`Department:`	`sales`
`Phone:`	`2317`
`Manager:`	`Ferguson`

If there are two or more answers (e.g. more than one employee called *"Thompson"*), the system responds by generating a stack of filled forms, one form per possible answer.

In a more powerful method, called *query by example* (proposed by Zloof, see [98] for a description), the user can combine an arbitrary number of forms, where each form reflects the structure of a database table. The manager of *"Thompson"* could be found using the following query-by-example forms:

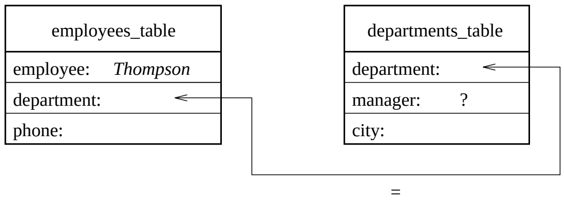

Figure 1: A graphical query

employees_table_form		
employee	department	phone
Thompson	X	

departments_table_form		
department	manager	city
X	$P.Y$	

The first form shows that the system should select from employees_table rows having *Thompson* as employee value. The two occurrences of variable X show that each row selected from employees_table should be joined with rows from departments_table that have the same department values. The $P.Y$ entry shows that the corresponding manager value is to be printed.

Finally, in one approach to graphical interfaces (similar to the one used in SunSimplify [87]) the user first selects the database tables to be used in the query (*employee_table* and *manager_table* in our example). Each selected table appears on the screen as a frame consisting of attribute-slots. The user can then fill in attribute slots by typing on the keyboard, he/she can join attributes across the various frames using the mouse, or he/she can impose restrictions on attributes using the mouse and menu options. To find the manager of *"Thompson"*, a user could use the graphical query shown in figure 1. A similar method is to allow frames to correspond to world entities (e.g. managers, employees, departments), rather than to database tables.

The following advantages and disadvantages of NLIDBs are often mentioned in the literature.

Some advantages of NLIDBs

No artificial language: One advantage of NLIDBs is supposed to be that the user is not required to learn an artificial communication language. Formal query languages are difficult to learn and master, at least by non-computer-specialists. Graphical interfaces and form-based interfaces are easier to use by occasional users; still, invoking forms, linking frames, selecting restrictions from menus, etc. constitute artificial communication languages, that have to be learned and mastered by the end-user. In contrast, an ideal NLIDB would allow queries to be formulated in the user's native language. This means that an ideal NLIDB would be more suitable for occasional users, since there would be no need for the user to spend time learning the system's communication language.

In practice, current NLIDBs can only understand limited subsets of natural language. Therefore, some training is still needed to teach the end-user what kinds of questions the NLIDB can or cannot understand. In some cases, it may be more difficult to understand what sort of questions an NLIDB can or cannot understand, than to learn how to use a formal query language, a form-based interface, or a graphical interface (see disadvantages below). One may also argue that a subset of natural language is no longer a natural language.

Better for some questions: It has been argued (e.g. [28]) that there are kinds of questions (e.g. questions involving negation, or quantification) that can be easily expressed in natural language, but that seem difficult (or at least tedious) to express using graphical or form-based interfaces. For example, *"Which department has no programmers?"* (negation), or *"Which company supplies every department?"* (universal quantification), can be easily expressed in natural language, but they would be difficult to express in most graphical or form-based interfaces. Questions like the above can, of course, be expressed in database query languages like SQL, but complex database query language expressions may have to be written.

Discourse: Another advantage of NLIDBs, mentioned in [57] and [28], concerns natural language interfaces that support anaphoric and elliptical expressions (sections 4.5 and 4.6). NLIDBs of this kind allow the use of very brief, underspecified questions, where the meaning of each question is complemented by the discourse context. In formal query languages, graphical interfaces, and form-based interfaces this notion of discourse context is usually not supported. This point will become clearer in sections 4.5 and 4.6.

Some disadvantages of NLIDBs

Linguistic coverage not obvious: A frequent complaint against NLIDBs is that the system's linguistic capabilities are not obvious to the user [90] [57] [28]. As already mentioned, current NLIDBs can only cope with limited subsets of natural language. Users find it difficult to understand (and remember) what kinds of questions the NLIDB can or cannot cope with. For example, MASQUE [7] is able to understand *"What are the capitals of the countries bordering the Baltic* and *bordering Sweden?"*, which leads the user to assume that the system can handle all kinds of conjunctions (false positive expectation). However, the question *"What are the capitals of the countries bordering the Baltic* and *Sweden?"* cannot be handled. Similarly, a failure to answer a particular query can lead the user to assume that "equally difficult" queries cannot be answered, while in fact they can be answered (false negative expectation).

Formal query languages, form-based interfaces, and graphical interfaces typically do not suffer from these problems. In the case of formal query languages, the syntax of the query language is usually well-documented, and any syntactically correct query is guaranteed to be given an answer. In the case of form-based and graphical interfaces, the user can usually understand what sorts of questions can be input, by browsing the options offered on the screen; and any query that can be input is guaranteed to be given an answer.

Linguistic vs. conceptual failures: When the NLIDB cannot understand a question, it is often not clear to the user whether the rejected question is outside the system's *linguistic* coverage, or whether it is outside the system's *conceptual* coverage [90]. Thus, users often try to rephrase questions referring to concepts the system does not know (e.g. rephrasing questions about salaries towards a system that knows nothing about salaries), because they think that the problem is caused by the system's limited linguistic coverage. In other cases, users do not try to rephrase questions the system could conceptually handle, because they do not realise that the particular phrasing of the question is outside the linguistic coverage, and that an alternative phrasing of the same question could be answered. Some NLIDBs attempt to solve this problem by providing diagnostic messages, showing the reason a question cannot be handled (e.g. unknown word, syntax too complex, unknown concept, etc.) (see section 5.5).

Users assume intelligence: As pointed out in [57], NLIDB users are often misled by the system's ability to process natural language, and they assume that the system is intelligent, that it has common sense, or that it can deduce facts, while in fact most NLIDBs have no reasoning abilities (see however section 8). This problem does not arise in formal query languages, form-based interfaces, and graphical interfaces, where the capabilities of the system are more obvious to the user.

Inappropriate medium: As mentioned in [15], it has been argued that natural language is not an appropriate medium for communicating with a computer system. Natural language is claimed to be too verbose or too ambiguous for human-computer interaction. NLIDB users have to type long questions, while in form-based interfaces only fields have to be filled in, and in graphical interfaces most of the work can be done by mouse-clicking. Natural language questions are also often ambiguous, while formal, form-based, or graphical queries never have multiple meanings. Section 4 presents some of the problems in this area, and discusses how NLIDBs attempt to address them.

Tedious configuration: NLIDBs usually require tedious and lengthy configuration phases before they can be used (see section 6). In contrast, most commercial database systems have built-in formal query language interpreters, and the implementation of form-based interfaces is largely automated.

Several experiments have been carried out, comparing how users cope with formal query languages, form-based interfaces, graphical interfaces, and NLIDBs.

[13] describes one such experiment, during which fifty five subjects, ranging from computer novices to programmers, were asked to perform database queries using a formal query language (SQL), a graphical interface (SunSimplify – see section 3), and a NLIDB (DATATALKER, later known as NATURAL LANGUAGE – see section 2). The subjects first received some training on how to use the three interfaces, and were then asked to perform database queries, most of which were similar to the queries the subjects had encountered during the training period. The experiment measured the number of queries the subjects managed to perform successfully, and the average time the subjects used to perform each successful query. None of the three interfaces could be said to be a winner. Each interface was better in some kinds of queries, and in most queries the subjects' performance was roughly the same, whether the subjects were asked to use SQL, the graphical interface, or the NLIDB. Roughly speaking, the NLIDB seemed to be better in queries where data from many tables had to be combined, and in queries that were not similar to the ones the users had encountered during the training period. Information about similar experiments can be found in [62] and [86].

[103] discusses various approaches that have been used in the evaluation of natural language systems, and describes an experiment where first a Wizard of Oz was used to collect sample user questions, and then the sample questions were used as input to an actual NLIDB. In a Wizard of Oz experiment, the user interacts with a person who pretends to be an NLIDB through a computer network. The user is not aware that he/she is not interacting with a real NLIDB. [74] provides a set of questions towards a NLIDB that was collected using a Wizard of Oz experiment. [37] describes in detail a Wizard of Oz experiment.

[18] provides information about several experiments on the usability of NLIDBs, and describes in detail one such experiment that assessed the usability of INTELLECT (see section 2). The authors conclude that "*natural language is an effective method of interaction for*

casual users with a good knowledge of the database, who perform question-answering tasks, in a restricted domain". [36] also describes an experiment on the usability of INTELLECT, and concludes that NLIDBs are a practical alternative.

4 Linguistic problems

This section presents *some* of the linguistic problems a NLIDB has to confront when attempting to interpret a natural language request. Most of these problems do not arise only in NLIDBs; they are common in most kinds of natural language understanding systems. It is stressed that this section is by no means a full list of the linguistic problems NLIDBs have to confront. This section merely presents some of the problems, mainly for the benefit of readers not familiar with computational linguistics.

The reader is reminded that messages generated by the NLIDB are printed in `this typeface`, and that user entries are printed in *this typeface*.

4.1 Modifier attachment

Let us consider the request *"List all employees in the company with a driving licence"*. Linguistically, both *"in the company"* and *"with a driving licence"* are *modifiers*, i.e. they modify the meaning of other syntactic constituents. The problem is to identify the constituent to which each modifier has to be attached.

A human would immediately understand that *"in the company"* refers to *"employees"*, and that *"with a driving licence"* refers to the *"employees in the company"*. This reading of the request corresponds to the structure:

List all [[employees [in the company]] [with a driving licence]].

and can be paraphrased as *"From the employees working in the company, list those having a driving licence"*. However, a computer system would also consider the following reading:

List all [employees [in the [company [with a driving licence]]]].

where *"with a driving licence"* refers to the *"company"*. The latter reading can be paraphrased as *"List all employees working in the company which has a driving licence"*. This second reading cannot be easily ruled out. For example, in the similar request *"List all employees in the company with an export licence."*, it is the second reading that probably has to be chosen (*"with an export licence"* refers to *"the company"*). To be able to choose the correct reading, the system must know that in the particular application domain employees can have driving licences but not export licences, and that companies can have export licences but not driving licences.

Alternatively, some systems attempt to resolve modifier attachment ambiguities by using heuristics. PRE [40] uses a "most right association principle" to resolve relative clauses attachment: relative clauses are assumed to modify the rightmost available constituent. For example, in *"List an employee who was hired by a recruiter whose salary is greater than $3,000."*, the relative clause *"whose salary is greater than $3,000"* is assumed to refer to the *"recruiter"*, not the *"employee"*. More advanced disambiguation techniques for prepositional phrase attachment can be found in [104].

Still, in some cases modifier attachment is truly ambiguous. Consider the request (borrowed from [79]) *"List all employees in the division making shoes"*. In some application domains *"making shoes"* may equally well refer to the *"division"* or to the *"employees"*. In such cases, a NLIDB can (a) present to the user paraphrases of both readings, and ask the user to chose one of the readings, or (b) it can print answers corresponding to both readings, indicating which answers refer to which readings.

4.2 Quantifier scoping

Determiners like *"a"*, *"each"*, *"all"*, *"some"* are usually mapped to logic quantifiers. In cases of sentences with many words of this kind, it is difficult to determine which quantifier should be given a wider scope. Let us consider the question *"Has every student taken some course?"*. The question has two readings. (*student* and *course* are variables.)

1. Check that
   $\forall student\ \exists course\ taken(student, course)$
2. Check that
   $\exists course\ \forall student\ taken(student, course)$

In the first reading, each student is allowed to have taken a different course, while in the second reading all students must have taken the same course.

A possible heuristic method to resolve scoping ambiguities [79] is to prefer scopings preserving the left-to-right order of the quantifiers. In *"Has every student taken some course?"*, *"every"* appears before *"some"*. Therefore, according to the heuristic, the first of the readings above should be preferred, since in this reading the universal quantifier introduced by *"every"*, precedes the existential quantifier introduced by *"some"*.

Another approach [79] is to associate a numeric strength to each determiner. In this approach, determiners with greater strengths are given wider scopes. Returning to the question *"Has every student taken some course?"*, if the strength of *"every"* is greater than the strength of *"some"*, then the first reading is again preferred. The reader is referred to chapter 8 of [2] for a detailed discussion of methods that can be used to resolve quantifier scoping problems.

4.3 Conjunction and disjunction

The word *"and"* is often used to denote disjunction rather than conjunction. This introduces ambiguities which are difficult to resolve. The following two examples, borrowed from [89], illustrate the problem:

In *"List all applicants who live in California and Arizona."* every applicant who lives *either* in California or Arizona should be reported. The possibility for *"and"* to denote a conjunction should be ruled out, since an applicant cannot normally live in more than one state. EUFID [89] detects cases where it is conceptually impossible for *"and"* to denote a conjunction, and turns *"and"*s to *"or"*s. (Section 4.2 of [99] describes a heuristic rule for detecting cases where *"and"* cannot have a conjunctive meaning.)

"Which minority and female applicants know Fortran?", however, could mean either *"Which female applicants belonging to a minority know Fortran?"* or *"Which applicants who are female or belong to a minority know Fortran?"*. In such cases *"and"* is truly ambiguous.

As in the case of EUFID, when asked *"How many people live in Belmont and Boston?"*, PARLANCE [12] understands that *"and"* cannot denote a conjunction, and reports the population of each city. In contrast, when asked *"What's the average salary of blacks and hispanics?"*, PARLANCE provides answers corresponding to both possible meanings, by generating the following tables:

Average Salary	Ethnic Group	Count
$42,481.81	Black	11
$45,230.00	Hispanic	10

Average Salary	Count
$43,790.47	21

4.4 The nominal compound problem

In English, nouns are often modified by other preceding nouns. The meaning of the resulting compound constituent is hard to predict. The following examples, based on examples from [79], illustrate the problem:

- *"city department"* could denote a department located in a city, or a department responsible for a city.
- *"research department"* is probably a department *carrying out* research.
- *"research system"* is probably a system *used in* research, it is not a system carrying out research.

