AN INTELLIGENT DATABASE INTERFACE FOR TURKISH

by

Osman Nuri DARCAN

B.S. in Computer Engineering, Boğaziçi University, 1987

Submitted to the Institute for Graduate Studies in Science and Engineering in partial fulfillment of the requirements for the degree of

Master of Science

in Computer Engineering

Boğaziçi University

1991

ACKNOWLEDGEMENTS

I would like to thank to Prof. Dr. Selahattin Kuru for his great help and guidance as the supervisor of this thesis.

I also thank to Doç. Dr. Oğuz Tosun and Doç. Dr. M. Akif Eyler for both their helpful comments and serving on my thesis committee.

In particular, I wish to express my gratitude to Doç. Dr. Sumru Özsoy for her invaluable guidance in reviewing many points in the syntax of Turkish

ABSTRACT

In this thesis, a portable natural language interface system for communicating with databases in Turkish is developed. The system does a two step transformation from a Turkish query in user's view to an intermediate meaning representation language D&Q and finally to a target database language SQL. It is composed of domain independent run-time modules for different processing stages, namely language processing, internal query generation and translation to SQL. Modules refer to the knowledge base in which diverse knowledge about the domain and the database are maintained. Two additional modules, to wit a spelling corrector and a history keeper are incorporated in the system.

A syntactic parser is used in analyzing queries. For the syntactic parser, a formalization of a subset of Turkish grammar based on the simple principle of general categorization incorporated with the notion of modifications between words is proposed and a grammar that consists of a collection of *rewrite rules* for the formal representation of sentences is discussed. A decision tree which works with suffix *strip off* approach is used for the morphological parsing of nouns. Parse trees produced for different types of sentences using the formalized grammar are given. Regarding to the meaning representation, an "intelligent" meaning representation generator which has a rule based reasoning capability is designed. The interpretations of some modification relations are discussed in details. Finally the interpretation of a full sentence is shown.

The system is tested on an imaginary *student-course-instructor* database, all examples refers to this database. Possible extensions for both the parser and the meaning representation generator are also proposed at the end of the thesis.

ÖZET

Bu tezde, veri tabanlarıyla Türkçe iletişim kurmayı hedefleyen, taşınabilir bir doğal dil arabirim sistemi geliştirilmiştir. Sistem, bir Türkçe sorgulama ifadesini önce ara birim anlam temsili dili olan D&Q'ya, daha sonra da hedef veri tabanı dili SQL'e çevirerek iki aşamalı bir dönüştürme işlemi gerçekleştirmektedir. Sistem, dil işleme, iç sorgulama ve SQL'e çeviri gibi birbirinden farklı üç aşamanın herbiri için kullanım alanından bağımsız modüllerden oluşmuştur. Modüller, içinde kullanım alanı ve veri tabanı hakkında çeşitli bilgilerin saklandığı bilgi tabanına başvururlar. Sistemin parçası olan diğer iki ek modül de imla düzelticisi ve tarih kaydedicisidir.

Sorgulamaların analizinde sentaktik bir ayrıştırıcı kullanılmaktadır. Sentaktik ayırıcı için, sözcükler arasında anlam niteleme nosyonuyla bütünleştirilmiş basit genel kategorizasyon ilkesi üzerine kurulan bir Türkçe gramer altkümesi formalizasyonu önerilmekte ve cümlelerin formel temsilleri için kullanılan *yeniden yazma kuralları* kümesinden oluşan bir gramer tartışılmaktadır. İsimlerin morfolojik ayrıştırılmasında sonekleri atarak çalışan bir karar ağacı yaklaşımı kullanılmaktadır. Formelleştirilmiş gramerden yararlanarak farklı tipde cümleler için üretilen ayrıştırma ağaçları verilmektedir. Anlam temsili bakımından da, kurala dayalı akıl yürütme yeteneğine sahip 'akıllı' bir anlam temsili üreticisi tasarlanmıştır. Bazı anlam niteleme ilişkilerinin yorumlanmaları da ayrıntılı biçimde tartışılmaktadır. Son olarak da tam bir cümlenin yorumlanışı gösterilmektedir.

Sistem, hayali bir *öğrenci-ders-hoca* veri tabanı üzerinde denenmiştir ve tezdeki tüm örnekler bu veri tabanına dayanmaktadır. Tezin sonunda da ayrıştırıcı ve anlam temsili üreticisi için olası geliştirmeler üzerinde durulmaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

Computers are widely used in almost every area of everyday life in Turkey. Computerized data processing is developing very fast and the use of management information systems is expanding rapidly in many organizations. To use the information stored in the database every user has to learn a query language as a special language for communicating with the database. As an example, consider the database **öğrenci-ders-hoca** 'student-course-instructor', given in Appendix C, consisting of three basic files *öğrenci* 'student', *ders* 'course', *hoca* 'instructor' and a relation file *karne* 'grade report' [1]. A user who wishes to know about students taking the course called CMPE100 should formulate a query in a declarative query language as follows :

```
SELECT st_no
FROM   student
WHERE
    student.st_no = stud_course.st_no
    stud_course.code = course.code
    course.c_name = CMPE100
```

However, it is usually the case that most of the people having a good idea of the information residing in these databases are unable to formulate their question as a sequence of requests and conditions[2] for a particular database. The need to learn a complex language becomes the main problem of people who wish to use computers. Therefore, the computerized data processing brings the need for easy database access interfaces for Turkish computer users from top-level managers to clerks who need to access the information residing in their computers. The easiest way of communication for a human is by using his natural language. The user has to translate his thoughts into a non-procedural query language where the use of an ordinary natural language is possible. In that way the burden of man-system communication will be totally put onto the machine. The system will translate the user's request given in his natural language

---

[1] *Karne* is named as *stud_course* in the example database given in the Appendix C. *stud_course* will be used for *karne* in place of *grade report* hereafter.

[2] The connective AND between each line in the list of conditions is omitted

into a sequence of instructions and conditions to be processed by a database query interpreter to generate the desired result.

The major advantage of having a natural language interface is not to free users from needing to know about query languages but the bridging effect between two views. The fact that the request is expressed in the way users perceive data as opposed to the way systems store them provides users with easy to use access environment to databases. This is because phrasing the query in a database query language requires not only the learning of strict templates but some understanding of how the data is represented in the database as well. For instance, in a natural language interface, the question posed earlier may easily be formulated as one of the following two sentences,

Hangi öğrenciler CMPE100 dersini alıyor 'which students are taking the course CMPE100'

CMPE100 alan öğrencileri göster ' show the students who take the CMPE100'

where the user does not need to know the external view of the structure, so called, conceptual schema of *student-course -teacher* database.

Over the last three decades a considerable amount of work has gone into the development of English language processing systems[1][1]. Early developments were in the formalization of the English grammar, in developing syntactic and semantic parsing techniques and in designing English language question answering systems. In the field of natural language query systems in English, many systems have contributed to advancing the state of art, pointing out problems in natural language front-end design, and some of them has already reached the commercial market. With the developments in the field of artificial intelligence (AI), AI techniques are incorporated with natural language processing, and intelligent database user interfaces are designed[2,3]. However, no attempt has been made to develop a natural language interface (NLI) for Turkish.

The aim of this thesis is to develop a portable natural language interface which will produce a query in a declarative language from a Turkish question. In designing a natural language interface, there are problems that need to be solved on the front end, and at the back end, namely parsing of the Turkish question and translating the question into a query in a declarative database language, respectively. Therefore, before

[1] References enclosed in brackets refer to the bibliography.

we design a natural language interface for databases, it is necessary to formalize the Turkish grammar since other than the study carried out by Meskill[4], which provides a transformational model for Turkish, no attempt has been made to formalize the Turkish grammar as a set of rules until now. In this thesis, we propose a formalism for the Turkish grammar and focus on designing a domain independent Turkish parser which is capable of parsing a subset of the Turkish grammar by accessing domain dependent information about words and concepts relevant to the database. It will be used in the Turkish language processing component of our natural language interface. We intent to design a domain independent natural language interface system that translates the query into an internal representation language, namely an extended version of Wallace's D&Q [5]. The internal query representation will be converted into a query in a declarative database query language; the query now contains actual values for a particular database. This final query can be processed by the database management system to produce the result.

Providing an easy access to databases increases the burden of machine translation. First of all, as in the above examples; the user's query does not address to the actual database schema; querying in the user's own view necessitates an extensive capability in matching user's view onto actual data files and their relevant field names. Reducing the requirement for the user to learn about the details of the database involves knowledge of the problem domain to map the user's query onto actual values. Secondly, queries having the same deep structure may be formulated in various ways. Reduction of the necessity for user to conform a list of artificial syntax in Turkish requires a powerful parsing mechanism[5]. Question words and relative clauses are extensively used in query formulation. Therefore the parser must process any phrase built with any combination of question words. However question words may cause semantic ambiguity because they are less descriptive. For instance the query *hangi ögrenciler CMPE100 dersini aliyor* may also be formulated less informatively as *kim CMPE100 aliyor* 'who is taking CMPE100'. Ambiguity is one of the primary problems in natural language understanding. A good system should clarify the ambiguity and try to recognize the user's intention correctly. In addition, once the natural language interface is written, it must be possible to adopt it to provide access to different databases, that is to different domains. Adaptation can be achieved by separating the domain dependent knowledge from the domain independent knowledge. Hence one who is not familiar with the natural language processing techniques can transfer the system to another domain.

As we pointed out above, natural language interface design requires more than just simple mapping from the natural language queries into formal representation. It also requires the natural language comprehension capability in representing the literal content of a natural language query into a formal representation which involves developing reasoning on concepts required. Reasoning is the basic part of the deep comprehension which involves not only linguistic capabilities but also deductions and analogies on domain specific knowledge. Therefore a natural language interface must incorporate a knowledge base containing knowledge about at least the natural language, the problem domain and the database.

Aside from the introductory chapter, this thesis contains six more chapters. In Chapter II we discuss some main concepts in natural language interface for databases (NLI). Chapter III gives an overview of the components of our NLI model, its knowledge base and contains a sample run to demonstrate the capabilities of the model. The syntactic parser for Turkish developed as a component of our NLI and the formalization of the Turkish grammar are discussed in Chapter IV. Chapter V briefly explains the syntax of the Wallace's D&Q meaning representation language, introduces some necessary changes made to it, and discusses representation of constituents of the Turkish sentences in extended D&Q. In Chapter VI we discuss some implementation issues. Finally, Chapter VII gives the conclusion and general remarks for future improvement of the NLI as well as the Turkish grammar.

This thesis contains four appendices following the conclusion part; in Appendix A the Turkish grammar used in NLI is represented in terms of rewrite rules and transition network separately. Appendix B gives the definition of Wallace's D&Q syntax and changes made to it. Appendix C gives the example database used in the interpretation. Appendix D gives the program listing together with the contents of the data files.

II. NATURAL LANGUAGE INTERFACE SYSTEMS FOR DATABASES

This chapter starts with a brief discussion of the approaches in designing the major components of an NLI. Some important issues in NLI design such as portability, lexicon types and value recognition are then summarized. Furthermore, two language problems in NLI are introduced with examples in Turkish.

2.1. Architecture of a Typical NLI System

The major components of a natural language interface system for database are the parser, the formal query generator and the database access routines. The parser is the main natural language processing module. It is actually a program that analyzes the grammar of the natural language query. The formal query generator translates the parsed query into a form that can be interpreted by the database access routines. This latter evaluates the query on the database. In addition to these routines, systems in the literature include additional routines such as spelling correctors [6,7], routines to solve anaphora and elliptic structures, help modules [2,3], knowledge acquisition routines [8] , and editors [2,3]. The difference between general purpose natural language systems and NLI systems is that both the query analysis and the formal query generation are restricted by both the structure of the database and the subject area covered by the database [5].

2.1.1. Parsing and Meaning Representation

The Parser analyses the grammar of the sentence. The analysis is based on the semantics of the subject area or the syntax of the language. In the NLI design, there are mainly two approaches used in parsing; semantic analysis and a combination of

syntactic and semantic analysis. Semantic parsers use a grammar that is specifically tailored for the applications. In fact all grammars for NLI systems must have semantic checks since the subject area is restricted to the database.

A good example of NLI systems that uses a purely semantic approach is LIFER [7]; its grammar includes sentence patterns like "what is < attribute > of < ship >", where nonterminal symbols, i.e. < attribute >, < ship > must be associated with individual words and fixed phrases from a set defined for each nonterminal symbol in LIFER's lexicon. In its pure semantic approach, the grammar has no noun phrases or verb phrases, but rather a special set of categories for the particular task. For instance, nouns are not grouped into a single category like NOUN, but they are categorized as SHIP or ATTRIBUTE.

The grammar used in REL [9,10] is similar to LIFER's grammar. REL's grammar includes rules consisting of more general categories than LIFER's grammar. For example REL's grammar has the category NAME instead of SHIP.

PLANES [6] also uses semantic grammars represented as an augmented transition network. It includes subnets such as PLANTYPE which are oriented towards the information held in the database. There are subnets in PLANES grammar for each different semantic object and each subnet matches only phrases with a specific meaning. Only the qualifier subnets in the PLANE grammar are syntax driven.

Some NLI systems use a combination of semantic and syntactic analysis. The grammar used by EUFID [11] parser is essentially semantic. The concepts are organized into a case system. The meaning of a query in EUFID is represented in a tree structure by connecting cases. Each connection corresponds to concept-to-concept linkage. Each node in the tree is a semantic-graph node. The parser uses syntactic information "on demand", that is when such information is necessary to solve semantic ambiguities. In FRED [2], the approach to parsing is also based on semantic case analysis. FRED's grammar is a collection of case frames. These case frames are associated with either the domain specific entities or general entities such as quantifiers, date and time expressions and arithmetic comparisons. The syntax of acceptable queries are stored in the case frames. There exist surface case frames which can be attached to case frames to parse complex sentences. Possible values for cases are recognized by syntactic parsing of the query. In KID [3], syntactic and semantic analysis proceed incrementally. A rule in KID grammar consists of both syntactic and semantic parts. Syntactical analysis determines the possible modification relationship between phrase constituents and semantic checks are made on these modification relationships. Similar to KID, the parser of TEAM [8] uses syntactic and semantic rules.

Semantic grammars suffer from many drawbacks; they embed semantic information in grammatical categories and they make no distinction between domain specific knowledge and general knowledge [3] , and they are difficult to expand because they become highly redundant when expanded for more general use since they fail to capture certain generalizations about the language. It is clear that a parser which uses a purely syntactic grammar is not sufficient to understand the meaning of the query. Furthermore, dividing the syntactic and the semantic parsing into two completely separate steps will lead to a tremendous increase in the number of possible parses. The best approach to parser design is to use combination of semantic and syntactic analysis. Syntactic categories such as noun phrases and verb phrases can be used for parsing and only at lower level need the grammar can be tailored to the application.

2.1.2. Query Understanding and Database Query Generation

The formal query generation is actually a translation process from the output of the parser into a database query language. In the literature, we distinguish different approaches to query generation; single step translation and multi-step translation. LADDER [7] is an example of systems that use single step translation in its language processing component LIFER. In LIFER, each phrase pattern has an expression associated with it. The expression represents the the meaning of the phrase. Each non terminal symbol in the phrase pattern has its own production rules and the meaning of each nonterminal symbol is computed using its production rule. The resulting expression consisting of meanings of each nonterminal symbol is a call to the database access routine. TEAM and PLANES resemble each other in that they use two step transformation. TEAM first produces a logical form and translates it into a formal database query. Similarly, PLANES express the meaning of the natural language query in interim form, this interim form is then translated into a relational calculus expression. FRED, EUFID and KID are very similar in design. Each has a multistep translation from the natural language to meaning representation, to an intermediate language and finally to a target database language. For instance, EUFID mapping module transforms the tree structure output of the analyzer into a string of token(IL). Semantic-graph nodes are converted into database files and field names, and connections are converted file-to-file or file-to-field connections of the database. IL is finally translated into the actual DBMS query language. KID uses an approach similar to EUFID. KID converts the meaning representation into a world model query which consists of the target list and a list of conditions. Rule-based translation mechanism is

used to translate the world model query into a database query. Rules perform basic mapping, derivation, generalization, and functional joins. FRED slightly differs from KID and EUFID because it converts the query into an intermediate *database* language. Query Planner translates the case frame based meaning representation of the query into a linear string of tokens expressed in the virtual database language V/DELPHI. The virtual query refers to domain objects rather than actual database files and fields. Frame-based production rules are used for this translation. At the next step V/DELPHI query is converted to a DELPHI query. During this conversion virtual fields in the domain file are converted to actual fields. Finally the DELPHI query is transformed to the target database language by applying a set of transformation rules expressed in a special rule-based transformation language Troll.

Database access routines evaluate the query generated and send back the tuples satisfying the conditions. Some systems [3,6] have also response generators deciding on how the output should look like.

2.1.3. Spelling Correction

Queries in natural language can contain misspelled words or words that do not exist in the lexicons. Some systems include a spelling corrector component which attempts to deal with unrecognized words. Given that the system detect an unrecognized word, there are two approaches to correct the spelling error. The first approach is to enter into a dialogue with the user. When INTELLECT [12] fails to recognize a word it engages a dialogue with user in which the user is required to correct the spelling or enter the field where the unrecognized word should appear in the database. It then searches the dictionary or the database for the unrecognized word. The second approach is to attempt to guest the misspelt word as LIFER and PLANES do. The spelling corrector in PLANES is called as soon as the input word is not found in the dictionary. It tries to match the misspelt word against a list of correct words and produces a list of candidates which consists of words close to the input word. Then, it enters into a dialogue with the user to confirm one of these candidates. If no candidates are found or if the user rejects all the suggested candidates then the user is required to add the word into the dictionary. On the other hand, LIFER records the failpoints on a failpoint list and attempts to complete the parse by trying other production rules. If a complete parse is found, the failpoints are ignored. But if an input can not be parsed the list is used by the spelling corrector which substitutes the closest match between the words of that category.

2.2. Portability

One of the fundamental features of the NLI is application independence, so called *portability*. Natural language systems can be grouped in two categories from the point of view of portability; *single domain systems*, those which are built to provide access for only one domain, and *transportable systems*, those which can be easily adaptable to provide access to databases for which they are not written.

A single domain system is a system whose grammar and query generator must be completely rewritten for each new application [7]. Such a system usually provides useful tools for constructing these components.

Considering *transportable systems*, there exist different levels of portabilities depending on how the system is transported and who does this transportation.

A system may be transported by the programmer as it is done in transporting one of the earliest NLI systems ROBOT [13] to several domain simply by supplying new rules of grammar and modifying meaning of some words.

Second level of transportability is the one at which the actual user provides information about the new domain. In REL , the user can define the entire database or extend it by adding his own definitions and changes. In these two levels of transportability information about the language and information about the domain and database structure are **intermixed.** Transporting such systems to a new domain requires the **changing of the parsing procedures and the domain information.**

Separation of domain dependent and domain independent information provides transportability of the third level, where information about the new domain is supplied through an interactive dialogue with data processing personnel who are not familiar with natural language processing techniques. TEAM is an example of transportability at this level where three kinds of information, the lexical information, the conceptual information and the information about the structure of the database, are required to be transported to a new database. EUFID, KID and FRED achieve portability in the same way.

For instance, application specific data are supplied to EUFID as tables. Application specific data involve the dictionary, the semantic information and the organization of the data in the database. FRED has a domain dependent knowledge base which contains the semantic grammar, frame-based production rules and transformation rules. KID has a world model containing a domain model, linguistic knowledge and database mapping knowledge. The user has only to update the domain dependent information or supply new information to transport the system to a new domain. In addition to that, EUFID, KID and FRED translate the natural language query into an intermediate language, which is then converted into target database language.