Similar difficulties arise in the case of adjective-noun compounds. For example, a *"large company"* may be a company with a large volume of sales or a company with many employees, a *"big department"* may be a department with many employees or a department occupying a lot of space, etc.

Because of the difficulty in determining the meaning of noun-noun and adjective-noun compounds from the meanings of the individual compounded nouns or adjectives, some NLIDBs require the meaning of each possible noun-noun or adjective-noun compound to be declared during the configuration phase (see section 6). In LDC-1 [9] [10], for example, whenever the system is taught a new noun (e.g. *"department"*), the configurer is asked for all adjectives (e.g. *"large"*, *"small"*, *"profitable"*) and nouns (e.g. *"sales"*, *"city"*) that can precede the new noun as modifiers. The configurer is then asked to define the meaning of each possible compound (*"large department"*, *"profitable department"*, *"sales department"*, etc.) in terms of concepts of the underlying database.

4.5 Anaphora

Pronouns (e.g. *"she"*, *"they"*), possessive determiners (e.g. *"his"*, *"their"*), and noun phrases (e.g. *"the project"*, *"these people"*) are often used to denote implicitly entities mentioned in the discourse, a linguistic phenomenon called *anaphora*. Consider the following dialogue between the user and ASK from [92]:

> \> *Is there a ship whose destination is unknown?*
> `yes`
> \> *What is it?*
> `What is [the ship whose destination is unknown]?`
> `Saratoga`

ASK has understood that *"it"* refers to *"the ship whose destination is unknown"*. ASK echoes back the user's question, with the pronoun replaced by what the system understands to be its meaning. This prevents the user from being misled, in cases where the pronoun anaphora is not resolved correctly.

The following is a (shortened) dialogue with LOQUI from [14]:

> *Who leads* TPI*?*

`E. Feron`

> *Who reports to him?*

`C. Leonard, C. Willems, E. Bidonnet, P. Cayphas, J.P. Van Loo`

> *What do they work on?*

project	worker
DOCDIS	C. Willems
	J.P. Van Loo
	P. Cayphas
EURS	C. Leonard
	C. Willems
	E. Bidonnet

> *Which of these are leaders?*

`J.P. Van Loo`

A simplistic method to handle pronoun anaphora is to keep a list of all entities mentioned in the discourse. When a pronoun is encountered, the system examines the list, usually starting from the most recent entries, and associates each pronoun with the most recently mentioned entity that satisfies grammatic and semantic constraints.

In *"Who reports to him?"* above, a system could reason that *"him"* refers to *"E. Feron"* rather than to the project "TPI", because the pronoun is of masculine gender, while a project is of neutral gender. In *"Which of these are leaders?"* above, LOQUI has probably reasoned that *"these"* refers to the people mentioned in the previous response, rather than to the projects mentioned in the previous response, using information stating that only persons not projects can be leaders.

Possessive determiner anaphora (e.g. *"their"*) is similar to pronoun anaphora. The following example is also from [14]:

> *Who reports to MM?*

`E. Feron, J.L. Binot`

> *What are their projects?*

project	person
BIM_LOQUI	J.L. Binot
EURS	E. Feron

Noun phrases can also be used as anaphoric expressions. In the following dialogue, from [14], *"the project"* is an anaphoric noun phrase referring to the project "MMI2":

> *Who leads* MMI2*?*

`J.L. Binot`

> *Who works on the project?*

`B. Vandecapelle, D. Sedlock, F. Balfroid, J.L. Binot, L. Debille`

Anaphora is a well-known problem in natural language processing. See [60] for an overview of the issues, and [61] for relatively simple methods that can be used to resolve many cases of pronoun anaphora.

4.6 Elliptical sentences

In every-day discourse people use incomplete (elliptical) sentences, the meaning of which is complemented by the discourse context. This phenomenon is illustrated in the following dialogue (from [12]) between the user and PARLANCE :

> *Does the highest paid female manager have any degrees from Harvard?*

`Yes, 1.`

> *How about* MIT*?*

`No, none.`

> *Who is the manager of the largest department?*

Name	Dept.	Count
Patterson	045	40

> *The smallest department?*

Name	Dept.	Count
Saavedra	011	2

"How about MIT*?"* and *"The smallest department?"* are elliptical sentences.

Handling complex cases of elliptical sentences requires a discourse model to be maintained by the system (see section 8.3). XCALIBUR [19] uses a set of *"discourse expectation"* and *"contextual substitution"* rules, that allow elliptical utterances, such as the ones shown below, to be understood:

> *What is the price of the three largest single port fixed media disks?*
> *speed?*
> *two smallest?*
> *How about the price of the two smallest?*
> *Also the smallest with two ports?*
> *speed with two ports?*

The handling of elliptical sentences is a desirable feature in a NLIDB. NLIDB users typically do not have single stand-alone questions to ask. The first, often exploratory query, is usually followed by many elliptical follow-up queries. Some NLIDBs (e.g. CHAT-80 [101]) are oriented towards the processing of stand-alone questions, providing no support for elliptical questions. In such systems, the questions shown above would have to be formulated as:

> *What is the price of the three largest single port fixed media disks?*
> *What is the speed of the three largest single port fixed media disks?*
> *What is the speed of the two smallest single port fixed media disks?*
> *What is the price of the two smallest single port fixed media disks?*
> *What is the price of the two smallest fixed media disks with two ports?*
> *What is the speed of the two smallest fixed media disks with two ports?*

Having to type every question in full can be annoying for the user, though the problem can be alleviated by providing a facility for editing previously entered questions.

4.7 Extragrammatical utterances

Most linguistic theories describe the structure and meaning of grammatical, perfectly formed utterances. However, every-day language often contains misspellings, syntactically ill-formed input, telegraphic utterances, etc. If the main goal of a NLIDB is to assist the user, then the system must be able to understand the user's requests, even when they are partially ill-formed.

Most of the existing NLIDBs can handle only simple cases of extragrammatical utterances. For example, PARLANCE forgives dropped articles, sentences where subject and verb cases or numbers do not agree, capitalisation errors and punctuation errors [12]. It is, thus, able to understand sentences like *"whats reynolds departments name"*, or *"people in dePT 45 who is supervisory scientist"*. ASK uses its dictionary to correct spelling errors and errors caused by missing spaces [93]. In *"What is the crago of the Orient Clipper?"* it converts *"crago"* to *"cargo"*, and in *"What is the cargoof the Orient Clipper?"* it converts *"cargoof"* to *"cargo of"*.

5 Architectures

5.1 Pattern-matching systems

Some of the early NLIDBs relied on pattern-matching techniques to answer the user's questions. To illustrate a simplistic pattern-matching approach, consider a database table holding information about countries:

COUNTRIES_TABLE

COUNTRY	CAPITAL	LANGUAGE
France	Paris	French
Italy	Rome	Italian
...	...	...

A primitive pattern-matching system could use rules like:

pattern: ... *"capital"* ... <country>
action : Report CAPITAL of row where COUNTRY = <country>

pattern: ... *"capital"* ... *"country"*
action : Report CAPITAL and COUNTRY of each row

The first rule says that if a user's request contains the word *"capital"* followed by a country name (i.e. a name appearing in the COUNTRY column), then the system should locate the row which contains the country name, and print the corresponding capital.

If, for example, the user typed *"What is the capital of Italy?"*, the system would use the first pattern-rule, and report "`Rome`". The same rule would allow the system to handle *"Print the capital of Italy."*, *"Could you please tell me what is the capital of Italy?"*, etc. In all cases the same response would have been generated.

According to the second rule, any user request containing the word *"capital"* followed by the word *"country"* should be handled by printing the capital of each country, as listed in the database table. For example, *"What is the capital of each country?"*, *"List the capital of every country."*, *"Capital and country please."*, etc., would all be handled by the second rule.

The main advantage of the pattern-matching approach is its simplicity: no elaborate parsing and interpretation modules (see later sections) are needed, and the systems are easy to implement. Also, pattern-matching systems often manage to come up with some reasonable answer, even if the input is out of the range of sentences the patterns were designed to handle. Returning to the example above, the second rule would allow the system to answer

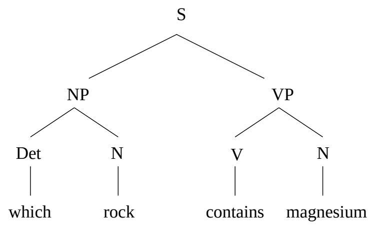

Figure 2: Parse tree in a syntax-based system

the question *"Is it true that the capital of each country is Athens?"*, by listing the capital of each country, which can be considered as an indirect negative answer.

Pattern-matching systems are not necessarily based on such simplistic techniques as the ones discussed above. SAVVY, a pattern matching system discussed in [63] (p.153), employs pattern-matching techniques similar to the ones used in signal processing.

According to [63], some pattern-matching systems were able to perform impressively well in certain applications. However, the shallowness of the pattern-matching approach would often lead to bad failures. In one case (mentioned in [63]), when a pattern-matching NLIDB was asked "TITLES OF EMPLOYEES IN LOS ANGELES.", the system reported the state where each employee worked, because it took "IN" to denote the post code of Indiana, and assumed that the question was about employees and states.

5.2 Syntax-based systems

In syntax-based systems the user's question is parsed (i.e. analysed syntactically), and the resulting parse tree is directly mapped to an expression in some database query language. A typical example of this approach is LUNAR [106].

Syntax-based systems use a grammar that describes the possible syntactic structures of the user's questions. The following example, based on a similar example from [79], shows an over-simplistic grammar in a LUNAR-like system.

$$\begin{array}{lcl}
S & \rightarrow & NP \quad VP \\
NP & \rightarrow & Det\, N \\
Det & \rightarrow & \text{"what"} \mid \text{"which"} \\
N & \rightarrow & \text{"rock"} \mid \text{"specimen"} \mid \text{"magnesium"} \mid \text{"radiation"} \mid \text{"light"} \\
VP & \rightarrow & V \quad N \\
V & \rightarrow & \text{"contains"} \mid \text{"emits"}
\end{array}$$

The grammar above says that a sentence (S) consists of a noun phrase (NP) followed by a verb phrase (VP), that a noun phrase consists of a determiner (Det) followed by a noun (N), that a determiner may be *"what"* or *"which"*, etc. Using this grammar, a NLIDB could figure out that the syntactic structure of the question *"which rock contains magnesium"* is as shown in the parse tree of figure 2. The NLIDB could then map the parse tree of figure 2 to the following database query (`X` is a variable):

```
(for_every X (is_rock X)
             (contains X magnesium) ;
             (printout X))
```

$S \rightarrow Specimen\_question \mid Spacecraft\_question$
$Specimen\_question \rightarrow Specimen\ Emits\_info \mid Specimen\ Contains\_info$
$Specimen \rightarrow$ *"which rock"* | *"which specimen"*
$Emits\_info \rightarrow$ *"emits"* $Radiation$
$Radiation \rightarrow$ *"radiation"* | *"light"*
$Contains\_info \rightarrow$ *"contains"* $Substance$
$Substance \rightarrow$ *"magnesium"* | *"calcium"*
$Spacecraft\_question \rightarrow Spacecraft\ Depart\_info \mid Spacecraft\ Arrive\_info$
$Spacecraft \rightarrow$ *"which vessel"* | *"which spacecraft"*
$Depart\_info \rightarrow$ *"was launched on"* $Date$ | *"departed on"* $Date$
$Arrive\_info \rightarrow$ *"returns on"* $Date$ | *"arrives on"* $Date$

Figure 3: A semantic grammar

which would then be evaluated by the underlying database system. This mapping would be carried out by rules, and would be completely based on the syntactic information of the parse tree. The system of our example could use mapping rules stating that:

- The mapping of *"which"* is `for_every X`.
- The mapping of *"rock"* is `(is_rock X)`.
- The mapping of an *NP* is Det' N', where Det' and N' are the mappings of the determiner and the noun respectively. Thus, the mapping of the *NP* subtree in our example is `for_every X (is_rock X)`.
- The mapping of *"contains"* is `contains`.
- The mapping of *"magnesium"* is `magnesium`.
- The mapping of a *VP* is `(`V' `X` N'`)`, where V' is the mapping of the verb, and N' is the mapping of the noun-sister of the verb. Thus, the mapping of the *VP* subtree in our example is `(contains X magnesium)`.
- The mapping of an S is `(`NP' VP' `; (printout X))`, where NP', VP' are the mappings of the *NP* and *VP* subtree accordingly. Thus, in our example, the mapping of the sentence is as shown above.

Syntax-based NLIDBs usually interface to application-specific database systems, that provide database query languages carefully designed to facilitate the mapping from the parse tree to the database query. It is usually difficult to devise mapping rules that will transform directly the parse tree into some expression in a real-life database query language (e.g. SQL).

5.3 Semantic grammar systems

In semantic grammar systems, the question-answering is still done by parsing the input and mapping the parse tree to a database query. The difference, in this case, is that the grammar's categories (i.e. the non-leaf nodes that will appear in the parse tree) do not necessarily correspond to syntactic concepts. Figure 3 shows a possible semantic grammar.

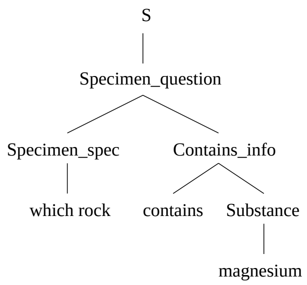

Figure 4: Parse tree in a semantic grammar

The reader will have noticed that some of the categories of the grammar of figure 3 (e.g. *Substance*, *Radiation*, *Specimen_question*) do not correspond to syntactic constituents (e.g. noun-phrase, noun, sentence). Semantic information about the knowledge domain (e.g. that a question may either refer to specimens or spacecrafts) is hard-wired into the semantic grammar. Semantic grammar categories are usually chosen to enforce semantic constraints. For example, the grammar above does not allow *"light"* to follow the verb *"contains"*. (In contrast, the grammar of the previous section allows *"contains light"*.)

The grammar's categories can also be chosen to facilitate the mapping from the parse tree to database objects. For example, in the parse tree for *"which rock contains magnesium"*, shown in figure 4, the existence of a `Specimen_question` node could direct the system to consult database tables containing information about specimens, rather than tables containing information about departures and arrivals. Similarly, the `Contains_info` subtree could direct the system to search a table called `Contains_info`, and locate the rows of which the `Substance` column is *"magnesium"*. According to [79], semantic grammars are used in PLANES [100], LADDER [58], and REL [94] [95]. A semantic grammar is also used in EUFID [89].

Semantic grammars were introduced as an engineering methodology, which allows semantic knowledge to be easily included in the system. However, since semantic grammars contain hard-wired knowledge about a specific knowledge domain, systems based on this approach are very difficult to port to other knowledge domains (see section 6): a new semantic grammar has to be written whenever the NLIDB is configured for a new knowledge domain. For example, the semantic grammar of figure 3 is completely useless in an application where the database contains information about employees and salaries. In contrast, (at least) some of the information of the grammar in the previous section (e.g. that a sentence consists of a noun phrase followed by a verb phrase) could probably still be used in an application where the questions refer to employees and their salaries.

5.4 Intermediate representation languages

Most current NLIDBs first transform the natural language question into an intermediate logical query, expressed in some internal meaning representation language. The intermediate logical query expresses the meaning of the user's question in terms of high level world concepts, which are independent of the database structure. The logical query is then translated to an expression in the database's query language, and evaluated against the database. Actually,

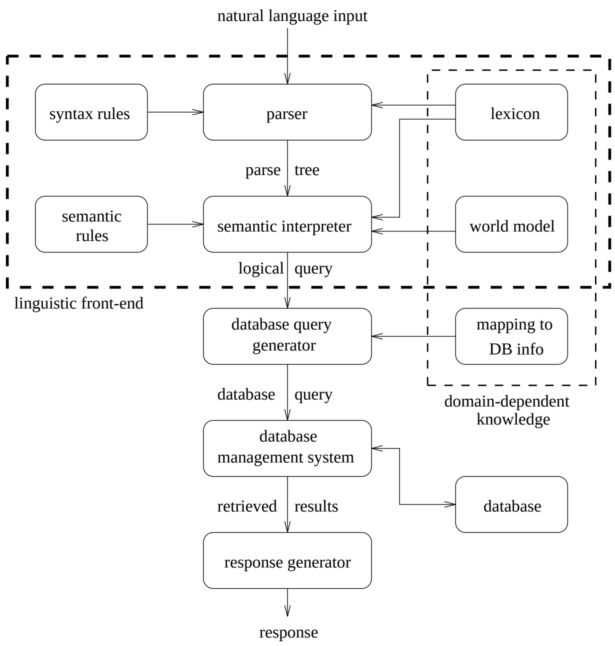

Figure 5: Possible architecture of intermediate representation language system

many modern natural language front-ends (e.g. CLE [2]) use several intermediate meaning representation languages, not just one.

Figure 5 shows a possible architecture of an intermediate representation language system. A similar architecture is used in MASQUE/SQL [4] [5]. The natural language input is first processed syntactically by the parser. The parser consults a set of syntax rules, and generates a parse tree, similar to the one shown in figure 2. (The reader is referred to [42] for information on parsing techniques.) The semantic interpreter subsequently transforms the parse tree to the intermediate logic query, using semantic rules similar to the mapping rules of section 5.2.

Some systems (e.g. JANUS [59]) build on the Montague-semantics tradition [71] [38], and carry out the semantic interpretation in a compositional, rule-to-rule manner. Each syntax rule is coupled to a semantics rule. The semantics rule computes the logic expression of the constituent in the left-hand side of the syntax rule, as a function of the logic expressions of the constituents in the right-hand side of the syntax rule. The logic expressions corresponding to words are declared in the lexicon.

In early NLIDBs the syntax/semantics rules were based on rather ad hoc ideas, and expressed in idiosyncratic formalisms. In modern systems the syntax/semantics rules are becoming increasingly influenced by principled linguistic theories, and they are often expressed in variations of well-known formalisms. LOQUI [15], for example, uses a grammar influenced by GPSG [43], and CLE's [2] grammar is expressed in a unification-based formalism similar to PATR-II [84].

In many systems the syntax rules linking non-terminal symbols (non-leaf nodes in the parse tree) and the corresponding semantic rules are domain-independent; i.e. they can be used in any application domain. The information, however, describing the possible words (leaf nodes) and the logic expressions corresponding to the possible words is domain-dependent,

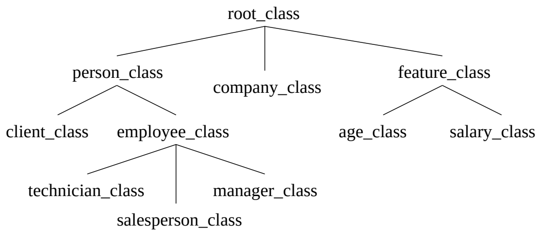

Figure 6: A hierarchy in a world model

and has to be declared in the *lexicon*.

In MASQUE [6] [7], for example, the lexicon lists the possible forms of the words that may appear in the user's questions (e.g. *"capital"*, *"capitals"*, *"border"*, *"borders"*, *"bordering"*, *"bordered"*), and logic expressions describing the meaning of each word. For example, the logic expression of *"capital"* (as in *"What is the capital of Italy?"*) could be $capital\_of(Capital, Country)$, where $Capital$ is a variable standing for a capital, and $Country$ is a variable standing for the corresponding country.

Similarly, the logic expression of the verb *"to border"* (as in *"Which countries border Germany?"*) could be $borders(Country_1, Country_2)$, and the logic expression of the noun *"country"* (as in *"Italy is a country"*) could be $is\_country(Country)$.

The semantic interpretation module uses the logic expressions of the words in the lexicon, to generate the logical query. For example, in MASQUE, the question *"What is the capital of each country bordering Greece?"* would be mapped to the following logical query. (Variable names start with capital letters.)

```
answer([Capital, Country]):-
   is_country(Country),
   borders(Country, greece),
   capital_of(Capital, Country).
```

The logic query above states that the meaning of the user's question is to find all pairs `[Capital, Country]`, such that `Country` is a country, `Country` borders Greece, and `Capital` is the capital of `Country`. The reader will have noticed that the logical query was constructed using the logic expressions of *"capital"*, *"country"*, and *"to border"*.

The semantic interpreter of figure 5 also consults a *world model*, that describes the structure of the surrounding world. Typically, the world model contains a *hierarchy* of classes of world objects, and *constraints* on the types of arguments each logic predicate may have. Figure 6 shows a world-model hierarchy of MASQUE, for a business application. Similar hierarchies are used in TEAM [47] [48] and CLE [2]. The hierarchy of figure 6 states that, in the particular application domain, a person can be either a client or an employee, that employees are divided into technicians, salespersons, and managers, etc. The hierarchy is typically used in combination with constraints on predicate arguments. These constraints specify the classes to which the arguments of a logic predicate may belong.

To illustrate the need for predicate-argument constraints, let us consider the business application of figure 6, and let us assume that the underlying database holds information about the salaries of employees, but that it does not hold information about the salaries of people not employed by the company. (Let us also assume that an employee cannot also be a client of the company.) Then, the question *"What is the salary of each client?"* will retrieve no information, because the database does not contain information about the salaries of clients. We would like the NLIDB to be able to detect that the question is conceptually problematic.

Let us assume that the logic expression of *"salary"* is $salary\_of(Salary, Person)$, and that the logic expression of *"client"* is $is\_client(Person)$. The corresponding MASQUE logical query would be:

```
answer([Salary, Person]):-
   is_client(Person),
   salary_of(Salary, Person).
```

Using predicate-argument constraints, we could specify that the argument of is_client must belong to the class (must be of type) $client\_class$, and that the arguments of $salary\_of$ must belong to the classes $salary\_class$ and $employee\_class$ respectively. These constraints would require the variable `Person` in the logical query to belong to both $client\_class$ and $employee\_class$. Since, neither $client\_class$ nor $employee\_class$ is a descendant of each other in the hierarchy of figure 6, the system would be able to reason that the logical query contains a class mismatch. The class mismatch could then be reported to the user. In contrast, assuming that the logic expression of *"technician"* is $is\_technician(Person)$, where $Person$ is a variable of type $technician\_class$, the question *"What is the salary of each technician?"* will generate the MASQUE logical query:

```
answer([Salary, Person]):-
   is_technician(Person),
   salary_of(Salary, Person).
```

In this case, $Person$ is required to belong to $technician\_class\ and\ employee\_class$. Since $employee\_class$ is an ancestor of (i.e. subsumes) $technician\_class$, the logical query contains no class mismatch.

The predicate-argument constraints can also help the system to resolve natural language ambiguities. Consider, for example, the request *"List all employees in a company with a driving licence"*. From the NLIDB's point of view, the request contains a modifier attachment ambiguity (see section 4.1): *"with a driving licence"* may refer either to *"employees"* or to *"company"*. In the first case, a MASQUE-like logic query would be:

```
answer([Employee]):-
   is_employee(Employee),
   is_company(Company),
   is_drv_licence(Driv_lic),
   in(Employee, Company),
   has(Employee, Driv_lic).
```

while in the second case (*"with a driving licence"* refers to *"company"*), the logic query would look like:

```
answer([Employee]):-
   is_employee(Employee),
   is_company(Company),
   is_drv_licence(Driv_lic),
   in(Employee, Company),
   has(Company, Driv_lic).
```

The world model could specify that the arguments of *is_employee*, *is_company*, and *is_drv_licence* are of types *employee_class*, *company_class*, and *drv_licence_class* respectively, and that if the second argument of *has* is of type *drv_licence_class*, then the first argument must be of type *person_class* (i.e. only persons are allowed to have driving licences). The hierarchy could then be used to state that *person_class* subsumes *employee_class* but not *company_class* (i.e. companies are not persons). This would rule out the second logical query above, where the second argument of *has* is of type *drv_licence_class*, and the first one is not of type *person_class*.

The logic query generated by the parsing and semantic interpretation module, expresses the meaning of the user's question in terms of logical, high-level concepts. The logic query does not refer to database objects (e.g. tables, columns), and it does not specify how to search the database to retrieve the necessary information. In order to retrieve the information requested by the user, the logic query has to be transformed into a query expressed in some database query language supported by the underlying DBMS (Database Management System). In figure 5 this transformation is carried out by the *database query generator*, using the *mapping to database information*.

The mapping to database information specifies how logic predicates relate to database objects. In the case of an interface to a relational database, the simplest approach would be to link each logic predicate to an SQL `SELECT` statement, selecting rows from one or more database tables. Let us assume, for example, that the database contains the tables:

countries_table

country	capital	language
France	Paris	French
Spain	Madrid	Spanish
Germany	Berlin	German
...	...	...

borders_table

country1	country2
France	Spain
France	Germany
Germany	France
...	...

Let us also consider the MASQUE logic query for *"What is the capital of each country bordering Greece?"*:

```
answer([Capital, Country]):-
   is_country(Country),
   borders(Country, greece),
   capital_of(Capital, Country).
```

The mapping information could link the predicate `is_country` to the SQL query:

```
SELECT country
FROM countries_table
```

Such linking would mean that `is_country` is true if its argument appears in the column *country* of *countries_table*. Similarly, `borders` could be linked to the SQL query:

```
SELECT country1, country2
FROM borders_table
```

which would mean that `borders` is true if there is a row in *borders_table*, such the first argument of `borders` is the *country*1 of the row, and the second argument of `borders` is the *country*2 of the row. Finally, `capital_of` could be linked to:

```
SELECT capital, country
FROM countries_table
```

A similar (but more complex) approach is used in MASQUE/SQL [4] [5]. Using the mapping information, MASQUE/SQL would generate an SQL query similar to:

```
SELECT countries_table.capital, countries_table.country
  FROM countries_table, borders_table
 WHERE countries_table.country = borders_table.country1
   AND borders_table.country2 = 'Greece'
```

The transformation of the logic query to a database query does not need to be made in one stage. In [34] [33] [69], for example, the writers describe a principled multi-stage transformation process, used in the NLIDB developed at the University of Essex. The Essex system first generates a logic query, expressed in a version of untyped λ-calculus [38]. The λ-calculus expression is then transformed into a first-order predicate logic expression, which is subsequently translated into universal-domain relational calculus, domain relational calculus, tuple relational calculus, and finally SQL (see [98] for a description of the various forms of relational calculus).

In figure 5 the database query generated by the database query generator is passed to the underlying DBMS, which executes the query against the database, and passes the retrieved data to the *response generator*. The latter is responsible for reporting the retrieved information to the user. Issues related to response generation are discussed in the following section.

In most modern NLIDBs the underlying database is assumed to be relational. Some NLIDBs support multiple underlying, possibly heterogeneous, databases (see section 5.6). Research systems often assume idiosyncratic database models, especially designed to facilitate the development of the NLIDB.

When constructing a new NLIDB, one design issue is the 'division of labour' between the NLIDB software and the DBMS's query language. The *de facto* industry standard query language is SQL. As a language, standard SQL is less expressive than a programming language, since it lacks constructs for iteration and recursion, amongst other things. Thus there are computationally reasonable queries that might be expressed in natural language, but which cannot be answered by a single SQL query. Of course, most current NLIDBs support only a limited subset of natural language. In many cases, this subset may be so narrow that less queries can be expressed in the supported natural language subset than in SQL. The point being made here, however, is that *full* natural language is more expressive than single SQL queries. Thus, as the natural language subset supported by the linguistic component grows, there will be inevitably natural language queries that cannot be mapped to single SQL queries.