2.3. Lexicons

Any natural language interface necessitate a lexicon in which valid sentence constituent are kept. As we mentioned above an NLI is restricted by the subject area covered by the database i.e. people (student and instructors) information for the university database, personnel information for a company and so on. Considering their semantics, a natural language query includes two kinds of words [5] ; **function words**, whose meaning are independent from application and **content words**, which gain meaning from the subject area covered by the database. A content word may refer to an entity or a property. It may also be a value with which an entity or a property is associated. Thus, for a portable NLI, new content words must be supplied when the subject area changes. Thus we must distinguish between lexicons containing these words. Broadly speaking, there are three different types of lexicons in NLI [14]: General entries, structural entries and volatile entries.

General lexical entries are words whose meaning are independent of any particular domain. Question words such as *kaç*, 'how many', *hangi* 'which', and "comparative" adjectives such as *az* 'less', *çok* 'more' and *büyük* 'big' are some of examples. They can be practically used for any domain.

Structural entries are terms which make reference to some aspects of the database. Entries in the structural lexicon are domain dependent. The words that make reference to specific entities or attributes, synonyms for words such as *talebe* for *öğrenci* 'a synonym for student, student', or verbs that are specially used in the domain

under consideration. An example of the latter is *al* 'take' which may mean *ders/not almak* 'take a course' or *satın almak* 'buy' considering *student-course* or *supplier-part-shipment* databases respectively.

Volatile entries are numbers or unknown terms encountered during the input, the word *CMPE100* in our introductory example is a volatile entry and is only useful for the duration of the session. In most of the natural language interface systems, the volatile lexicon is not a part of the lexicon and special approaches are used to recognize values corresponding to volatile entries in natural language queries.

2.4. Value Recognition

It is necessary to identify the word *CMPE100* in our introductory example as a noun representing a course name. If this entry is explicitly coded in the lexicon for future reference, the query system may do false inferences when the course *CMPE100* is not offered any longer. There are three basic ways to recognize a value in a query. They can be explicitly listed in the dictionary, found in the database itself or recognized by a pattern or text.

One solution to this problem is to use the database itself as a volatile lexicon[10]. For a question like *kim CMPE100 dersini alıyor* 'who is taking the course CMPE100'. The natural language processing system may refer to the database to check the existence of a field containing the value *CMPE100*.

An alternative solution to this problem is to infer fields that could contain unknown terms [14]. An unknown term may be treated as an item of one of these fields.

Values can also be recognized by a pattern. Hence it is not necessary to itemize all instances in the dictionary. For example, a *date* may be entered as gün/ay/yıl 'day/month/year' so an input matching that format is recognized as the *date*.

Each of these solutions has disadvantages [14]. If all values are stored in the dictionary it would result in an enormous number of dictionary entries. Patterns can be used only if users can be enforced to fit the data to the pattern. Especially proper names are poor choices for patterns. Solution using the database itself is costly and

unsatisfactory because of the need for a database search for every unrecognized word. It can only be accepted as a satisfactory solution if the database is small. In our model we do not focus on value recognitions, some of the volatile words are stored in the dictionary since the number of entries are small. The user is forced to begin a proper noun entry with a capital letter.

2.5. Two Language Problems in NLI

Here are introduced two language problems in NLI, namely conjunctions [11] and ambiguities [14] .

The difference between the way the user perceives the data as opposed to the way the machine stores them brings the conjunction scooping problem in question. The natural language use of *and* which usually denotes a conjunction, must be interpreted in some cases as *or* which usually denotes a disjunction. Consider the sentence,

kimya ve fizik hocalarını göster

'show the chemistry *and* the physics instructors'

The meaning of *ve* 'and' must be changed to logical **or**, the phrase *kimya* **and** *fizik* 'chemistry and physics' must be formally stated as *kimya* **or** *fizik*. The disjunctive use of **and** occurs when it joins two alternative values. Therefore the solution to this problem is to change **and** to logical **or** when the two constituents with the scope of the conjunction are value for the same field.

Ambiguities may be of two types in language understanding; semantic ambiguities and syntactic ambiguities.

Semantic ambiguities which occur when the parsed constituents may have several possible meanings. Consider the question

kim CMPE100 alıyor 'who is taking CMPE100'

The question pronoun *kim* 'who' may refer to a student or to an instructor depending on the context. To resolve this ambiguity, the information gathered in the rest of the

sentence must be examined. Determining the intended referent in the database may be much more difficult when the following query is considered,

Ahmetin derslerini göster 'show Ahmet's courses'

Ahmet is a proper noun corresponding to a particular name, as in the above example, but its meaning may be either a student's or a instructor's name depending on the context. The referent can be determined by observing the database schema provided that the conceptual database schema reflects the semantics of the domain.

Syntactic ambiguities occur when modifiers or modifying phrases are physically separated from the term they modify. No words are ambiguous in our grammar when the set of possible constituents of a phrase are well determined. The distance between modifiers and the actual word they modify is zero in our grammar.

III. A MODEL FOR A TURKISH NLI

After having discussed some important concepts of NLI design in Chapter II; in the first part of this chapter we introduce our design objectives. Next section gives an overview of our knowledge based NLI model being designed, with a brief explanation of each component of the design and the knowledge base. We conclude this chapter with a list of natural language queries acceptable in our model.

3.1. Design Objectives

Our main objective is to design a portable natural language database interface that will allow a non programmer to easily obtain information from a database. We formulate our subgoals as follows :

-The system should have a parser for Turkish with a wide range of semantic and syntactic structures which may appear in a natural language query.

-The system should allow querying in conceptual level rather than on a factual level.

-The system should free the user from having to know the physical database organization.

-The system must correct some spelling errors, and tolerate a range of nongrammatical but correctly understandable requests. Certain kinds of extensions such as addition of new words or synonyms, should be possible to perform.

-The system should be transportable to another domain without any knowledge of the program.

3.2. Components of the System

In order to satisfy the subgoals given in Section 3.1., a natural language interface should make a distinction between different processing phases and maintain a clear cut separation between domain independent and domain dependent parts of the NLI[15]. Our approach to the NLI design is to have general purpose run-time modules for different processing stages and to supply the domain dependent knowledge as data to these modules in order to minimize the consequence of changes in the interface environment that may occur due to the changes in the topic of discourse, in the structure of the database or in the database management system. As shown in Figure 3.1., our system consists of four domain independent run-time modules, namely the analyzer, the translator, the spelling corrector and the history keeper, and a knowledge source. Considering the above mentioned parts that may vary from one domain to another, the domain dependent knowledge include the synonym dictionary (part of the lexicon), the semantic relationships dictionary, a dictionary containing semantic definitions of words, the conceptual database schema in user's view and the domain-to-database mapping table. The domain independent knowledge is kept in the basic dictionary.

The analyzer accomplishes parsing of the natural query and generates a D&Q expression which represents the meaning of the query. The output of the analyzer is the input for the translator which converts the D&Q expression into a SQL query. The spelling corrector attempts to correct a misspelled word by suggesting a list of candidates that can substitute it and update the lexicon if necessary. The history keeper is a register which keeps information about the previous query, which may be referred to in order to provide missing information in the current query. The knowledge source provides domain independent and domain dependent knowledge.

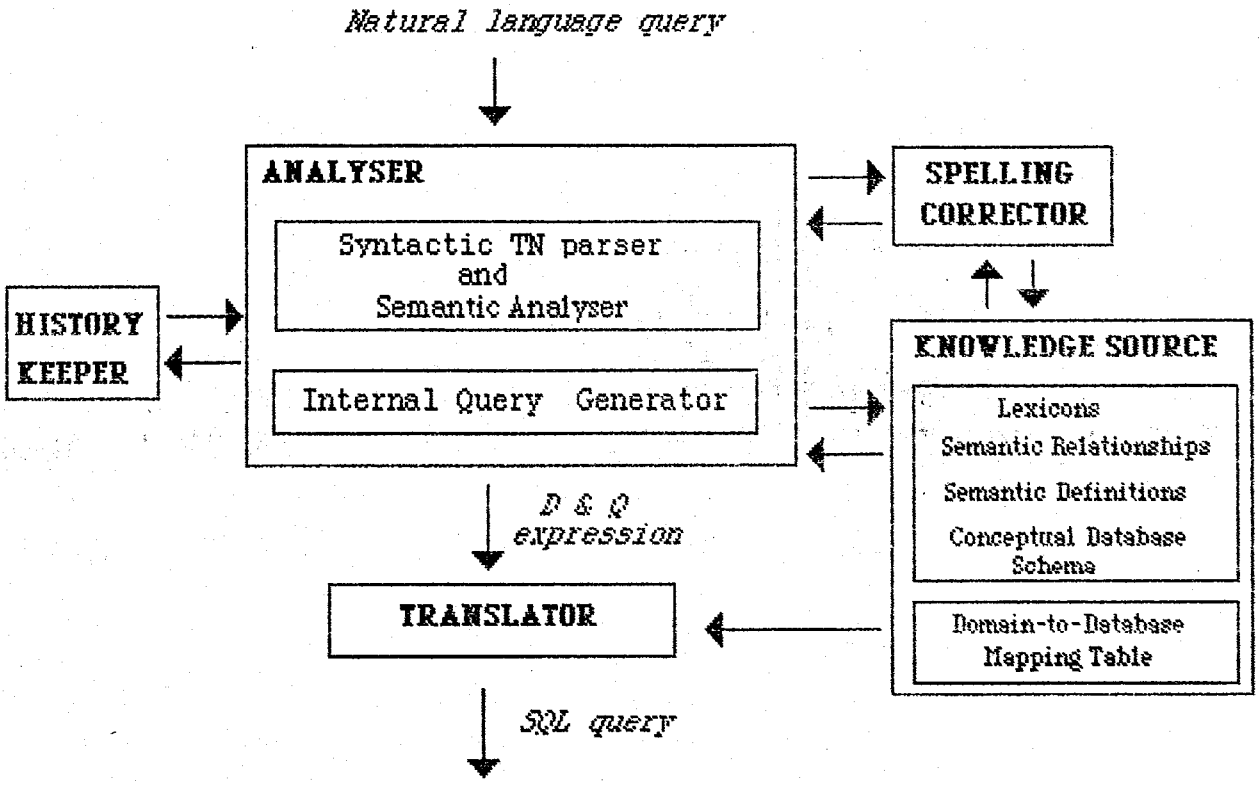


Figure 3.1. An overview of our model

3.2.1. Analyzer and Translator

Processing of a user query is divided into two main phases; syntactic parsing integrated with the semantic analysis and internal query generation. The analyzer is a general purpose domain-independent language processor and a meaning representation generator. The language processor does syntactic and morphological parsing, and semantic analysis. A phrase in Turkish is a collection of semantically ordered, modified or unmodified noun phrases. The syntactic parser splits the sentence into noun phrases by locating head nouns and their corresponding modifiers. For this process, the syntactic parser uses a context-free grammar. Rules state the way of linking different phrase constituents into a noun phrase and specify the way in which noun phrases can be combined. Syntactic parsing is accomplished by matching input sentences against phrase patterns represented as transition networks. Morphological analysis is used to decide on the functions of nouns. Semantically ordered phrase constituents are transformed into a canonical form by substituting for synonyms or abbreviations the equivalent words found in the conceptual database schema. For instance, the word *ogrenci* 'student' will substitute the given input *talebe* 'a synonym for student'. The

syntactic parser uses the basic lexicon and the synonym dictionary. The semantic analysis is integrated with the syntactic parsing. It checks whether the relations between the noun phrases and the verb are meaningful. It solves semantic ambiguities and finds the referent of question pronouns and proper nouns. The knowledge on the semantic relations is used in this analysis.

The meaning representation process can be visualized as "the sentence tree produced by the analyzer is traversed bottom up and at each node the description of the head noun created or an existing description is extended or two descriptions are combined". However, meaning representation and linguistic analysis are concurrent processes in our model. They proceed incrementally. While parsing user's query, the query is also translated into a formal meaning representation language, namely an extension of D&Q of Wallace. Internal query generator is called after the successful parse of a constituent. It maps the parsed word onto an entity and attribute name. It creates the descriptions of the entity when the parsed constituent is mapped to an entity or extends the existing description of the entity when a word corresponding to a property of this entity is parsed. Regarding the relations, the internal query generator consults the conceptual database schema to decide on whether the semantically correct relation between the noun phrases and the verb are applicable in the database. Noun phrases stored in the memory are combined into a nested description according to the relation involved in the natural language query and in the database schema. Therefore, the meaning representation process involves the following four decisions:

(1) Deciding on which relations to take

(2) Deciding on how to modify relation

(3) Deciding on how to combine relations

(4) Deciding on which operations to perform on return fields

The second run-time module in our system is the translator. The translator transforms the expression represented in the meaning representation language into the database specific target language of a relational database management system, namely SQL. First by applying a set of database independent transformation rules, the query is converted into a general declarative language. Next, domain objects found in the query are mapped onto actual database files and fields. The conversion from user view to actual database files is specified in the domain to database mapping table.

3.2.2. Spelling Corrector and History Keeper

The spelling corrector module is called when a word is not found in the lexicon during the syntactic parsing. This module attempts to find the lexicon entry close to the input string. The method used to find candidates for an unrecognized word is to split the word into two parts and to attempt to match the first part of the word with the first parts of the existing words in the lexicon. When no matching word is found then the second part is tried. The user is asked for candidates one by one until one of the suggested candidates is accepted. If no candidate is found or all of the candidate are rejected then the user is asked to enter the function of the unrecognized word and its domain value if it is a noun. Consider the NLI query

kim hangi ***derti*** *alıyor* ' who is taking which course (course is misspelled) '

The parser first notes that the word *dert* is not in the lexicon and the spelling corrector is called. It finds that the word *ders* is the most similar word to *dert* and prints the following message;

dert yerine *ders* kullanılabilir mi ? ' substitute *ders* for *dert* ? '

If the user types **e** (for evet/yes), the system substitutes *ders* for *dert* and continues to parse.

The history keeper component stores descriptions used in the last query together with its form and the result returned for it. When some information is missing in the query, the history keeper component is referred to in order to fetch its previous value. Pronoun references or the referent of the demonstrative adjectives can also be recognized in the same way. As an example, demonstrative adjective *o* 'that' in the phrase *o dersi* 'that course' can be recognized by referring to the previous description of *ders* 'course' and using the properties of the last description as the current one. Assume that ders has the following D &Q description in the history keeper,

the-1-qual(X,[],ders(**ad=kimya**))

In representing the meaning of *o ders* 'that course', the current description of *ders* 'course' will be extended with the addition of the selection **ad=kimya**. As another example, consider the sentence *dersini göster* 'show his course'. The word *ders* has the possessive suffix *-i* but its genitive component is missing in the sentence. It is supplied by taking the last word in possessive relation with the word *ders* from the history keeper. The user is asked to provide the necessary information when the description does not exist.

3.2.3. Knowledge Source

As explained in the Section 3.2.1., each stage involved in transforming a natural language query into a database language query necessitates different types of knowledge. Our system has a knowledge source which consists of five different kinds of information; lexical information, semantic relationships, semantic definitions, conceptual database schema in user's view, and domain to database mapping.

Lexical information consists of the syntactic property of the words that will be used in querying and the concept information. Concept information is called the class information, which defines the kind of concepts to which the word refers. There are two types of lexical information; the **basic dictionary** which keeps domain dependent words such as pronouns, adjectives, conjunctives, and the **synonym dictionary**, which stores domain dependent words which will be used to substitute user terms in transforming the query into the canonical form.

Information about the user's view of the database is actually a **database schema in user's terms.** It includes the definition of the structure of each file in the database. Each file is about some kind of an object such as *ders* 'course', *öğrenci* 'student', *hoca* 'instructor' and the fields of the file contain object's properties like *kod* 'code', *ad* 'name' *kredi* 'credit' for the file *ders*. This dictionary contains three sorts of information for each field of a file; file name, type of class from which it gets its value and an indicator for key fields. This information is used to select the correct meaning of the query through the knowledge of how the information is stored in the structure of the database.

Semantic relationships dictionary keeps information about the functional relations between objects and verbs that can take objects as arguments. Objects are entity names or attribute names. Thus, semantic relationship dictionary describes the correspondence between different entities, attributes of entities, and verbs. Each group consists of an act and a list of objects that can meaningfully occur with that act.

For example, entries for the verb *al* 'take' are;

(al, ders, object)

(al, not, object)

(al, öğrenci, subject)

(al, hoca, ablative)

Semantic relationships dictionary contains entries representing the cooccurence of two objects as well, i.e. the cooccurence of the words *not* 'grade' and *ders* 'course' is also semantically meaningful provided that second object has the case marker locative on it, e.g.

(not, ders, locative)

Figure 3.2. shows an associative network representation of the verb *al* 'take' and *ver* 'give' through their common arguments and a possible relationships between arguments.

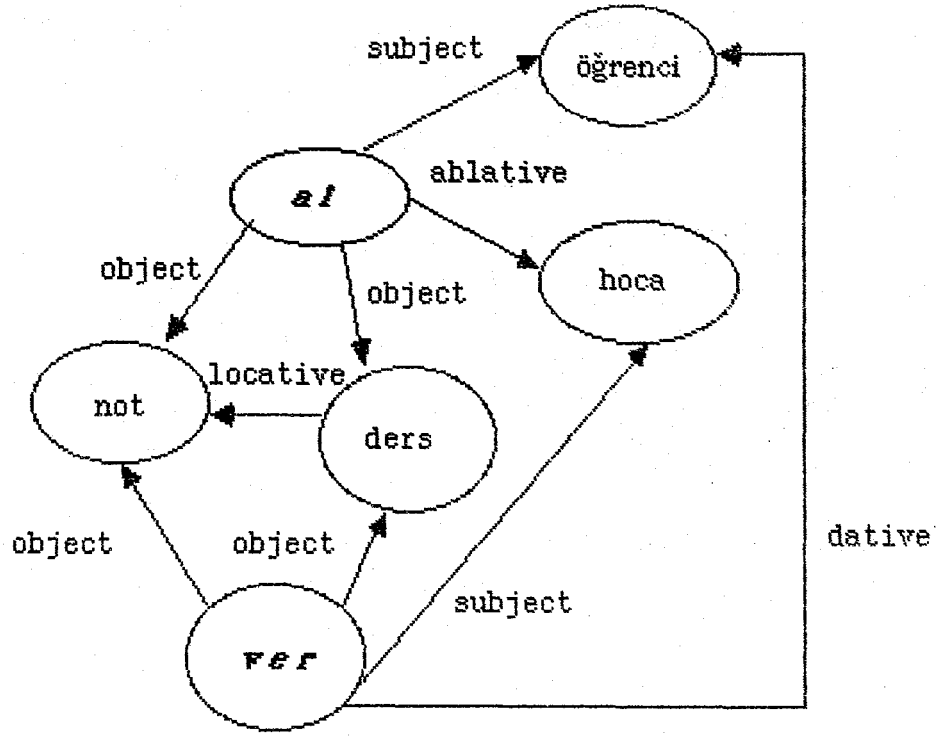


Figure 3.2. Relationships between two verbs : *al* 'take' and *ver* 'give'

Dictionary of semantic definitions contains entries showing how an adjective or a noun will modify an entity or a relationship in terms of its attributes. The need for such a dictionary can be explained on an example as follows: the meaning of *iyi* 'good' occuring as the modifier of *ders* 'course' can not be expressed by extending the description of *ders*. Therefore, the meaning of the compound *iyi ders* 'good course' must be explicitly stated. *Iyi ders* 'good course' is defined in terms of its attribute *hocano* 'instructor's id' of the entity *ders* as *hocano = 1234*.