In such cases, the designer of the NLIDB must choose between either restricting the natural language subset to make it match the expressivity of single SQL queries, or allowing a natural language query to be translated into some data-dependent number of SQL queries, and piecing together the answer to the original natural language query from the set of answers to the SQL queries. Thus the iteration or recursion would have to be controlled from within the NLIDB software.

Developments in SQL are likely to have an impact on the design of NLIDBs. SQL3 has a richer set of storage structures, types and querying constructs than SQL. On the one hand, this means that more can be done in the DBMS, and less in the NLIDB, but on the other hand, broader linguistic coverage will be needed in the NLIDB to ensure that the supported natural language subset captures the expressivity of the DBMS's query language.

One of the advantages of NLIDBs based on intermediate representation languages is that the *linguistic front-end* (the part of the system that generates the logic queries – see figure 5) is independent of the underlying DBMS. Thus, the NLIDB can be ported to a different DBMS, by rewriting the database query generator module (see section 6.2). This approach also allows the development of generic linguistic front-ends, that can be used as parts of interfaces to systems other than databases (e.g. expert systems, operating systems etc). One example of such a generic front-end is the CLE system [2].

Another advantage of the particular architecture shown in figure 5, is that the domain-dependent knowledge (i.e. knowledge that has to be modified whenever the system is used in a new knowledge domain – e.g. questions about flights, instead of questions about books in a library) is clearly separated from the rest of the front-end, thus allowing knowledge-domain portability (see section 6.1).

Finally, NLIDBs based on logical intermediate representation languages, allow reasoning modules to be added between the semantic interpreter and the database query generator, so that the NLIDB can carry out reasoning based on the information stored in the database (see section 8). In systems where the parse tree is mapped directly to an expression in a database query language (e.g. SQL), such reasoning would have to be carried out in the database query language. This is practically impossible, since database query languages are not designed to facilitate machine reasoning.

5.5 Response generation

The discussion in section 5.4 focused on the interpretation of the natural language input. Generating a suitable response is also important. Some NLIDBs that interface to relational databases simply print the tuples retrieved by the database query, an approach which is not always acceptable:

- The retrieved database tuples may contain encoded information (e.g. department codes instead of department names).
- The system may not be able to "understand" the question, in which case no tuples will be retrieved. The cause of the failure should be explained to the user (e.g. unknown word, syntax too complex, no relevant information in the database, etc).
- The user's question may contain false presuppositions about the database or the world, in which case the system should report the false presuppositions (e.g. *"Do all managers earning more than 50,000 work in sales?"*, when no manager earns more than 50,000).

- The user's request may not express literally what the user wants to know. For example, it may not be acceptable to answer a *"yes"/"no"* question (e.g. *"Is there a flight to Athens?"*) by a yes/no response.

To illustrate some of these points, consider the following dialogue between the user and LOQUI, based on examples from [14] and [15]:

> *Does every person that works on 20 projects work on* HOSCOM*?*

`There is no such person.`

> *Does David Sedlock work on something?*

`Yes.`

BIM_LOQUI, MMI2, NLPAD

> *Does RV work on* LOQUI*?*

`No. RV works on` BIM-PROLOG.

In the first question, the system has detected the false presupposition that there are people working on 20 projects, and it has reported a suitable message. (Strictly logically speaking the response could have been "`yes`", since no person works on 20 projects.) In the second and third questions, the system did not simply print *"yes"/"no"* responses; it also printed additional information, that the user probably wanted to know.

In some cases, such "cooperative responses" (see [64] [65]) can be generated by using relatively simple mechanisms. For example, in yes/no questions like *"Does David Sedlock work on something?"*, if the answer is affirmative one strategy is to print a "`yes`", followed by the answer to *"On what does David Sedlock work?"*. (It is usually not difficult to generate automatically questions like *"On what does David Sedlock work?"* from questions like *"Does David Sedlock work on something?"*.) A description of some relatively simple mechanisms that have been used in MASQUE to generate cooperative responses can be found in [83], and [41] gives a more formal account of some of these techniques.

It is not always possible, however, to generate reasonable cooperative responses by using simple mechanisms. In some cases, in order to generate an acceptable cooperative response, the NLIDB must be able to reason about the user's goals. This is discussed in section 8.2.

Another problem is that the NLIDB may misinterpret a question submitted by the user, without the user becoming aware that his/her query has been misinterpreted. Natural language questions often have several readings; the NLIDB may select a reading of a question that is different from the reading the user had in mind when he/she typed the question. In these cases, it may be hard for the user to understand that the system has actually answered a different question. To avoid such misunderstandings, TQA [31] contains a module that converts the SQL query back to natural language. The reconstructed natural language request is presented to the user, to ensure that none of the intermediate transformation stages has caused his/her request to be misinterpreted. Similar paraphrasing modules are used in several other NLIDBs. [35] [67] describes how the paraphrasing module of a particular NLIDB works.

To make the retrieved information easier to read, many NLIDBs contain response formatting modules, that allow encoded information retrieved from the database (e.g. "`dpt341`") to be expanded to more natural formats (e.g. "`Sales department`"). Finally, some NLIDBs (e.g. NATURAL LANGUAGE [75]) can present some kinds of retrieved data in graphics (e.g. pie-charts).

5.6 Multiple underlying systems

The architectures described so far, allow the user to interact with a single database. In some cases, however, access to more than one underlying applications may be necessary.

In certain domains, the information needed to answer the user's question may be distributed in several, possibly heterogeneous, databases. (See [22] for an introduction to distributed and heterogeneous databases.) Some NLIDBs (e.g. INTELLECT according to [96]) allow the user to access transparently multiple databases through a single English question. The NLIDB translates the logic query into a set of database queries, each one expressed in the query language of the corresponding DBMS. The NLIDB then collects and joins the partial results retrieved by each DBMS, and reports the answer to the user.

INTELLECT can be configured to interact with the database through the KBMS [97] expert system shell. This way a reasoning layer can be added between the NLIDB and the database, which allows the NLIDB to carry out reasoning based on the data stored in the database (see section 8).

ASK [93] [92] can be configured to interface transparently to several kinds of software systems (e.g. electronic mail systems, screen painters, word processors, etc). ASK translates each natural language request into a suitable expression, which is then directed to the appropriate underlying system. This way, the user views an integrated computer system, with natural language understanding abilities.

JANUS [17] has the ability to interact with databases, expert systems, decision support systems, graphics systems, etc., and all the underlying systems may be involved in the answering of a user's question. For example (the example is based on [17]), when asked *"Which submarines have the greatest probability of locating A within 10 hours?"*, the system might use a database to retrieve the current positions of the submarines, and two separate expert systems to compute (a) the navigation course each submarine would have to follow, and (b) the probability of success for each submarine.

6 Portability

Early NLIDBs were each designed for a particular database application. LUNAR [106], for example, was designed to support English questions, referring to a database of a particular structure, holding data about moon rocks. Such application-tailored NLIDBs were very difficult to port to different applications. In recent years a large part of the research in NLIDBs has been devoted to portability, i.e. to the design of NLIDBs that can be used in different knowledge-domains, with different underlying DBMSs, or even with different natural languages. This section discusses the different kinds of NLIDB portability, and presents some methods that have been used to approach the goal of portability. Most of the discussion relates to the architecture of figure 5.

6.1 Knowledge-domain portability

Existing NLIDBs can only cope with questions referring to a particular *knowledge domain* (e.g. questions about train services, questions about bank accounts). A NLIDB provides knowledge-domain portability, if it can be configured for use in a wide variety of knowledge domains.

Typically, whenever a NLIDB is being reconfigured for a new knowledge-domain, someone has to "teach" the system the words and concepts used in the new domain, and how these concepts relate to the information stored in the database. In systems adopting architectures similar to the one of figure 5, this "teaching" modifies the lexicon, the world model, and the mapping to the database.

Different NLIDBs assume that the knowledge-domain configuration will be carried out by people possessing different skills:

Programmer: In some systems a part (usually small and well defined) of the NLIDB's code has to be rewritten during the knowledge domain configuration. It is, therefore, assumed that the person that will carry out the configuration will be a programmer, preferably familiar with the NLIDB's code.

Knowledge engineer: Other NLIDBs provide tools that can be used to configure the system for a new knowledge domain, without the need to do any programming. It is still assumed, however, that the tools will be used by a knowledge engineer, or at least a person familiar with basic knowledge representation techniques, databases, and linguistic concepts.

In the following dialogue, between MASQUE's "domain editor" [7] and the knowledge engineer, the verb *"to exceed"* (as in *"The population of Germany exceeds the population of Portugal."*) is added to the system's knowledge. (Actually, MASQUE's domain-editor does not shield completely the knowledge-engineer from the programming language. The knowledge engineer is still required to write some Prolog code to map logic predicates to database objects.)

```
editor> add verb
what is your verb ? exceed
what is its third sing. pres ? exceeds
what is its past form ? exceeded
what is its perfect form ? exceeded
what is its participle form ? exceeding
to what set does the subject belong ? numeric
is there a direct object ? yes
to what set does it belong ? numeric
is there an indirect object ? no
is it linked to a complement ? no
what is its predicate ? greater_than
do you really wish to add this verb? y
```

The logic expression of *"to exceed"* was defined to be $greater\_than(arg_1, arg_2)$, where both arguments were declared to belong to the hierarchy class *numeric* (see section 5.4).

Some systems (e.g. PARLANCE [12], CLE [2]) provide morphology modules, that allow the system to determine the various forms of the words using stems stored in the lexicon and morphology rules. This way, the user does not have to specify all the word forms as in the example above (*"exceed"*, *"exceeds"*, *"exceeded"*, *"exceeding"*, etc). This ability is particularly important in highly-inflected languages, where each word may have numerous possible forms. Some systems (e.g. NATURAL LANGUAGE [75], CLE [2]) have built-in dictionaries, listing the most common words. The dictionaries can be customised by the person configuring the system to include domain-specific terminology.

DB administrator: According to the designers of TEAM [47] [48], a realistic scenario is to assume that the NLIDB will be bought by a company already using a database to which the NLIDB is to be linked. In such situations, the responsibility of installing and configuring the NLIDB would probably be assigned to the local database administrator. Therefore, the configuration tools should be designed, so that they can be used by persons with a good understanding of database concepts, familiar with the particular database to which the NLIDB is to be linked, but with no knowledge of AI, logics, or linguistics. TEAM tries to infer what questions can be asked about the database, by eliciting from the database administrator information about the use and structure of the database.

In the following dialogue (from [47] – slightly simplified), the system collects information about the structure and use of the database table (“`file`”) `CHIP`. (The reader is reminded that messages generated by the system are shown in `this typeface`, and user entries are shown in *this typeface*.)

`file name: CHIP`
`fields:` *ID MAKER WIDTH ...*
`subject:` *processor*
`synonyms for processor:` *chip*
`primary key:` *ID*
`Can one say ‘‘Who are the processors?’:` *no*
`Pronouns for subject (he, she, it,they):` *it*
`field: MAKER`
`type of field (symbolic, arithmetic, feature):` *symbolic*
...

The user is asked to provide information about the table’s fields (attributes), the subjects the table describes (*“processor”*), etc. The dialogue would then continue with the system asking several questions for each field of `CHIP`. The reader will have noticed that the system’s questions do not refer to logic concepts (e.g. hierarchy classes, predicates) or advanced linguistic concepts, although they contain database terminology (e.g. “`primary key`”).

End-user: Other NLIDB designers emphasise that the configuration phase can never be complete, and that the NLIDB will always have to learn new words and concepts. Therefore, the NLIDB should allow the end-users to teach the system on-line, and using definitions expressed in natural language. Systems adopting this approach usually have a large built-in vocabulary, and a set of primitive concepts (meanings). The end-user can extend the system’s capabilities by teaching it new words, and by defining the meanings of the new words in terms of primitive concepts or in terms of words the system already knows.

The following examples from [12] show some of the capabilities of PARLANCE:

\> *define: a yuppie is a person under 30 with a graduate degree who earns over 5K per month*
\> *define: a rich employee is an employee who earns over 100K per year*

6.2 DBMS portability

A NLIDB provides DBMS portability, if it can be easily modified to be used with different underlying database management systems (DBMSs).

In the case of NLIDBs that generate queries in widely-supported database query languages (e.g. SQL), it may be possible to port the NLIDB to another DBMS, with only minor modifications, if the new DBMS supports the same database query language as the old one.

If the original database query language is not supported by the new DBMS, then a NLIDB based on the architecture of figure 5 can be ported to the new DBMS by rewriting the module that translates the intermediate logic queries to database queries. In contrast, NLIDBs that do not use intermediate representation languages may require extensive modifications before they can be used with a new DBMS that does not support the old database query language.

In the case of NLIDBs that clearly separate the linguistic front-end (see figure 5) from the rest of the system, it should be possible to port a NLIDB configured for a particular knowledge-domain to a different underlying DBMS containing data about the same knowledge-domain, without modifying the modules of the linguistic front-end.

6.3 Natural language portability

The largest part of the research that has been carried out in the area of NLIDBs assumes that the natural language requests will be written in English. Modifying an existing NLIDB to be used with a different natural language can be a difficult task, since the NLIDB may contain deeply embedded assumptions about the natural language to be used.

In NLIDBs based on the architecture of figure 5, modifying the NLIDB for a different natural language typically requires rewriting/modifying the syntax rules, the semantic rules, and the lexicon. An attempt to build a Portuguese version of CHAT-80 [101] is described in [66]. Information about a Swedish version of CLE can be found in [2].

6.4 Hardware and programming language portability

Up to the early eighties, the ability to port the NLIDB to different hardware and software environments were also important aspects of NLIDBs. Some NLIDBs could be used only on expensive research computing systems, supporting "exotic" programming languages (e.g. Lisp machines), thus making NLIDBs almost impossible to use in real-life applications. In recent years the availability of powerful low-cost computers and the fact that "AI" languages like Prolog and Lisp are now commonly available have allowed NLIDBs to become easy to port across computing platforms.