Domain to database mapping table includes an entry for each field of a file. Each entry contains the file name and the field name in user's view and their corresponding equivalents in the actual database. An example for the file *ders* 'course' is the following ;

(ders, kod, course, code)

(ders, ad, course, c_name)

(ders, kredi, course, credit)

(ders, hocano, course, instr_no)

3.3. Sentences Accepted in Our Model

This section gives a list of acceptable natural queries in our model. Each natural language query is followed by the SQL expression generated for it. The list is intended to include all the different types of queries that can be processed by our model.

kim CMPE100 alıyor

```
SELECT st_no
FROM   student
WHERE
 student.st_no = stud_course.st_no
 stud_course.code = course.code
 course.c_name = CMPE100
```

kim CMPE100 veriyor

```
SELECT instr_no
FROM   instructor
WHERE
 instructor.instr_no = course.instr_no
 course.c_name = CMPE100
```

hangi hoca CMPE100 veriyor

```
SELECT instr_no
FROM   instructor
WHERE
 instructor.instr_no = course.instr_no
 course.c_name = CMPE100
```

kim hangi dersi veriyor

```
SELECT instr_no code
FROM   instructor course
WHERE
 instructor.instr_no = course.instr_no
```

kim ne veriyor

```
SELECT instr_no code
FROM   instructor course
WHERE
 instructor.instr_no = course.instr_no
```

kim ne alıyor

```
SELECT st_no code
FROM   student course
WHERE
 student.st_no = stud_course.st_no
 stud_course.code = course.code
```

Mehmet hangi dersleri alıyor

```
SELECT code
FROM   course
WHERE
 course.instr_no = instructor.instr_no
 instructor.inst_name = Mehmet
```

CMPE100 alan öğrencileri göster

```
SELECT st_no
FROM   student
WHERE
 student.st_no = stud_course.st_no
 stud_course.code = course.code
 course.c_name = CMPE100
```

Mehmetin derslerini göster

```
SELECT code
FROM   course
WHERE
 course.code = stud_course.code
 stud_course.st_no = student.st_no
 student.st_name = Mehmet
```

Ahmetden ders alanları ver

```
SELECT st_no
FROM   student
WHERE
 student.st_no = stud_course.st_no
 stud_course.code = course.code
 course.instr_no = instructor.instr_no
 instructor.inst_name = Ahmet
```

Ahmetden ders alan kimya öğrencilerini göster

```
SELECT st_no
FROM   student
WHERE
 student.st_no = stud_course.st_no
 student.dept = kimya
 stud_course.code = course.code
 course.instr_no = instructor.instr_no
 instructor.inst_name = Ahmet
```

kim kaç ders alıyor

```
SELECT st_no count(*)
FROM   student course
WHERE
 student.st_no = stud_course.st_no
 stud_course.code = course.code
```

kim Ahmetden ders alıyor

```
SELECT st_no
FROM   student
WHERE
 student.st_no = stud_course.st_no
 stud_course.code = course.code
 course.instr_no = instructor.instr_no
 instructor.inst_name = Ahmet
```

Ahmetden ders alanı ver

```
SELECT st_no
FROM   student
WHERE
 student.st_no = stud_course.st_no
 stud_course.code = course.code
 course.instr_no = instructor.instr_no
 instructor.inst_name = Ahmet
```

Mehmetin aldığı dersleri ver

```
SELECT code
FROM   course
WHERE
 course.code = stud_course.code
 stud_course.st_no = student.st_no
 student.st_name = Mehmet
```

Mehmete ders veren hocaları göster

```
SELECT instr_no
FROM   instructor
WHERE
 instructor.instr_no = course.instr_no
 course.code = stud_course.code
 stud_course.st_no = student.st_no
 student.st_name = Mehmet
```

5ten fazla notları ver


```
SELECT grade st_no
FROM   stud_course stud_course
WHERE
 stud_course.grade > 5
```

50 ile 100 arasındaki notları ver

```
SELECT grade st_no
FROM   stud_course stud_course
WHERE
 stud_course.grade > 50
 stud_course.grade < 100
```

CMPE100 dersini veren kimdir

```
SELECT code
FROM   course
WHERE
 instructor.instr_no = course.instr_no
 course.c_name = CMPE100
```

Ahmetin aldığı ders nedir

```
SELECT code
FROM   course
WHERE
 course.code = stud_course.code
 stud_course.st_no = student.st_no
 student.st_name = Ahmet
```

CMPE100 dersinin hocası kimdir

```
SELECT instr_no
FROM   instructor
WHERE
 instructor.instr_no = course.instr_no
 course.c_name = CMPE100
```

iyi dersleri göster

```
SELECT code
FROM   course
WHERE
 course.c_name = CMPE100
```

Mehmetin iyi derslerini göster

```
SELECT code
FROM   course
WHERE
 course.code = stud_course.code
 course.c_name = CMPE100
 stud_course.st_no = student.st_no
 student.st_name = Mehmet
```

Ahmet kaç ders veriyor

```
SELECT count(*)
FROM   course
WHERE
 course.instr_no = instructor.instr_no
 instructor.inst_name = Ahmet
```

Ahmet kaç kimya dersi veriyor

```
SELECT count(*)
FROM   course
WHERE
 instructor.instr_no = course.instr_no
 instructor.inst_name = Ahmet
 course.c_name = kimya
```

kimyada kaç öğrenci var

```
SELECT count(*)
FROM   student
WHERE
 student.dept = kimya
```

kimya bölümünde kaç öğrenci var

```
SELECT count(*)
FROM   student
WHERE
 student.dept = kimya
```

Ahmet adlı hoca hangi dersleri veriyor

```
SELECT code
FROM   course
WHERE
 instructor.instr_no = course.instr_no
 instructor.inst_name = Ahmet
```

kimya alan öğrencilerin adlarını ver

```
SELECT st_name st_no
FROM   student student
WHERE
 student.st_no = stud_course.st_no
 stud_course.code = course.code
 course.c_name = kimya
```

Mehmet adlı öğrencinin numarasını ver

```
SELECT st_no
FROM   student
WHERE
 student.st_name = Mehmet
```

Mehmet adlı öğrencinin notlarını göster

```
SELECT grade st_no
FROM   stud_course stud_course
WHERE
 stud_course.st_no = student.st_no
 student.st_name = Mehmet
```

Mehmet adlı öğrencinin notlarının ortalamasını ver

```
SELECT avg(grade)
FROM   stud_course
WHERE
 stud_course.st_no = student.st_no
 student.st_name = Mehmet
```

CMPE100 derslerinin sayısını göster

```
SELECT count(*)
FROM   course
WHERE
 course.c_name = CMPE100
```

en yüksek notu göster

```
SELECT max(grade) st_no
FROM   stud_course stud_course
```

kimya hocalarını göster

```
SELECT instr_no
FROM   instructor
WHERE
 instructor.dept = kimya
```

kimya ve fizik hocalarını göster

```
SELECT instr_no
FROM   instructor
WHERE
 instructor.dept = kimya  OR instructor.dept = fizik
```

IV. A SYNTACTIC TURKISH PARSER FOR NLI

After having introduced the model, it is necessary to determine the types of sentences that natural processor component of model will deal with and define a formal representation of these linguistic data so that it can be processed on the computer. In the first part of this chapter, the types of sentences necessary in formulating database queries are intuitively introduced. Next, a grammar that consists of a collection of *rewrite rules* [1] for the formal representation of sentences are discussed. In the remainder of the chapter a simple algorithm for morphological parsing of nouns and syntactic parsing of sentences using the rules[1] defined in the previous chapter are discussed with examples of different sentence types.

4.1. Parsing Capabilities

Since the parser is to be used as a component of a natural language interface for databases it is restricted to queries about databases. Queries can be formulated as imperative sentences, or as affirmative sentences about the third person containing at least one interrogative (question word). Therefore, we restrict our grammar to these two types of sentences, and our model to the domain of the database. We also include *var/yok* 'there is/not' type of sentences and noun sentences with the suffix *-dir* (third person form of the verb *to be* where *to be* is the main verb). These four clause forms are the complete set of sentence types that may be used to formulate any database query. Thus, our parser can analyze the following types of sentences :

_ Imperative sentences, e.g. *CMPE100 dersinin hocasını* ***göster*** 'show the instructor of the course CMPE100'

[1] the word **rule** is used to mean **rewrite** rule hereafter

– Affirmative sentences about the third person containing at least one question word, e.g. ***kim*** *kimya dersini* ***veriyor*** 'who is giving the course CMPE100'

– Sentences with *var/yok* e.g. ***kaç*** *kimya dersi* ***var*** 'how many CMPE100 course(s) are there'

– Noun sentences with the suffix *-dir*, the so called copula, e.g. *CMPE100 dersinin hocası* ***kimdir*** 'who is the instructor of CMPE100'

It also allows sentences in negative form and passive construction of affirmative sentences.

The last constituent of the sentence determines the type of the sentences, the remainder of the sentence is called noun phrase. Noun phrases accepted by our parser consists of three different types :

(A) Proper nouns, e.g. Ahmet, CMPE100

(B) Question pronouns, e.g. *kim* 'who', *ne* 'what'

(C) Descriptive noun phrases,

Descriptive noun phrases are composed of a noun preceded by zero or more premodifiers. Permissible premodifiers in our current grammar are given in Table 4.1.

Table 4.1. Premodifiers

Type	**Example**	
noun modifier	*CMPE100* dersi	'the course *CMPE100* '
simple adjective	*yüksek* not	'a *high* grade'
question adjective	*hangi* öğrenci	' *which* student'
demonstrative adjective	*o* ders	' *that* course'
superlative	*en yüksek* not	' *the highest* grade'
possessive	*dersin* hocası	'the instructor *of the course*'

Multiple premodifiers are also accepted provided that the ordering of premodifiers is grammatically correct. Some examples are given below.

kaç CMPE100 dersi	' *how many CMPE100* course(s)'
o kimya dersi	' *that chemistry* course'
en yüksek CMPE100 notu	' *the highest CMPE100* grade'
Hocanın kaç iyi öğrencisi	' *how many good* student(s) of *the instructor*'

Any combination of more than one noun modifier or adjective may be used as a modifier provided that the last two premodifiers are separated by a conjunctive, e.g.

kaç CMPE100 ve MATH151 dersi '*how many CMPE100 and MATH151* course(s)'

en yüksek ve en düşük notlar '*the highest and the lowest* grades'

However, according to the correct ordering, the following constructions are not allowed:

CMPE100 yüksek not '*CMPE100 high* grade'

(noun modifier should immediately precede the head noun)

o kaç ders '*that how many* course(s)'

(a demonstrative adjective and a question adjective can not modify the some head noun)

Furthermore, some forms of comparative structures can be used as premodifiers, e.g.