7 Restricted NL input

Current NLIDBs support only a limited subset of natural language. The limits of this subset are usually not obvious to the user (see section 3). Users are often forced to rephrase their questions, until a form the system can understand is reached. In other cases, users never try questions the NLIDB could in principle handle, because they think the questions are outside the subset of natural language supported by the NLIDB.

Many research projects attempt to expand the linguistic coverage of NLIDBs, hoping that eventually NLIDBs will be able to understand most user requests, at least in particular knowledge domains. An alternative approach, presented in this section, is to deliberately and explicitly restrict the set of natural language expressions the user is allowed to input, so that the limits of this subset are obvious to the user. By choosing carefully the subset of the allowed natural language expressions, the design and implementation of the NLIDB can also be simplified.

This section presents two approaches that drastically restrict the set of natural language expressions the user is allowed to input. In the first one, only questions of a very specific syntactic pattern are allowed; in the second one, the user has to compose his/her queries by choosing words from limited lists of words displayed on the screen.

7.1 Restricted NL syntax

In PRE [40] the user is allowed to input only questions of a specific syntactic pattern. All questions must have the form:

what
conjoined noun phrases
    nested relative clauses
        conjoined relative clauses

The following question (from [40] – simplified) is an example of a question acceptable by PRE:

what are	
the names, ids, and categories of the employees	(1)
who are assigned schedules	(2)
that include appointments	(3)
that are executions of orders	(4)
whose addresses contain 'maple' and	(5)
whose dates are later than 12/15/83 and	(6)
whose statuses are other than 'comp'	(7)

In the example above, line (1) contains the conjoined noun phrases. Lines (2)–(4) contain the nested relative clauses. Notice that these clauses are nested, because (3) refers to *"schedules"* in (2), and (4) refers to *"appointments"* in (3). Lines (5)–(7) contain the conjoined relative clauses. The clauses in (5)–(7) are not nested; they all refer to *"orders"* in (4).
In contrast, the following question is not acceptable by PRE:

what are	
the addresses of the appointments	(1)
that are included in schedules	(2)
whose call times are before 11:30 and	(3)
that are executions of orders	(4)
whose statuses are other than 'comp'	(5)

In the latter example, line (2) contains the only clause of the *nested relative clauses* part. Both (3) and (4) are conjoined relative clauses referring to *"schedules"* in (2). However, (5) is nested with respect to (4), since it refers to *"orders"* in (4), not to *"schedules"* in (2). The PRE question pattern does not allow the *conjoined relative clauses* part to be followed by another *nested relative clauses* part, and hence the question is ill-formed.

[40] claims that the PRE question pattern allows the users to have a clear view of the questions the system is able to answer, and that the pattern is easy to remember. The PRE question pattern is designed to facilitate the mapping from user questions to navigation graphs

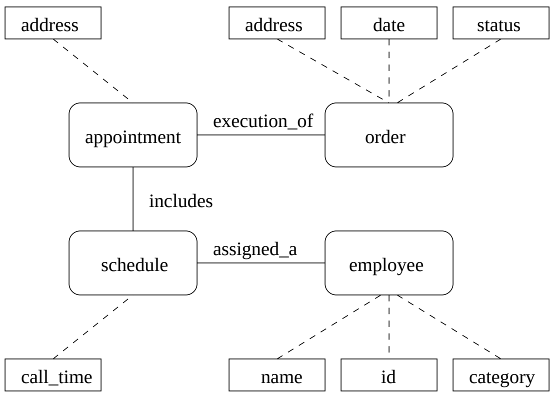

Figure 7: A Pre-like database schema

for Pre's entity-relationship-like [98] database model. Figure 7 shows the database schema to which the above two examples refer. (The schema of figure 7 may not have exactly the same form as the schemata used in Pre.) Oval boxes correspond to entity classes, rectangular boxes correspond to attributes, and continuous lines correspond to relations.

- The *conjoined noun phrases* part of each Pre question corresponds to a projection operator, similar to the ones used in relational algebra [98]. Projection operators determine the attributes of an entity to be reported.
- The *nested relative clauses* part corresponds to traversals of continuous line links in the Pre model.
- The *conjoined relative clauses* part corresponds to select operators, similar to the ones used in relational algebra. A select operator selects entities from an entity class, according to some criteria.

To answer the question of the first example in this section, Pre first transforms the *conjoined relative clauses* part of the question to a select operator, that selects all entities from class `order`, containing *"maple"* in their `address`es, having `date`s later than 12/15/83, and having `status`es other than *"comp"*.

Next, Pre transforms the *nested relative clauses* part of the question to a sequence of relation (continuous line) traversals. In the case of our example, the nested relative clauses are mapped to a sequence of traversals, mapping `order`s to `appointment`s via the `execution_of` relation, `appointment`s to `schedule`s via the `includes` relation, and `schedule`s to `employee`s via the `assigned_a` relation.

Finally, Pre transforms the *conjoined noun phrases* part of the question to a projection operator. For each `employee` entity reached during the previous step, the system reports its `name`, `id`, and `category`.

7.2 Menu-based systems

In menu-based systems the user is not able to type directly his/her questions. Instead, each question has to be constructed by choosing possible words or phrases from menus. The menus

commands	***nouns***	***modifiers***
Find Delete	suppliers	whose part city is
Insert	parts	whose color is
attributes	shipments	whose part name is
weight	<specific suppliers>	whose part# is
quantity	<specific parts>	whose supplier city is
city	<specific shipments>	whose supplier name is
color	***comparisons***	whose supplier supplier# is
name	between	whose shipment part# is
part#	greater than	whose shipment supplier# is
supplier#	less than	whose supplier status is
status	greater than or equal to	whose part weight is
	less than or equal to	whose shipment quantity is
	equal to	which are shipments of
	connectors	which were shipped by
	the number of of	who ship
	the average and	who supply
	the total or	which are supplied by
>		

Figure 8: The initial screen of NLMENU

are displayed on the screen, and they are continuously updated. This method, used in the NLMENU system [90] [91], will be illustrated by examples from [91]. Q&A (section 2) provides a similar menu-based interface.

NLMENU first displays a screen like the one shown in figure 8. (The actual NLMENU screens are slightly more complicated.) Highlighted borders indicate menus that are currently active. The user first selects *"Find"*. (NLMENU also has update capabilities – see section 9.1 – which will not be discussed in this paper.) *"Find"* becomes the first word of the user's question, and it is displayed in the lower part of the screen. After *"Find"* has been selected, the `attributes`, `nouns`, and `connectors` menus become activated. The user selects *"color"* from the `attributes` menu, and the question becomes *"Find color"*. In a similar manner, the user then selects *"and"*, *"name"*, *"of"*, and *"parts"*. At this stage the NLMENU screen has become as shown in figure 9. (After selecting each word, the contents of the menus are modified.)

The user selects *"whose color is"* from the `modifiers` menu, and the question becomes *"Find color and name of parts whose color is"*. At this point, a special menu appears on screen, listing the possible colors. After selecting a color (e.g. *"red"*), the question is complete. The user executes the query by selecting an option on the screen, and the system reports the results retrieved from the database.

NLMENU uses a context-free semantic grammar (see section 5.3). Whenever a new word or phrase is selected, the system parses the sentence assembled so far, and processes the grammar to determine the words/phrases that can be attached to the existing partial sentence to form a sentence the system can understand. These words/phrases are then shown in the active menus. This way, only sentences the system can understand can be input, and the user can browse through the menus to explore the kinds of questions the system can handle.

Although a new semantic grammar is needed for each new knowledge domain (see section 6), NLMENU grammars do not have to cope with a large number of possible user sentences. The menus can be used to greatly restrict the kinds of sentences the user can input, and

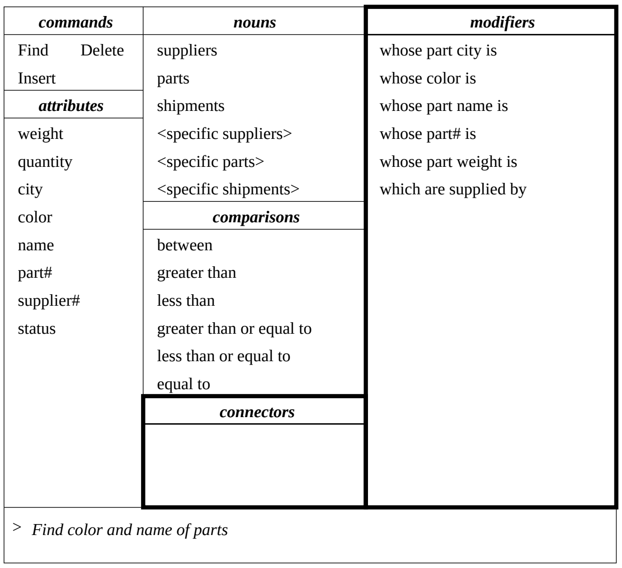

Figure 9: The NLMENU screen after several more words have been selected

hence the semantic grammars of NLMENU are relatively simple to write. According to [91], NLMENU is typically customised for a new application in 1 – 30 man-hours.

The menu-based approach guarantees that the user's input is free from spelling errors, and allows user queries to be input using a simple pointing device. However, as mentioned in [91], in applications requiring long questions menu-searching can become tedious.

8 NLIDBs as intelligent agents

Most NLIDBs described so far are direct interfaces to the underlying database, in the sense that they simply translate user questions to suitable database queries. The database is seen as the only source of information about the world. (The NLIDB itself only contains a simplistic world-model – section 5.4). This section will discuss NLIDBs that contain knowledge about the surrounding world and about the user. This knowledge allows them to handle questions that cannot be answered directly by the DBMS, and to understand better the user's goals.

8.1 Reasoning about the world

In some cases it may not be possible to answer a natural language question, although all the necessary raw data are present in the database. Questions involving common sense or domain expertise are typical examples. In these cases, the answer to the question is not explicitly stored in the database; to produce the answer, the NLIDB must be able to carry out reasoning based on the data stored in the database.

Let us assume, for example, that the database holds the medical records af all patients in a hospital (e.g. the names of the patients, their ages, diagnoses made for each patient, treatments the patients have received, etc.), and let us assume that the NLIDB is asked *"Which patients are likely to develop lung disease?"*. Let us also assume that the database

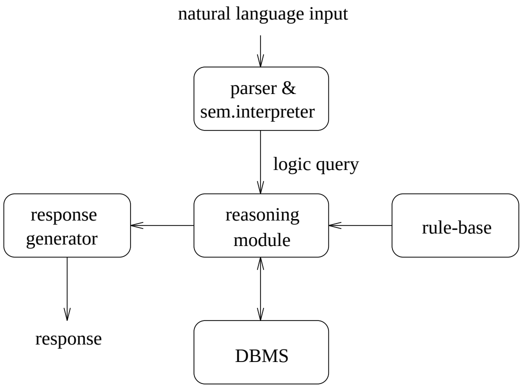

Figure 10: Architecture of a NLIDB with a reasoning module

does not show explicitly how probable it is for each patient to develop each disease (i.e. these probabilities are not stored in the database). In this case the NLIDB will not be able to answer the question, unless it is told how to use the medical data in the database to determine how probable it is for each patient to develop lung disease. In other words, the NLIDB must have access to domain expertise (in this case medical expertise, perhaps in the form of rules), showing how to carry out reasoning based on the raw data in the database.

The architecture of figure 5 can be extended to allow the NLIDB to carry out reasoning based on the data in the database. One possible extension, exemplified by LOQUI [15], is to add a rule-based reasoning module on top of the database as shown in figure 10.

Systems adopting the architecture of figure 10 still transform the natural language question into a logic query that captures the meaning of the question, as discussed in section 5.4. When asked *"Which patients are likely to develop lung disease?"*, such a system could generate the logic query (variables start with capital letters):

```
report Patient such that:
   is_patient(Patient) and
   likely_to_develop(Patient, lung_disease)
```

Some of the predicates in the logic queries may be linked directly to tables in the database, as discussed in section 5.4. For example, *is_patient*(*Patient*) could be linked to a database table *patients* listing all the patients in the hospital. Thus, *is_patient*(*Patient*) would be true if *patients* showed that *Patient* is a patient in the hospital.

For predicates not directly linked to database tables, suitable rules must be present in the rule-base, showing how to evaluate instances of these predicates. Continuing with the same example, the rule-base could contain the rule:

```
likely_to_develop(Patient, Disease) if
  prob(develop(Patient, Disease)) > 0.7
```

The rule above says that a patient is likely to develop a disease if the probability of the patient developing the disease is greater than 0.7

Other rules would show how to use the data in the database to assess the probability of a patient developing a disease. For example:

```
if has(Patient, blood_disease) then
  prob(develop(Patient, lung_disease)) = 0.6

if is_smoker(Patient) then
  prob(develop(Patient, lung_disease)) = 0.7
```

Finally, other rules would show how to combine the probabilities predicted by the rules above, in cases where more than one rules apply (e.g. patient with blood disease who is also a smoker). (See [45] for an introduction to rule-based reasoning with probabilities.)

In Loqui, according to [14], both the logic queries and the rules of the rule-base are expressed in Prolog. The logic query is treated as a Prolog goal to be evaluated by the Prolog interpreter, and rows in the database tables are treated as Prolog facts. (See for example [70] [21] [20] [39] for information on how Prolog can be linked to external databases.) Thus, the full power of Prolog is available when evaluating a query, or when defining the rules in the rule-base, and the internal Prolog database can be used to hold knowledge about the world that cannot be easily encoded in the external relational database. Chat-80 [101] and Masque [6] [7] offer similar capabilities.