5ten büyük(iyi, fazla) '(better, more) *greater than 5*'

5 ile 3 arasındaki notlar ' grades *between 5 and 3*

Our parser also accepts certain relative clause structures. The first type of relative clauses is the construction with a verb and participle suffixes *-en/-an*, e.g.

ders alan öğrenci 'the student *who takes a course*'

not vermeyen hoca 'the instructor *who does not give a grade*'

Second type of relative clauses is the construction with the suffix *-dik*, e.g.

*öğrencinin aldığı*dersler 'the courses *that the student takes*'

hocanın vermediği not 'the grade *that the instructor does not give*'

Relative clauses in Turkish can also be used as the **object** of a verb:

kimya dersi alanları(göster) '(show) *those who take* the chemistry course'

hocanın Ahmete verdiği(nedir) '(what is it) *that the instructor gives to Ahmet*

4.2. Formal Representation of Turkish Grammar Used in NLI

The major components of a computer system for analyzing the structure of a Turkish sentence are a set of categories and a lexicon in which each word is assigned to the categories, a grammar of Turkish which will specify the well formed sentence structures of the language and a parsing program [16]. Syntactic categories and the formalization of a subset of the Turkish grammar are discussed in the following paragraphs.

The basic elements of a sentence are verbs and noun phrases. In the lexicon, words are assigned to word categories, and rules are generalized terms of these categories. Therefore before proposing rules it is important to define these word categories. Words are categorized in order to simplify the interpretation of their semantic and syntactic functions. Words are traditionally classified according to their functions as noun, adjective, pronoun, verb, etc... Considering their functionality in communication for a database interface, we further classify words into subcategories. For instance adjectives are subclassified as question adjectives, demonstrative adjectives, etc... The list of the categories is given in Table 4.2. together with examples and thei notation used in rule definitions.

Table 4.2. Categories of Words

Category	Notation	Examples
Simple noun	noun	ders, hoca, öğrenci
Proper-noun	propernoun	Ahmet, Mehmet
Pronoun	pronoun	o
Question pronoun	qpronoun	kim, ne, kaç
Simple adjective	adj	iyi, kötü, zor
Question adjective	$qualadj_{ques}$	hangi, kaç
Demonstrative adjective	$qualadj_{dem}$	bu, o
Qualitative adjective	$qualadj_{qual}$	hiçbir, hiç, bütün
Quantitative adjective	adjquan	çok, az, fazla
Superlative	-	en
Conjunctive	conj	ve, veya, ile

The Turkish grammar that we intuitively use to differentiate between correct and incorrect sentences can be formalized as a phrase structure grammar which consists of a set of phrase structure rules each representing a labelled phrase to be matched against a sequence of constituents of the input sentence. Each phrase structure rule consists of a pattern name followed by an arrow and a string of symbols that make it up,

$$A \longrightarrow B\ C\ D$$

where the symbols on the right hand-side of the arrow will be substituted in place of the one on the left hand-side. Each symbol is either a word category or a pattern name.

A phrase structure rule specifies what a grammatical sentence can look like. Together with other rules specifying what other sentences can look like, all of these will form the grammar of Turkish. Therefore in formalizing the Turkish grammar, we must define correct sequence of word categories that will form different structure patterns. In addition to that, since the result of parsing will be used in representing the meaning of the sentence, parsing of the syntactic structures must be intermixed with the analysis of interrelation of words in order to recognize the function of each. The major aim of a parser is then not only to analyze the word sequence to check whether the sentence is grammatical or not, but to produce a parse tree representing the relations between words. In defining phrase structure rules, the notion of a relation must then incorporate with the categorical notion. The only relation between words can be a *modification*. Words can modify the verb or each other. A modifier can also be modified to produce a multi-layered structure. At each level there is a head noun and a sequence of modifiers. It is then sensible to break down the sentence into phrases each consisting of a head noun carrying the central assertion of the phrase and a sequence of modifiers, each of them providing some factual details to the information in the head. A sentence can then be thought of as a combination of phrases whose head modifies the head of the successive phrase and/or the verb. The head noun as well as modifiers may be a single word or a group of words, all of the same type. Therefore the main principle is to write *a rule* for

(1) each couple of noun phrases that modify each other

(2) each possible group of one or more words, all of the same type.

As mentioned above, the head of the phrase can modify one of the successive phrases and the verb at the same time, it is then necessary to distinguish levels of modifications. The grammar is thus arranged in a hierarchical structure and each level

defines a type of modification. The types of modifications can be classified into two groups as the modifiers of the verb and the modifiers of the successive phrase.

Considering the internal structure of a phrase , it is quite possible that several types of words modify the head noun. One simply can classify modifiers according to the position in which each occurs within another. We differentiate three different levels of modification depending on how close the type of a modifier can be to the head noun when this latter is modified with at least two different types of modifiers. These four levels are comparative modifiers, qualitative and quantitative modifiers, adjective modifiers, and noun modifiers.

In Turkish, a grammatically correct sentence with minimal constituents consists of a noun phrase followed by a verb, therefore the phrase structure of the **major sentence** in Turkish for a single sentence is

$$\mathbf{S} \longrightarrow \mathbf{Nph}_{\mathbf{caseM}}\ \mathbf{Vph}$$

where the structure of **Nph** and **Vph** must be independently specified.

Considering different types of sentences listed in Section 4.1. the **verb phrase, Vph** can be rewritten as

Vph $\longrightarrow$ **verb** |

verb + { *-iyor* | *-di* } |

var | *yok* |

(**noun** | **adj** | **verb** + *-en*) + *-dir*

Our grammar analyzes noun phrases in stages. The analysis of noun phrases is classified into three different noun phrase levels. These levels are arranged in a hierarchical manner according to the order of suffix combination of the head. This is provided by defining a set of rules for each level of modification, namely **Nph, Nph1, NF** where each is defined in terms of the other.

The highest level for a noun phrase is **Nph**. It describes possible combinations of a noun with one of the case markers, namely nominative, accusative, dative, locative, ablative abbreviated as **cases, obj, dat, loc** and **abl** respectively. The case markers indicate the role of the noun phrase designated by the verb in which it occurs. We consider five different roles corresponding to the above listed cases. They are

subject/agent, object, destination, location and source. The constituent of a simple **noun phrase, Nph** can be

- a noun phrase whose head has a case marker
- two or more noun phrases, each with a different case marker
- a null value.

Hence, **Nph** can be formalized recursively to handle an infinite number of noun phrases as

$$\mathbf{Nph}_{\mathbf{caseM}} \longrightarrow \mathbf{Nph}_{\mathbf{caseN}}\ \mathbf{Nph1}_{\mathbf{caseM}} \mid \mathbf{Nph1}_{\mathbf{caseM}}$$

The next level for a noun phrase, **Nph**, deals with the highest level of modification that may occur between two noun phrases, namely **possessive relation**, a noun phrase whose head noun has a genitive suffix *-in* on it modifying its successor noun phrase whose head noun has a possessive suffix *-i* on it [17] . The structure is represented by the following rule :

$$\mathbf{Nph1}_{\mathbf{\{caseM/gen\}}} \longrightarrow \mathbf{Nph1}_{\mathbf{gen}}\ \mathbf{Nph1}_{\mathbf{(caseM/gen)+poss}}$$

This rule states the requirement of a noun phrases with genitive property when one with possessive property has been parsed. Note that this rule allows recursive applications, it can thus parse nested possessive relations.

Examples :

hocanın dersi 'instructor's course / course of the instructor'

hocanın dersinin öğrencisi 'students of the instructor's course'

Combination of two noun phrases with a conjunctive or disjunctive must be handled at this level. Combination of two **Nph1**s yields another **Nph1** as defined in the following rule :

$$\mathbf{Nph1}_{\mathbf{(caseM/gen)+(poss)}} \longrightarrow$$

$$\mathbf{Nph1}_{\mathbf{(caseM/gen)+(poss)}}\ \mathbf{conj}\ \mathbf{Nph1}_{\mathbf{(caseM/gen)+(poss)}}$$

Example :

hocayı ve dersi 'the instructor and the course'

The three rules described above deals with the relations between noun phrases and the verb. The remainder of this section is devoted to the definition of the internal structure of noun phrases. Just to remind, the internal structure of a noun phrase consists of a head noun or a head noun group preceded by a string of modifiers, the so called premodifiers. Considering the difference in modifiability of different nounlike words , we introduce the third level **NF**. The modified head noun together with its modifiers are considered as a noun phrase simply by adding the following rule into the grammar

$$\mathbf{Nph1}_{\mathbf{\{caseM/gen\}+(poss)}} \longrightarrow \mathbf{NF}_{\mathbf{\{caseM/gen\}+(poss)}}$$

Basic constituents of a noun phrase at the third level **NF** include

- pronouns
- question pronouns
- proper nouns or a group of proper nouns
- noun or a group of nouns.

Although they are all nounlike words, only nouns can be preceded by modifiers. This leads us to introduce four different options for **NF** as given below

$$\mathbf{NF}_{\mathbf{\{caseM \mid gen\}+(poss)}} \longrightarrow$$

$$\mathbf{pronoun}_{\mathbf{\{caseM \mid gen\}+(poss)}} \mid$$

$$\mathbf{qpronoun}_{\mathbf{\{caseM \mid gen\}+(poss)}} \mid$$

$$\mathbf{Ng2}_{\mathbf{\{caseM \mid gen\}}} \mid$$

$$\mathbf{\{ASF \mid AFS \mid (CS)\ (QF)\ (AF)\}\ Np}_{\mathbf{\{caseM \mid gen\}+(poss)}}$$

The first two options state that occurrence of a single pronoun or question pronouns is considered as a noun phrase. The occurrence of several proper nouns is grouped together under the rule **Ng2**. These three options define the set of nounlike words that can not take any premodifiers. The last one covers five different types of modifiers which may precede the noun or the noun group. They are called premodifiers. The structure of each modifier category is explained in detail later in this section. Here is examined the position in which each occurs within another. The grammar is extended by putting an option for every possible correct order of these premodifiers. There are in fact three options since some of them are grouped in the same option. These three

different options are adjective modifiers in superlative form **ASF**, relative clauses in adjective function **AFS** occuring singly and a sequence consisting of comparative modifiers **CS**, followed by qualitative and quantitative modifiers **QF** and adjective modifiers **AF**. Considering the variety of possible combinations of premodifiers in the last option, some of them may be absent in certain grammatical constructions but the correct order is still preserved. Thus, we have *kaç iyi* ders ' how many good course' **(QF, AF, NP)**, not *iyi kaç ders* 'good how many course' **(AF, QF, NP)**, *5ten büyük iyi not* 'good grades greater than 5 '**(CS, AF, NP)**, but not *iyi 5ten büyük not* 'good greater than 5 grade ' **(CS, AF, NP)** and so forth. The brace notation indicates optionality of appearance of a constituent in that position.