Intellect, according to [96], can be configured to use the Kbms [97] expert system shell as its reasoning module. The logic query is passed to Kbms, and it is the responsibility of Kbms to find all the possible answers using its specialised inferencing techniques and the facts stored in the external database. This way, the full power of the expert system shell (e.g. forward chaining, backward chaining, hypothetical reasoning, etc.) is available when evaluating the logic query, and the shell's tools can be used to develop the rule-base. The natural language capabilities of Intellect can be used to define Kbms rules. For example [97], a Kbms rule can be entered as "IF *the customer has an overdue balance outstanding,* Then *credit risk is high*". Natural language questions can also query the shell's reasoning. For example, *"Why was the application of Smith rejected?"* would cause the system to report the steps it took to reject Smith's application.

Perhaps the reasoning module should not be considered part of the Nlidb, but rather an independent layer on top of the database system (or a component of an "expert database" system). A strong argument in favour of this view is that the ability to carry out reasoning on top of the database is not needed only in Nlidbs. A graphical interface would also need to access the reasoning module in order to answer a graphical query corresponding to *"Which patients are likely to develop lung disease?"*. The reasoning module should not, therefore, be part of the Nlidb's code. It should be an independent module, available to any application that may need it.

8.2 Reasoning about the user's goals

User requests often do not express literally what the user wants to know. Let us consider the following dialogue, based on an example from [63]:

> *Do American Airlines have a night flight to Dallas?*

`yes`

It would be better if the system could respond:

`Yes, flight 712.`

or:

`No, but United have flight 655.`

In the case of the last two responses, instead of simply printing a yes/no response, the NLIDB has also reported additional information that the user would probably be interested to know. Notice that in the last response an alternative United flight was reported, even though the user had requested an American Airlines flight.

As already mentioned in section 5.5, in some cases reasonably cooperative responses can be generated by using relatively simple mechanisms. For example, in yes/no questions like *"Do American Airlines have a night flight to Dallas?"*, if the answer is negative, the NLIDB could report a "`no`", along with the answer to a "weaker" question like *"Does a carrier have a night flight to Dallas?"*. The weaker question is generated from the original one by removing one restriction (the restriction that the carrier must be American Airlines in our example). (A similar but more elaborate method is proposed in [65].)

Unfortunately, simple strategies, like dropping an arbitrary restriction, do not always generate acceptable responses. As pointed out in [63], the previous dialogue could have been:

> *Do American Airlines have a night flight to Dallas?*

`No, but they have one to Miami.`

In the latter dialogue the system has dropped the restriction that the flight must go to Dallas, rather than the restriction that the carrier must be American Airlines. The resulting "cooperative" response would probably be irritating for the user. To avoid such inappropriate responses, the NLIDB must be able to "understand" the user's goals. (This is often called "plan recognition" in the literature – see for example [1].) In the previous question, we would like the NLIDB to be able to understand that the user's goal is to go to Dallas, and that therefore it would be inappropriate to report a flight to Miami.

This could be achieved by having a rule stating that when a person asks about a flight to a particular location, the goal of the person is probably to reach that location. Another rule could state that if the person has specified a particular carrier, then flying with that carrier is probably a preference but not a goal. Finally, another rule could state that if it is not possible to satisfy all goals and preferences, then preferences, but not goals, can be dropped. Hence, in *"Do American Airlines have a night flight to Dallas?"*, if there is no American Airlines night flight to Dallas, the system would drop the carrier constraint, and not the destination constraint.

The reasoning about the user's goals could be carried out by reasoning modules similar to the ones described in the previous section (figure 10). In this case, however, the rule-base would contain rules explaining how the user's requests relate to his/her goals, and what to do to satisfy the user's goals. The user's request would be mapped to a logic query, and the logic query would be passed to the reasoning module. The reasoning module would then try to satisfy the user by reasoning about his/her goals, and by retrieving information from the database.

The rules in the rule-base can be thought as constituting a *model of the user*, a logic system describing how user utterances relate to the user's intentions and beliefs. The need for elaborate user models becomes essential in dialogue systems, discussed in the following section.

8.3 Dialogue systems

The architectures discussed so far focus on the understanding of individual user questions. The parsing and semantic interpretation modules form the backbone of the system, while

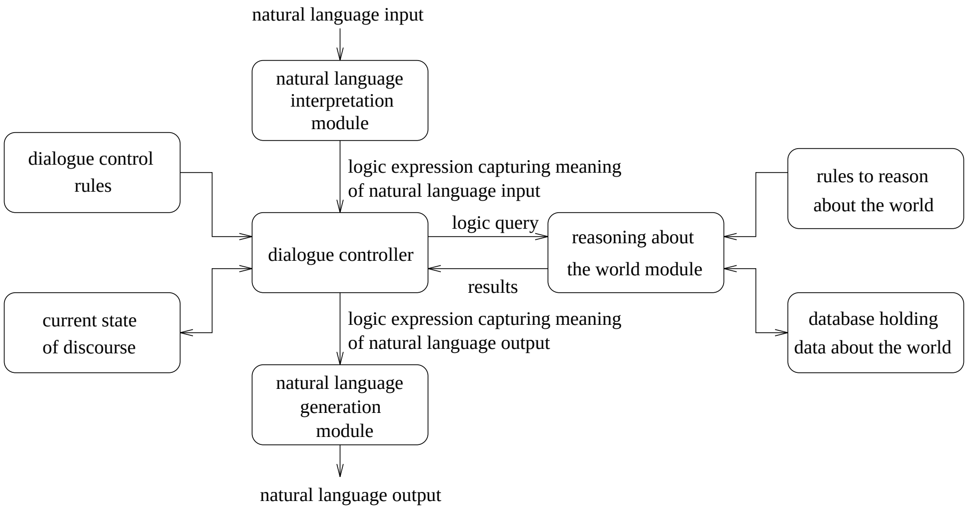

Figure 11: Possible architecture of a dialogue-oriented system

reasoning and response generation modules are treated as peripheral accessories. In other words, it is the processing of each individual question that drives the system's behaviour.

In dialogue-oriented systems the system's behaviour is driven by a reasoning module, which reasons about the goals and beliefs of both the user and the system. A possible architecture of a dialogue-oriented system is shown in figure 11. (The architecture shown is greatly influenced by the IR-NLI architecture described in [49].)

The central module of the architecture in figure 11 is the *dialogue controller*. The dialogue controller is a reasoning module, whose purpose is to understand the user's needs and to convey the relevant information to the user. The dialogue controller consults a rule-base, that contains rules explaining how to carry out a dialogue. For example, the following is a simplified dialogue-control rule for WISBER [44], a natural language consultation system:

$$
\begin{array}{lcl}
(want\ system & & (know\ mutual \\
\quad (know\ system\ X)) \wedge & (ask & \quad (want\ system \\
(believe\ system & system & \quad\quad (know\ system\ X))) \wedge \\
\quad (know\ user\ (related\ X))) \wedge & \Rightarrow\ user\ \Rightarrow & (know\ mutual \\
\neg(know\ system & X) & \quad (believe\ system \\
\quad \neg(know\ user X)) & & \quad\quad (know\ user\ (related\ X))))
\end{array}
$$

The rule says that if:

- the system wants to know a piece of knowledge, called X, and
- the system believes that the user knows something related to X, and
- the system does not know that the user does not know X

then, the system should ask the user about X, and after the question has been asked, the system should assume that:

- both the system and the user know that the system wants to learn X, and

- both the system and the user know that the system believes the user knows something related to X.

In figure 11 the *current state of discourse* contains facts describing the current goals and beliefs of the system, and facts describing what the system currently understands to be the goals and beliefs of the user.

Both the *natural language interpretation* and *natural language generation* modules are subordinate to the dialogue controller. Whenever the dialogue controller wishes to communicate information to the user (e.g. ask the user a question, report retrieved information), it formulates a logic expression that captures the meaning of the message to be generated. The logic expression is then passed to the natural language generation module, which transforms it into a natural language message. The natural language interpretation module is then used to transform the user's response to a logic expression, and this logic expression is passed to the dialogue controller.

Finally, whenever the dialogue controller wishes to retrieve information about the world, it formulates a logic query which is passed to the *reasoning about the world* module. The latter uses the *rules to reason about the world* to carry out reasoning based on the data stored in the external database (as discussed in section 8.1).

In dialogue-oriented systems, the generation of the system's messages (see section 5.5) is equally important to the interpretation of the user's input. The system's messages must be carefully chosen, to guide the dialogue towards areas that will clarify the user's goals and beliefs. The system's messages must also reflect correctly its understanding and its capabilities.

9 Further Topics

This section highlights some areas of NLIDB research that have been less explored, namely database updates, meta-knowledge questions, temporal questions, and multi-modal interfaces.

9.1 Database updates

Although in several NLIDBs the user is allowed to update the interface's linguistic knowledge using natural language requests (section 6.1), few systems support natural language requests to update the underlying database. Database updates by natural language introduce several new problems, which will be illustrated in this section.

In ASK [93] [92] update requests can appear at any point during the dialogue with the system. The following dialogue is based on examples from [93] [92]:

> *What is the home port of the Tokyo Maru?*
`Yokohama`
> *Home port of the Tokyo Maru is Hong Kong.*
`Yokohama has been replaced by Hong Kong as home port of Tokyo Maru.`
> *The cargo of the Tokyo Maru is coal.*
`Coal has been added as the cargo of`
`Tokyo Maru.`
> *What is the cargo of the Tokyo Maru?*
`Grain`
`Coal`

ASK knows that each ship may only have one home port. Therefore, when informed that *"Home port of the Tokyo Maru is Hong Kong."*, it replaces *"Yokohama"* (previously known to be the home port of the Tokyo Maru) by *"Hong Kong"*. In contrast, a ship may carry more than one kinds of cargo. Thus, when informed that *"The cargo of the Tokyo Maru is coal."*, ASK adds *"Coal"* to the kinds of cargo already known to be carried by the Tokyo Maru (*"Grain"*).

Unfortunately, satisfying a natural language update request is not always that straightforward. As explained in [32], natural language update requests often cannot be satisfied, or they are ambiguous, or they lead to unanticipated side-effects, due to database constraints the user is not aware of. Consider a database containing the following tables (the example is borrowed from [32]):

employees_table

employee	salary	department
Adams	3000	inventory
White	3500	marketing
Brown	2500	sales
Smith	2500	sales

departments_table

department	manager
sales	Jones
marketing	Baker
inventory	Fisher

The following dialogue takes place:

> *List the employees and their managers.*

`employee`	`manager`
`Adams`	`Fisher`
`White`	`Baker`
`Brown`	`Jones`
`Smith`	`Jones`

> *Change Brown's manager from Jones to Baker.*

`Done.`

> *List the employees and their managers.*

`employee`	`manager`
`Adams`	`Fisher`
`White`	`Baker`
`Brown`	`Baker`
`Smith`	`Baker`

Notice that Smith's manager has also changed from Jones to Baker, although this was not requested. To a user not aware of the database's structure, such a behaviour would appear erratic. The user does not know that in the database employees are linked to managers through their departments. Since Brown works in the sales department, the system has changed the sales manager from Jones to Baker, which has also caused other employees in the sales department (e.g. Smith) to get a new manager.

Because of the indirect link between employees and managers, the update request is actually ambiguous. Instead of changing the sales manager from Jones to Baker, the system could have moved Smith from the sales department (managed by Jones) to the marketing department (managed by Baker).

PIQUET [32] maintains a connection graph, which models the user's view of the database structure. Whenever a user utterance reflects awareness or presupposition of a database object, link, restriction etc., the graph is modified accordingly. Every time an ambiguous update request is encountered, the system consults the connection graph to determine which of the possible interpretations of the update request are meaningful with respect to the user's current view of the underlying database. It then ranks the remaining candidate interpretations, using a set of heuristic rules. For example, interpretations causing fewer side-effects with respect to the user's view are preferred. Returning to the dialogue above, after *"List the employees and their managers."* has been answered, the user's view of the database is equivalent to a table of the form:

employee	manager
Adams	Fisher
White	Baker
Brown	Jones
Smith	Jones

If the system interprets the request *"Change Brown's manager from Jones to Baker."* as a request to change the sales manager from Jones to Baker, then the user's view of the database becomes as shown below on the left. (Side-effects are shown in **this typeface**.)

employee	manager
Adams	Fisher
White	Baker
Brown	Baker
Smith	**Baker**

employee	manager
Adams	Fisher
White	Baker
Brown	Baker
Smith	Jones

If the system interprets the request as a request to move Brown from the sales department (managed by Jones) to the marketing department (managed by Baker), then the user's view of the database becomes as shown above on the right. The second interpretation causes no side-effects to the user's view, and thus it would have been preferred by PIQUET's heuristics.

9.2 Meta-knowledge questions

Meta-knowledge questions are questions referring to knowledge about knowledge. For example, some meta-knowledge questions could be [57] [92]:

> *What information is in the database?*
> *What is known about ships?*
> *What are the possible employee job titles?*
> *Can you handle relative clauses?*

The following example [92] shows how ASK reacts to one meta-knowledge question:

> *What is known about ships?*

```
Some are in the following classes: navy, freighter, tanker
All have the following attributes: destination, home port
...
```

An interesting kind of meta-knowledge questions are *modal questions*. In modal questions the user asks whether something can or must be the case. For example:

> *Can a female employee work in sales?*
> *Can an employee earn more than his manager?*
> *Is it true that all candidates must be over 20?*

In [68] the authors present a method for handling modal questions in NLIDBs to relational databases. This method is used in the NLIDB developed at Essex University [34] [33] [69].

9.3 Temporal questions

Most NLIDBs were designed to interface to snapshot databases. Snapshot databases only store information about one state of the world, usually taken to be the "present" state. Consequently, most NLIDBs only support questions referring to the present state of the world. For example, a NLIDB to a company's database would typically be able to answer questions like:

> *Who is the manager of the sales department?*
> *What is the maximum manager salary?*

but would usually not support temporal questions referring to the past (or the future):

> *Who was the previous manager of the sales department?*
> *What was the maximum manager salary during the last 10 years?*
> *While Smith was sales manager, did the annual sales income ever exceed 1 million?*

Most NLIDBs cannot answer temporal questions because: (a) they cannot cope with the semantics of natural language temporal expressions (e.g. tenses/aspects, temporal subordinators), and (b) they were designed to interface to "snapshot" databases, that do not facilitate the manipulation of time-dependent information.