Examples :

5 ten büyük not,	'the grade (that is) greater than 5',
en iyi ders,	' the best course',
ders veren hoca ,	' professors giving a course'
kaç iyi ders,	' how many good course'

Ahmetin notundan fazla kaç (iyi) not 'how many (good) grade(s) greater than Ahmet's grade (are there)'

Possessive compounds form the lowest level of modification. This construction is like possessive relation; the difference is that in compounds genitive suffix *-in* is omitted from the first noun although possessive suffix is still attached to the second noun. Nothing may exist between the two members of a possessive compound. Any adjective or another modifier must come before the entire group. They are actually two nouns that function together to make a single unit, therefore they are considered in a separate rule **Np**. The rule handling possessive compounds defines the last level of internal modification of noun phrases, and it has the following form:

$$\mathbf{Np}_{\{\mathbf{caseM} \mid \mathbf{gen}\}} \longrightarrow \mathbf{Ns}\ \mathbf{Ng1}_{\{\mathbf{caseM} \mid \mathbf{gen}\}+\mathbf{poss}}$$

Ng1 denotes a string of nouns and it is preceded by **Ns**, which is another string of one or more nouns.

Examples :

CMPE100 dersi	'CMPE100 course'
fizik dersi	'the physics course'

A head can simply consists of a single noun. A nesting of an infinite number of nouns can be in the head provided that the last two are separated by a conjunctive or a disjunctive. As the elements in both sides of the conjunctive as well as the disjunctive must be identical, parsing of the nested noun necessitates two different sets of rules. These two sets of rules for nouns and proper nouns are **Ng1, Ngn1** and **Ng2, Ngn2,** respectively. They have recursive patterns and they are identical in structure.

$$\mathbf{Ng1}_{\mathbf{\{caseM \mid gen\}+(poss)}} \longrightarrow \mathbf{noun}_{\mathbf{\{caseM \mid gen\}+(poss)}} \mid \mathbf{Ngn1}_{\mathbf{\{caseM \mid gen\}+(poss)}}\ \mathbf{conj}\ \mathbf{noun}_{\mathbf{\{caseM \mid gen\}+(poss)}}$$

$$\mathbf{Ng2}_{\mathbf{\{caseM \mid gen\}}} \longrightarrow \mathbf{propernoun}_{\mathbf{\{caseM \mid gen\}}} \mid \mathbf{Ngn2}_{\mathbf{\{caseM \mid gen\}}}\ \mathbf{conj}\ \mathbf{propernoun}_{\mathbf{\{caseM \mid gen\}}}$$

$$\mathbf{Ngn1}_{\mathbf{\{caseM \mid gen\}+(poss)}} \longrightarrow \mathbf{noun}_{\mathbf{\{caseM \mid gen\}+(poss)}} \mid \mathbf{Ngn1}_{\mathbf{\{caseM \mid gen\}+(poss)}}\ \mathbf{noun}_{\mathbf{\{caseM \mid gen\}+(poss)}}$$

$$\mathbf{Ngn2}_{\mathbf{\{caseM \mid gen\}}} \longrightarrow \mathbf{propernoun}_{\mathbf{\{caseM \mid gen\}}} \mid \mathbf{Ngn2}_{\mathbf{\{caseM \mid gen\}}}\ \mathbf{propernoun}_{\mathbf{\{caseM \mid gen\}}}$$

Examples :

Ahmet, Ahmetin 'Ahmet's', *Ahmet ve Mehmet*,' Ahmet and Mehmet' *Ahmetin ve Mehmetin* 'Ahmet's and Mehmet's'

not ders ve hoca 'grade course and instructor'

A special case of the rules **Ng1** and **Ngn1** is **Ns** and **Nsn** where the noun constituents can have zero case marker on them, the zero case marker is denoted as **caseø.**

$$\mathbf{Ns} \longrightarrow \mathbf{noun}_{\mathbf{case\emptyset}} \mid \mathbf{Nsn}\ \mathbf{conj}\ \mathbf{noun}_{\mathbf{case\emptyset}} \mid []$$

$$\mathbf{Nsn} \longrightarrow \mathbf{noun}_{\mathbf{case\emptyset}} \mid \mathbf{Nsn}\ \mathbf{noun}_{\mathbf{case\emptyset}}$$

Examples :

CMPE100 dersi 'the course *CMPE100*'

matematik fizik ve kimya dersleri ' *mathematics physics and chemistry* courses'

There are four different types of premodifiers; simple adjectives , adjectives in superlative form, qualitative and question adjectives, comparative sentences. The rules for constructing each of these premodifier types are given in the following paragraphs.

Adjective modifiers are strings of adjectives. The following two rules **AF** and **ADF** cover adjectives occuring singly or as an indefinitely long sequence provided that the last two are separated by a conjunctive;

AF $\longrightarrow$ **adj | ADF conj AF**

ADF $\longrightarrow$ **adj | ADF adj**

Examples :

iyi ders, 'a *good* course'

zor ve iyi ders, 'a *difficult and good* course'

Adjectives in superlative form are compounds consisting of the word *en* followed by an adjective. Definition of the rule permitting only occurrence of a single compound and two compounds is represented as

ASF $\longrightarrow$ *en* **adj | ASF conj ASF**

Examples :

en iyi ders, '*the best* course'

en zor ve en iyi ders '*the most difficult and the best* course'

Question adjectives, demonstrative adjectives, and **quantitative adjectives** can not cooccur in the same noun phrase. Therefore they are collected in the single category **qualadj**. They may also be preceded by a superlative compound which consists of the word *en* followed by a quantitative adjective;

QF $\longrightarrow$ **(SupF) qualadj**
SupF $\longrightarrow$ *en* **adjquan**

Examples :

kaç ders, *bu* ders, '*how many* course, *this* course'

en çok kaç ders '*at most how many* course(s) '

Comparative sentences CS are embedded noun phrases that modify the head noun. Comparative forms are rich structurally. One way to handle them is to see similarities between several possible constructions. Comparative sentences can be obtained by adjoining a noun phrase to any word which has a comparative meaning. Words which have comparative meaning include all the members of the category **adjquan**, and some of the elements in the category **adj**. In addition to them we explicitly state some words that can occur in comparative constructions. They are *eşit, kadar, arasındaki*. We also treat any sentence consisting of a noun with the derivational suffix *-li* on it that adjoins a noun phrase as a comparative sentence ;

CS $\longrightarrow$ **CS1 | CS1 conj CS1**

CS1 $\longrightarrow$ $\mathbf{Nph_{abl}}$ **adjquan |**

$\mathbf{Nph_{abl}}$ **adj |**

$\mathbf{Nph_{\{dat\ |\ case0\}}}$ **(** *eşit* | *kadar* | *arasındaki* **) |**

$\mathbf{Nph_{case0}}$ **noun +** *-li*

Examples :

Ahmetin notuna eşit not	'grade (that is) *equal to Ahmet's grade*
3 ile 5 arasındaki notlar	'grades *between 3 and 5*
5ten fazla ve 10dan az notlar	'grades (which are) *grater than 5* and *less than 10*
Ahmet adlı öğrenci	'the student *named Ahmet*

The major capability of the parser is that it can process embedded constructions with participles suffixes *-en/-dik* which correspond to relative clauses in English. In its function, a relative clause can be the modifier of a noun when it adjoins a noun or it can modify the main verb of the sentence provided that the verb of the embedded clause has a case marker other than **caseø**. Thus, we differentiate two types of embedded clauses that we name as **adjective clauses** and **noun clauses**.

Adjective clauses are embedded sentences with the case marker ø which modify a noun, such as

ders veren hoca, ' instructor *who gives the course*'

öğrencinin aldığı ders ' the course *that the student takes*'

Noun clauses are embedded sentences with non-null case suffix modifying a verb. For simplicity, we only consider noun clauses as the subject or the object of the sentence, such as

ders vereni göster ' show *the one who gives the course*'

öğrencinin aldığı nedir ' what is *it that the student takes*'

The relative clause can have the characteristics of an adjective or a noun depending on the case marker of the verb. In addition to that, the relative clause itself is a sentence. It contains a subject, a verb and all the characteristics that identify a sentence. We can explicitly state this fact by expanding the phrase structure rules of **NF** and **AF**. Addition of two more rules allows the parser to process these new consistent structures;

$$\mathbf{AFS} \longrightarrow \mathbf{Nph}_{\mathbf{caseM}}\ \mathbf{verb}\ +\ \{-en|-dik\}_{\mathbf{caseø}}$$

$$\mathbf{NF} \longrightarrow \mathbf{Nph}_{\mathbf{caseM}}\ \mathbf{verb}\ +\ \{-en|-dik\}_{\mathbf{caseM}}$$

4.3. Two Different Parsings Used for the Analysis of Turkish Sentences

Functional properties of nouns can only be determined from their suffixes. Hence syntactical analysis of a sentence requires morphological parsing of its nouns. Both of these analysis are quite difficult in Turkish. In this study, the attempt is to solve only a very limited part of these two problems.