In the last decade, computer scientists have become increasingly interested in *temporal database systems*, i.e. database systems designed to store information about previous or future states of the world, and to generally support the notion of time. (The reader is referred to [88] for an introduction to temporal databases.) Clifford [24] [25] [23], Hafner [51], and Hinrichs [59] are among those who have considered natural language interfaces to temporal databases.[5]

9.4 Multimodal interfaces

As discussed in section 3, form-based interfaces and graphical interfaces appear to have (at least some) advantages over natural language interfaces. Some systems attempt to merge natural language with graphics, menus, and forms, to combine the strengths of all modalities.

In JANUS [17] the user is allowed to combine text, menus, graphics, and speech when forming his/her queries. A user could, for example, type the question *"Which of those submarines has the greatest probability of locating A within 10 hours?"*, and then use the mouse and a screen displaying the positions of ships to determine the set of ships to which *"those"* refers.

One way to integrate natural language and other modalities is exemplified by SHOPTALK [28]. SHOPTALK is actually a complex manufacturing information and decision support system with multimodal input capabilities, rather than a simple NLIDB. The methods, however, used in SHOPTALK could also be used in simple NLIDBs. The following examples are all borrowed from [28].

SHOPTALK initially displays a set of menus, listing the available kinds of actions and queries (e.g. `move a lot`, `put an object` etc). This way, the user has a clear view of the system's capabilities. When an action/query-type has been selected, SHOPTALK displays a form containing fields that have to be filled in. For example, if the user selects the `move_lots` action, the system displays the form:

[5] The authors are also currently involved in research on natural language interfaces to temporal databases.

move_lots
`which lots:`
`to which station:`
`whenever:`

Each field can be filled using input in the form of any modality supported by Shoptalk. For example, the `which lots` field could be filled by selecting lots from a diagram displayed on screen, or by typing natural language. The form above, could be filled as follows (*<here>* denotes pointing with the mouse):

move_lots	
`which lots:`	*each baked lot*
`to which station:`	*the manual inspection station with the smallest queue*
`whenever:`	*it has been moved <here>*

By allowing both natural language and mouse-pointing, the user can combine the strengths of both modalities. In the case of the `which lots` field, the user has chosen natural language (*"each baked lot"*), which allows universal quantification (*"each"*) to be expressed easily. It would be difficult to express universal quantification using graphics and the mouse: the system would have to display all the baked lots, and the user would have to select all the baked lots from the display using the mouse.

Similarly, natural language allows *"the manual inspection station with the smallest queue"* to be expressed easily and concisely. Without natural language support, the user would probably have to use a formal specification language, or to use numerous nested menus. In contrast, in the case of the `whenever` field, the user has substituted part of the natural language expression by a pointing action on a diagram, thus avoiding the need to provide a lengthy natural language description of the location.

The form-filling approach used in Shoptalk also by-passes some kinds of natural language ambiguity. In the following simplified example from [28],

put	
`object:`	*the block in the box*
`destination:`	*on the table in the corner*

it is clear to the system that *"in the box"* is a modifier of *"the block"*, that *"in the corner"* is a modifier of *"the table"*, and that *"on the table in the corner"* is the description of the destination and not, for example, a modifier of the object to be moved (see section 4.1 for a description of the modifier attachment problem).

After each question, Shoptalk graphically displays objects to which anaphoric expressions (e.g. pronouns – see section 4.5) of follow-up questions may refer. The user can subsequently deactivate some of these possible referents by pointing on their graphical representations, thus reducing the number of possible referents the system has to consider in follow-up questions.

Shoptalk also displays a graphical representation of the discourse structure, allowing the user to specify the context to which follow up questions refer. Figure 12 shows a modified example from [28]. The user has specified that both *"Which of them were baked before lot3?"* and *"Where were they when lot3 was being baked?"* are follow-up questions to *"Which lots*

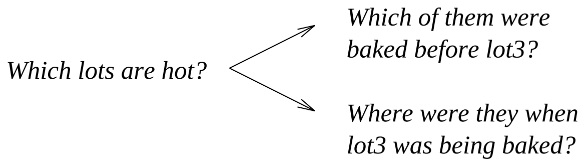

Figure 12: A context graph in SHOPTALK

are hot?". Therefore, the pronoun *"they"* in *"Where were they when lot3 was being baked?"* refers to the hot lots, not to the hot lots that were baked before lot3.

Finally, SHOPTALK supports questions referring to past events or states. The user is allowed to specify the previous time to which a query refers by either dragging a "time-slider", or by filling special `when` fields (see [28] for more details).

10 The state of the art

This paper has attempted to serve two purposes: to introduce the reader to the field of NLIDBs by describing some of the central issues, and to indicate the current situation in this area by outlining the facilities and methods of typical implemented systems. The goal of surveying the field can be achieved only incompletely at any given moment. Nevertheless, our summary here is sufficient to give rise to some general observations about recent work on NLIDBs.

We have not dealt in detail with the evaluation of NLIDBs. This is a complex problem. It is difficult enough to formulate general quantitative (or subtle qualitative) measures of the effectiveness of parsers (i.e. programs which extract syntactic information from sentences); when we have to consider an entire system, involving semantics, querying, response generation, and user interfacing, the situation is much more complicated. Hence, no standard benchmarks have yet been developed (see [78] for some discussion). Under these circumstances, any appraisal of the current state of the field must be impressionistic and subjective.

In order to structure our presentation round issues, we have deliberately overlooked one dimension of distinction which might be held to be important, namely, the extent to which the systems are experimental research vehicles or working commercial applications. The majority of the systems we have referred to are oriented more towards exploration of possible techniques than towards the production of marketable products. Those which seem to fall into the category of experimental prototypes include LUNAR, CHAT-80, MASQUE, TEAM, ASK, EUFID, LDC, TQA, and others, while INTELLECT, PARLANCE, LANGUAGEACCESS, Q&A, NATURAL LANGUAGE, LOQUI, and ENGLISH WIZARD are actual products or systems used in real applications.

In the context of these caveats, the following observations seem to us to be valid:

1. There are several facets of NLIDBs (e.g. dictionaries, parsing) where there has been a great deal of research and development, and where practical system modules can be constructed fairly straightforwardly.

2. Even in the relatively well-understood subareas mentioned above, there is not a single agreed consensus theory or technique. In the other aspects of NLIDBs, there are several

difficult theoretical problems which remain to be solved before NLIDBs will be fully useful and easily constructed.

3. The development of NLIDBs is no longer as fashionable a topic within academic research as it was in the 1980s.

4. Although there are several commercially available systems which purport to allow querying of databases in English, it must be remembered that this claim is flexible in its possible interpretations.[6] These systems may be extremely effective tools for database access, and may use some form of English (or other natural languages). This does not mean that the problem of completely unrestricted use of natural language queries has been solved.

5. Interfaces which solely use keyboard input of natural language are not likely to be hugely practical; in the long term, the use of spoken input, and/or integration with graphical user interfaces, are more likely to be the route to practical success. However, keyboard-based NLIDBs are still a useful testbed for developing the necessary techniques (e.g. semantic interpretation) for subsequent incorporation into these more complex systems.

Acknowledgments

The authors would like to thank Lieve Debille (BIM), Kathleen Pierco (BIM), Madeleine Bates (BBN Systems and Technologies), Barclay Rockwood (Natural Language Inc.), John Menkart (Trinzic), Hilary Quick (IBM UK), and Bruce Bovill (SAS Software Ltd.) for providing information about systems developed by their companies. The authors are also grateful to Chris McNeilly, David Boltz, and John Nerbonne, who provided pointers to the literature and addresses of companies that have developed NLIDBs. Special thanks to John Nerbonne for comments on an earlier draft of this paper.

This paper reports work carried out by the first author under the supervision of the other two authors. The first author is grateful to the Greek State Scholarships Foundation for funding his studies in Edinburgh.

References

[1] J. Allen. Recognizing Intentions from Natural Language Utterances. In M. Brady and R.C. Berwick, editors, *Computational Models of Discourse*, chapter 2, pages 107–166. MIT Press, Cambridge, Massachusetts, 1983.

[2] H. Alshawi. *The Core Language Engine*. MIT Press, Cambridge, Massachusetts, 1992.

[3] H. Alshawi, D. Carter, R. Crouch, S. Pulman, M. Rayner, and A. Smith. CLARE – A Contextual Reasoning and Cooperative Response Framework for the Core Language Engine. Final report, SRI International, December 1992.

[4] I. Androutsopoulos. Interfacing a Natural Language Front-End to a Relational Database (MSc thesis). Technical paper 11, Department of Artificial Intelligence, University of Edinburgh, 1993.

[6]As recently as 1986, a major manufacturer advertised a system which allowed you to program it "in English"; on closer inspection this turned out to be an interpreter for the programming language Basic!

[5] I. Androutsopoulos, G. Ritchie, and P. Thanisch. An Efficient and Portable Natural Language Query Interface for Relational Databases. In P.W. Chung, G. Lovegrove, and M. Ali, editors, *Proceedings of the 6th International Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems, Edinburgh, U.K.*, pages 327–330. Gordon and Breach Publishers Inc., Langhorne, PA, U.S.A., June 1993. ISBN 2–88124–604–4.

[6] P. Auxerre. MASQUE Modular Answering System for Queries in English - Programmer's Manual. Technical Report AIAI/SR/11, Artificial Intelligence Applications Institute, University of Edinburgh, March 1986.

[7] P. Auxerre and R. Inder. MASQUE Modular Answering System for Queries in English - User's Manual. Technical Report AIAI/SR/10, Artificial Intelligence Applications Institute, University of Edinburgh, June 1986.

[8] B. Ballard and D. Stumberger. Semantic Acquisition in TELI. In *Proceedings of the 24th Annual Meeting of ACL, New York*, pages 20–29, 1986.

[9] B.W. Ballard. The Syntax and Semantics of User-Defined Modifiers in a Transportable Natural Language Processor. In *Proceedings of the 22nd Annual Meeting of ACL, Stanford, California*, pages 52–56, 1984.

[10] B.W. Ballard, J.C. Lusth, and N.L. Tinkham. LDC-1: A Transportable, Knowledge-based Natural Language Processor for Office Environments. *ACM Transactions on Office Information Systems*, 2(1):1–25, January 1984.

[11] M. Bates, M.G. Moser, and D. Stallard. The IRUS transportable natural language database interface. In L. Kerschberg, editor, *Expert Database Systems*, pages 617–630. Benjamin/Cummings, Menlo Park, CA., 1986.

[12] BBN Systems and Technologies. *BBN Parlance Interface Software – System Overview*, 1989.

[13] J.E. Bell and L.A. Rowe. An Exploratory Study of Ad Hoc Query Languages to Databases. In *Proceedings of the 8th International Conference on Data Engineering, Tempe, Arizona*, pages 606–613. IEEE Computer Society Press, February 1992.

[14] BIM Information Technology. *Loqui: An Open Natural Query System – General Description*, 1991. (Commercial leaflet).

[15] J.-L. Binot, L. Debille, D. Sedlock, and B. Vandecapelle. Natural Language Interfaces: A New Philosophy. *SunExpert Magazine*, pages 67–73, January 1991.

[16] R.J. Bobrow. The RUS System. In *Research in Natural Language Understanding, BBN Report 3878*. Bolt Beranek and Newman Inc., Cambridge, Massachusetts, 1978.

[17] R.J. Bobrow, P. Resnik, and R.M. Weischedel. Multiple Underlying Systems: Translating User Requests into Programs to Produce Answers. In *Proceedings of the 28th Annual Meeting of ACL, Pittsburgh, Pennsylvania*, pages 227–234, 1990.

[18] R.A. Capindale and R.G. Crawford. Using a Natural Language Interface with Casual Users. *International Journal of Man-Machine Studies*, 32:341–361, 1990.

[19] J.G. Carbonell. Discourse Pragmatics and Ellipsis Resolution in Task-Oriented Natural Language Interfaces. In *Proceedings of the 21st Annual Meeting of ACL, Cambridge, Massachusetts*, pages 164–168, 1983.

[20] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990.

[21] S. Ceri, G. Gottlob, and G. Wiederhold. Efficient Database Access from Prolog. *IEEE Transactions on Software Engineering*, 15(2):153–163, February 1989.

[22] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, New York, 1984.

[23] J. Clifford. Natural Language Querying of Historical Databases. *Computational Linguistics*, 14(4):10–34, December 1988.

[24] J. Clifford. *Formal Semantics and Pragmatics for Natural Language Querying*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, England, 1990.

[25] J. Clifford and D.S. Warren. Formal Semantics for Time in Databases. *ACM Transactions on Database Systems*, 8(2):215–254, June 1983.

[26] E.F. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[27] E.F. Codd. Seven Steps to RENDEZVOUS with the Casual User. In J. Kimbie and K. Koffeman, editors, *Data Base Management*. North-Holland Publishers, 1974.

[28] P.R. Cohen. The Role of Natural Language in a Multimodal Interface. Technical Note 514, Computer Dialogue Laboratory, SRI International, 1991.

[29] A. Copestake and K. Sparck Jones. Natural Language Interfaces to Databases. *The Knowledge Engineering Review*, 5(4):225–249, 1990.

[30] F. Damerau. Operating statistics for the transformational question answering system. *American Journal of Computational Linguistics*, 7:30–42, 1981.

[31] F. Damerau. Problems and Some Solutions in Customization of Natural Language Front Ends. *ACM Transactions on Office Information Systems*, 3(2):165–184, April 1985.

[32] J. Davidson and S.J. Kaplan. Natural Language Access to Data Bases: Interpreting Update Requests. *Computational Linguistics*, 9(2):57–68, April–June 1983.

[33] A.N. De Roeck, C.J. Fox, B.G.T. Lowden, R. Turner, and B.R. Walls. A Natural Language System Based on Formal Semantics. In *Proceedings of the International Conference on Current Issues in Computational Linguistics, Pengang, Malaysia*, 1991.

[34] A.N. De Roeck, H.E. Jowsey, B.G.T. Lowden, R. Turner, and B.R. Walls. A Natural Language Front End to Relational Systems Based on Formal Semantics. In *Proceedings of InfoJapan '90, Tokyo, Japan*, 1990.

[35] A.N. De Roeck and B.G.T. Lowden. Generating English Paraphrases from Formal Relational Calculus Expressions. In *Proceedings of the 11th International Conference on Computational Linguistics, Bonn, Germany*, August 1986.

[36] S.M. Dekleva. Is Natural Language Querying Practical? *Data Base*, pages 24–36, May 1994.

[37] D. Diaper. Identifying the Knowledge Requirements of an Expert System's Natural Language Processing Interface. In M.D. Harrison and A.F. Monk, editors, *People and Computers: Designing for Usability – Proceedings of the Second Conference of the British Computer Society, Human Computer Interaction Specialist Group, University of York*, pages 263–280. Cambridge University Press, September 1986.

[38] D.R. Dowty, R.E. Wall, and S. Peters. *Introduction to Montague Semantics.* D.Reidel Publishing Company, Dordrecht, Holland, 1981.

[39] C. Draxler. *Accessing Relational and Higher Databases Through Database Set Predicates in Logic Programming Languages.* PhD thesis, University of Zurich, 1992.

[40] S.S. Epstein. Transportable Natural Language Processing Through Simplicity – the PRE System. *ACM Transactions on Office Information Systems*, 3(2):107–120, April 1985.

[41] A. Gal, G. Lapalme, P. Saint-Dizier, and H. Somers. *Prolog for Natural Language Processing.* Wiley, Chichester, England., 1991.

[42] G. Gazdar and C. Mellish. *Natural Language Processing in Lisp.* Addison-Wesley, 1989. Prolog version also available.

[43] G. Gazdar, G. Pullum, and I. Sag. *Generalized Phrase Structure Grammar.* Blackwell, Oxford, 1985.

[44] M. Gerlach and H. Horacek. Dialog Control in a Natural Language System. In *Proceedings of the 4th Conference of the European Chapter of ACL, Manchester, England*, pages 27–34, 1989.

[45] J.G. Giarratano and G. Riley. *Expert Systems: Principles and Programming.* PWS–Kent Publishing Company, Boston, 1989. ISBN 0-87835-335-6.

[46] J.M. Ginsparg. A Robust Portable Natural Language Database Interface. In *Proceedings of the 1st Conference on Applied Natural Language Processing, Santa Monica, California*, pages 25–30, 1983.

[47] B.J. Grosz. TEAM: A Transportable Natural-Language Interface System. In *Proceedings of the 1st Conference on Applied Natural Language Processing, Santa Monica, California*, pages 39–45, 1983.

[48] B.J. Grosz, D.E. Appelt, P.A. Martin, and F.C.N. Pereira. TEAM: An Experiment in the Design of Transportable Natural-Language Interfaces. *Artificial Intelligence*, 32:173–243, 1987.

[49] G. Guida and C. Tasso. IR-NLI: An Expert Natural Language Interface to Online Data Bases. In *Proceedings of the 1st Conference on Applied Natural Language Processing, Santa Monica, California*, pages 31–38, 1983.

[50] C.D. Hafner. Interaction of Knowledge Sources in a Portable Natural Language Interface. In *Proceedings of the 22nd Annual Meeting of ACL, Stanford, California*, pages 57–60, 1984.

[51] C.D. Hafner. Semantics of Temporal Queries and Temporal Data. In *Proceedings of the 23rd Annual Meeting of ACL, Chicago, Illinois*, pages 1–8, July 1985.

[52] C.D Hafner and K. Godden. Portability of Syntax and Semantics in Datalog. *ACM Transactions on Office Information Systems*, 3(2):141–164, April 1985.

[53] L.R. Harris. User-oriented Data Base Query with the ROBOT Natural Language Query System. *International Journal of Man-Machine Studies*, 9:697–713, 1977.

[54] L.R. Harris. The ROBOT System: Natural Language Processing Applied to Data Base Query. In *Proceedings of the ACM'78 Annual Conference*, 1978.

[55] L.R. Harris. Experience with ROBOT in 12 Commercial Natural Language Data Base Query Applications. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence, Tokyo, Japan*, pages 365–368, 1979.

[56] L.R. Harris. Experience with INTELLECT: Artificial Intelligence Technology Transfer. *The AI Magazine*, 5(2):43–50, 1984.

[57] G. Hendrix. Natural Language Interface (panel). *Computational Linguistics*, 8(2):55–61, April–June 1982.

[58] G. Hendrix, E. Sacerdoti, D. Sagalowicz, and J. Slocum. Developing a Natural Language Interface to Complex Data. *ACM Transactions on Database Systems*, 3(2):105–147, 1978.

[59] E.W. Hinrichs. Tense, Quantifiers, and Contexts. *Computational Linguistics*, 14(2):3–14, June 1988.

[60] G. Hirst. *Anaphora in Natural Language Understanding: A Survey.* Springer-Verlag, Berlin, 1981.

[61] J.R. Hobbs. Resolving Pronoun References. In B.J. Grosz, K. Sparck Jones, and B.L. Webber, editors, *Readings in Natural Language Processing*, pages 339–352. Morgan Kaufmann Publishers, California, 1986.

[62] M. Jarke, J.A. Turner, E.A. Stohr, Y. Vassiliou, N.H. White, and K. Michielsen. A Field Evaluation of Natural Language for Data Retrieval. *IEEE Transactions on Software Engineering*, SE-11(1):97–113, 1985.

[63] T. Johnson. *Natural Language Computing: The Commercial Applications.* Ovum Ltd., London, 1985.

[64] S.J. Kaplan. Cooperative Responses from a Portable Natural Language Data Base Query System. *Artificial Intelligence*, 19:165–187, 1982.

[65] S.J. Kaplan. Cooperative Responses from a Portable Natural Language Database Query System. In M. Brady and R.C. Berwick, editors, *Computational Models of Discourse*, chapter 3, pages 167–208. MIT Press, Cambridge, Massachusetts, 1983.

[66] G.P. Lopes. Transforming English Interfaces to Other Languages: An Experiment with Portoguese. In *Proceedings of the 22nd Annual Meeting of ACL, Stanford, California*, 1984.

[67] B.G.T. Lowden and A.N. De Roeck. REMIT: A Natural Language Paraphraser for Relational Query Expressions. *ICL Technical Journal*, 5(1):32–45, May 1986.

[68] B.G.T. Lowden, B.R. Walls, A.N. De Roeck, Fox C.J., and R. Turner. Modal Reasoning in Relational Systems. Technical Report CSM-163, University of Essex, Dept. of Computer Science, 1991.

[69] B.G.T. Lowden, B.R. Walls, A.N. De Roeck, C.J. Fox, and R. Turner. A Formal Approach to Translating English into SQL. In Jackson and Robinson, editors, *Proceedings of the 9th British National Conference on Databases (BNCOD)*, 1991.

[70] R. Lucas. *Database Applications Using Prolog.* Halsted Press, 1988.

[71] M.G. Main and D.B. Benson. Denotational Semantics for "Natural Language" Question Answering Programs. *Computational Linguistics*, 9(1):11–21, January–March 1983.

[72] J.L. Manferdelli. Natural Languages. *Sun Technology*, pages 122–129, Summer 1989.

[73] P. Martin, D. Appelt, and F. Pereira. Transportability and Generality in a Natural-Language Interface System. In B.J. Grosz, K. Sparck Jones, and B.L. Webber, editors, *Readings in Natural Language Processing*, pages 585–593. Morgan Kaufmann Publishers, California, 1986.

[74] W.A. Martin, K.W. Church, and R.S. Patil. Preliminary Analysis of a Breadth-First Parsing Algorithm: Theoretical and Experimental Results. Technical report MIT/LCS/TR-261, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1981.

[75] Natural Language Inc. *Natural Language 5.0.* (Commercial leaflets).

[76] N. Ott. Aspects of the Automatic Generation of SQL Statements in a Natural Language Query Interface. *Information Systems*, 17(2):147–159, 1992.

[77] Ovum Ltd., London. *Natural Language Markets*, 1991. ISBN 0-903 969 610.

[78] M. Palmer and T. Finin. Workshop on the Evaluation of Natural Language Processing Systems. *Computational Linguistics*, 16(3):175–181, 1990.

[79] C.R. Perrault and B.J. Grosz. Natural Language Interfaces. In H.E. Shrobe, editor, *Exploring Artificial Intelligence*, pages 133–172. Morgan Kaufmann Publishers Inc., San Mateo, California, 1988.

[80] Relational Technology Inc. *Using INGRES through Forms and Menus*, 1989.

[81] P. Resnik. Access to Multiple Underlying Systems in JANUS. BBN report 7142, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, September 1989.

[82] R.J.H. Scha. Philips Question Answering System PHILIQA1. In *SIGART Newsletter, no.61.* ACM, New York, February 1977.

[83] S. Sentance. Improved Responses from an English Language Front End. Master's thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1989.

[84] S.M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes Number 4, Centre for the Study of Language and Information, Stanford, California, 1986.

[85] W. Sijtsma and O. Zweekhorst. Comparison and Review of Commercial Natural Language Interfaces. In F.M.G. de Jong and A. Nijholt, editors, *Natural Language Interfaces, From Laboratory to Commercial and User Environments – Proceedings of the 5th Twente Workshop on Language Technology, Enschede, Twente University, NL*, June 1993. Also MMC Preprint no. 13, Institute for Language Technology and Artificial Intelligence (ITK), Tilburg University, NL.

[86] D.W. Small and L.J. Weldon. An Experimental Comparison of Natural and Structured Query Languages. *Human Factors*, 25(3):253–263, 1983.

[87] Sun Microsystems Inc. *SunSimplify 2.0 Reference Manual*, 1989.

[88] A. Tansel, J. Clifford, S.K. Gadia, S. Jajodia, A. Segev, and R.T. Snodgrass. *Temporal Databases – Theory, Design, and Implementation.* Benjamin/Cummings, California, 1993.

[89] M. Templeton and J. Burger. Problems in Natural Language Interface to DBMS with Examples from EUFID. In *Proceedings of the 1st Conference on Applied Natural Language Processing, Santa Monica, California*, pages 3–16, 1983.

[90] H.R. Tennant, K.M. Ross, M. Saenz, C.W. Thompson, and J.R. Miller. Menu-Based Natural Language Understanding. In *Proceedings of the 21st Annual Meeting of ACL, Cambridge, Massachusetts*, pages 151–158, 1983.

[91] R. Tennant, K.M. Ross, and Thompson C.W. Usable Natural Language Interfaces through Menu-Based Natural Language Understanding. In *Proceedings of CHI'83, Conference on Human Factors in Computer Systems, Boston.* ACM, 1983.

[92] B.H. Thompson and F.B. Thompson. Introducing ASK, A Simple Knowledgeable System. In *Proceedings of the 1st Conference on Applied Natural Language Processing, Santa Monica, California*, pages 17–24, 1983.

[93] B.H. Thompson and F.B. Thompson. ASK is Transportable in Half a Dozen Ways. *ACM Transactions on Office Information Systems*, 3(2):185–203, April 1985.

[94] F.B. Thompson, P.C. Lockermann, B.H. Dostert, and R. Deverill. REL: A Rapidly Extensible Language System. In *Proceedings of the 24th ACM National Converence, New York*, pages 399–417, 1969.

[95] F.B. Thompson and B.H. Thompson. Practical Natural Language Processing: The REL System Prototype. In M. Rubinoff and M.C. Yovits, editors, *Advances in Computers*, pages 109–168. Academic Press, New York, 1975.

[96] Trinzic Corporation, Bethesda, MD. *INTELLECT – Natural Language System*. (Commercial leaflet).

[97] Trinzic Corporation, Bethesda, MD. *KBMS – Knowledge Base Management System*. (Commercial leaflets).

[98] J.D. Ullman. *Principles of Database and Knowledge-Base Systems– Volume 1*. Computer Science Press, Rockville, Maryland, 1988.

[99] M. Wallace. *Communicating with Databases in Natural Language*. Ellis Horwood, New York, 1984.

[100] D.L. Waltz. An English Language Question Answering System for a Large Relational Database. *Communications of the ACM*, 21(7):526–539, July 1978.

[101] D. Warren and F. Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *Computational Linguistics*, 8(3-4):110–122, July-December 1982.

[102] R. Weischedel. A Hybrid Approach to Representation in the JANUS Natural Language Processor. In *Proceedings of the 27th Annual Meeting of ACL, Vancouver, British Columbia*, pages 193–202, 1989.

[103] S. Whittaker and P. Stenton. User Studies and the Design of Natural Language Systems. In *Proceedings of the 4th Conference of the European Chapter of ACL, Manchester, England*, pages 116–123, April 1989.

[104] G. Whittemore, K. Ferrara, and H. Brunner. Empirical Study of Predictive Powers of Simple Attachment Schemes for Post-Modifier Prepositional Phrases. In *Proceedings of the 28th Annual Meting of ACL, Pittsburgh, Pensylvania*, pages 23–30, June 1990.

[105] W.A. Woods. Procedural Semantics for a Question-Answering Machine. In *Proceedings of the Fall Joint Computer Conference*, pages 457–471, New York, NY, 1968. AFIPS.

[106] W.A. Woods, R.M. Kaplan, and B.N. Webber. The Lunar Sciences Natural Language Information System: Final Report. BBN Report 2378, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, 1972.