4.3.1. Morphological Parsing

One major difference between Turkish and English is in the morphology of words. Turkish is a suffixing language. Functions are assigned to nouns by suffixation. Therefore, a syntactic parsing necessitates morphological analysis of nounlike words for determining their syntactic functions. Thus, we discard the derivational suffixes in this analysis and consider only three types of inflectional suffixes, i.e. case suffixes *-i, -e, -de, -den,* the possessive and genitive suffixes *-i* and *-in* respectively and the plural suffix *-ler.* Regarding the lexical representation of nouns, forms involving derivational suffixes are listed in the dictionary while those involving inflectional suffixes are not. In Turkish, there are rules regulating the order of suffixes. For instance, the genitive suffix always follows the possessive suffix or any other case suffix. Suffix order in Turkish is as diagrammed in Figure 4.1.

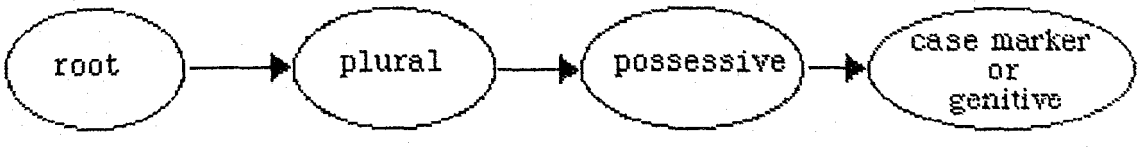


Figure 4.1. Suffix Order in Turkish

Considering two possible approaches to the actual analysis of words[18,19], namely affix-stripping and root driven analyses, the former is more appropriate since the morphological parsing is trivial in our case for the following two reasons. One reason is that syntactic parsing requires recognition of a few suffixes at the end of nounlike words. In addition to that, there is a very limited number of lexical entries. In the affix-stripping approach, parsing processes by stripping suffixes off the word and attempting to look up the remainder in the lexicon. Our parser proceeds from right to left by stripping letters. A decision tree is used to split the nouns into their root and suffix constituents. The decision algorithm is given in terms of a decision tree in Figure 4.2.

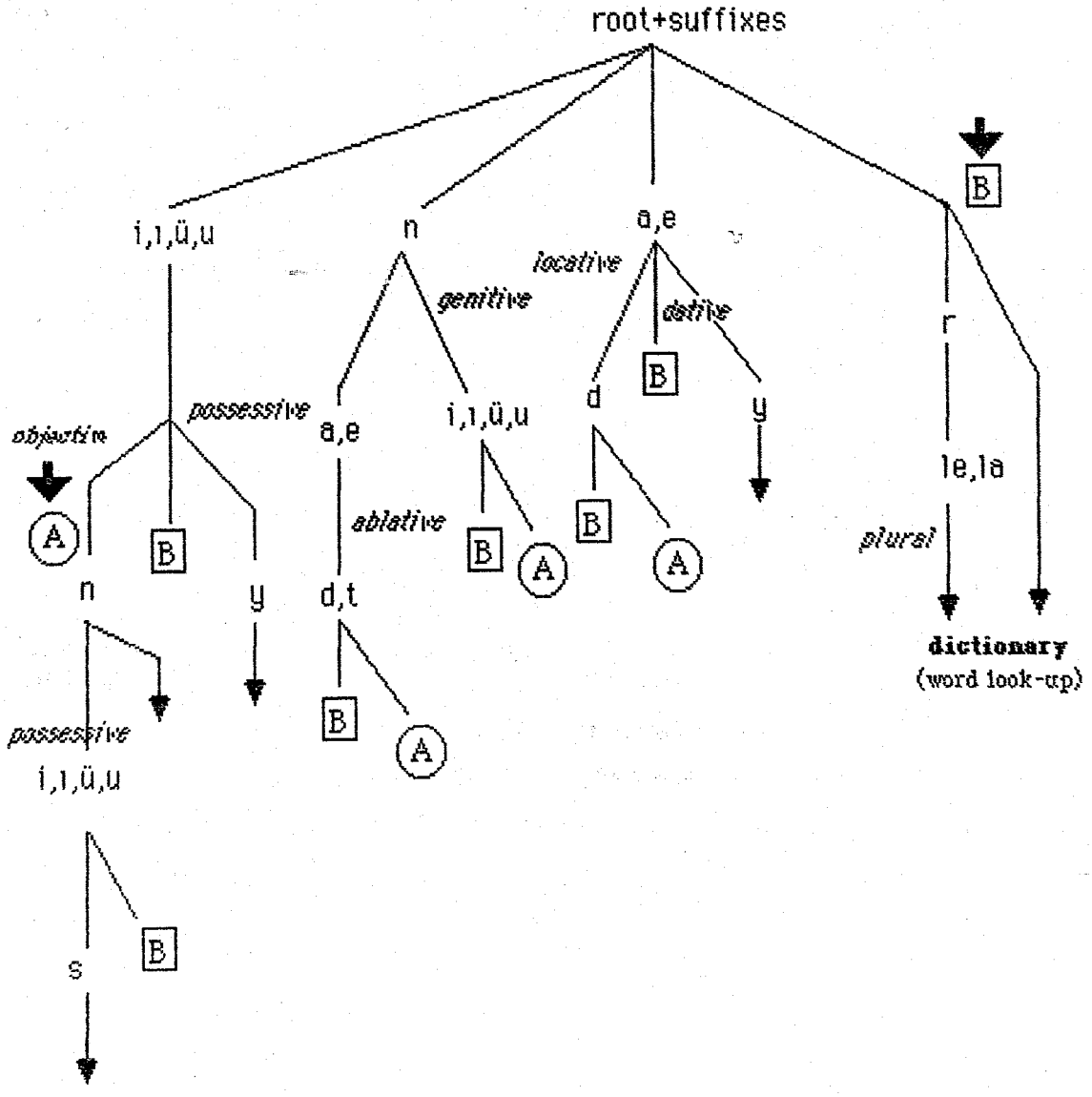


Figure 4.2. Decision Tree for Suffix Elimination

Taking into account letter alterations like (p,ç,t,k) to (b,c,d,ğ), and buffer consonant insertion *-y-*, letters of the noun are processed from right to left until a letter which can not possibly be part of a possible suffix is reached. Then the remaining part is searched in the dictionary as the root word. The decision tree consists of three levels, each level corresponding to one type of suffix as indicated on each branch. A correctly spelled noun whose final letter is not one of ["i","ı","u","ü","n","a","e","r"] must exist in root form and is directly searched in the dictionary, represented with a downward arrow →. Depending on the last letter of the word, suffix stripping process branches to different nodes of the tree. Each last node of the tree is a dictionary lookup. For instance a noun with the last letter *r* will only be checked for plural property. Going down in the tree, the parser attempts to match the last two letters of the word with *le* or *la* (remaining part of the plural suffix *le+r* or *la+r*). When the match holds the rest of the noun must be in the root form, and it is looked up in the dictionary. When the match does not hold, the parser backtracks, concatenates the two splitted parts of the word and looks in the dictionary once more.

As another example the noun *ders+i+ni* 'his course' will branch to the right nodes till the last one and the left node in the last decision point will be followed since a noun such as *der* does not exist in Turkish; dictionary lookup for the root form *ders* 'course' will be satisfied. Our decision tree does not necessitate more than two dictionary lookups for any noun in Turkish and the average is around one lookup per noun.

4.3.2. Syntactic Parsing

The top-down parsing method is used in processing the natural language queries. The parsing is represented with a parse tree. Construction of the tree is guided by the types of the modification which can be determined from the type of the word or from its suffixes if it is a noun. Before going into the detail of how parsing proceeds, let us explain the order in which rules are expanded since it is different than the usual way. Since Turkish is a **SOV** (**S**ubject **O**bject **V**erb) ordered language, the verb constituent is at the end of any correct Turkish sentence. In addition to that, the head noun and the verb of an embedded clause are always at the end of each phrase. The last constituent holds lots of information about the structure of the sentence and of each phrase. Therefore, it is more suitable to start parsing from the last constituent of the sentence. In the previous section, the right-hand side of the rule defines the ordering of sentence

constituents from left to right. During the parsing process, expansion of the rule begins from its last constituent and continues towards the left.

We start with the major sentence **S** and the tree is expanded by continually replacing the right-hand side of the current rule with one of the possible left-hand sides until we reach the first constituent of the phrase matched as the first element of the rule. Information gained in the morphological parser has been propagated up on the tree and it confirms the correctness of the selection.

Using the rules in our grammar given in Appendix A and the lexicon from Appendix D, we can illustrate how top-down parser constructs a structural representation of the following simple sentence

kim kimya veriyor
who chemistry is-giving
who is giving chemistry

Parsing begins by expanding the major sentence **S** which will be replaced with **Nph Vph** where **Vph** is further expanded to **verb + {iyor}**,which can be matched with the last constituent of the input sentence, *ver + iyor* since *ver* is in the category **verb**. Having found **Vph**, now our parser must complete the sequence **Nph Vph** in the definition of **S**, so it looks for **Nph**, which is replaced by

$$\mathbf{Nph}_{\mathbf{caseM}}\ \mathbf{Nph1}_{\mathbf{caseN}}$$

and continuing in the same manner the following replacements are done for **Nph1**

$$\mathbf{Nph1}_{\mathbf{caseN}} \longrightarrow \mathbf{NF}_{\mathbf{caseN}}$$

$$\mathbf{NF}_{\mathbf{caseN}} \longrightarrow \mathbf{Np}_{\mathbf{caseN}}$$

$$\mathbf{Np}_{\mathbf{caseN}} \longrightarrow \mathbf{Ng1}_{\mathbf{caseN}}$$

Finally, the following replacement is done.

$$\mathbf{Ng1}_{0} \longrightarrow \mathbf{noun\ (\ kimya\ )}$$

Noun is a terminal symbol, so it may be matched against the word *kimya*.

Nph in the second rule is again replaced by **Nph1**. *Kim* 'who' having the category **qpronoun** is parsed by rewritting **NF** with

$$\mathbf{NF}_{0} \longrightarrow \mathbf{qpronoun\ (\ kim\ )}$$

Figure 4.3. shows the parse tree for our example sentence.

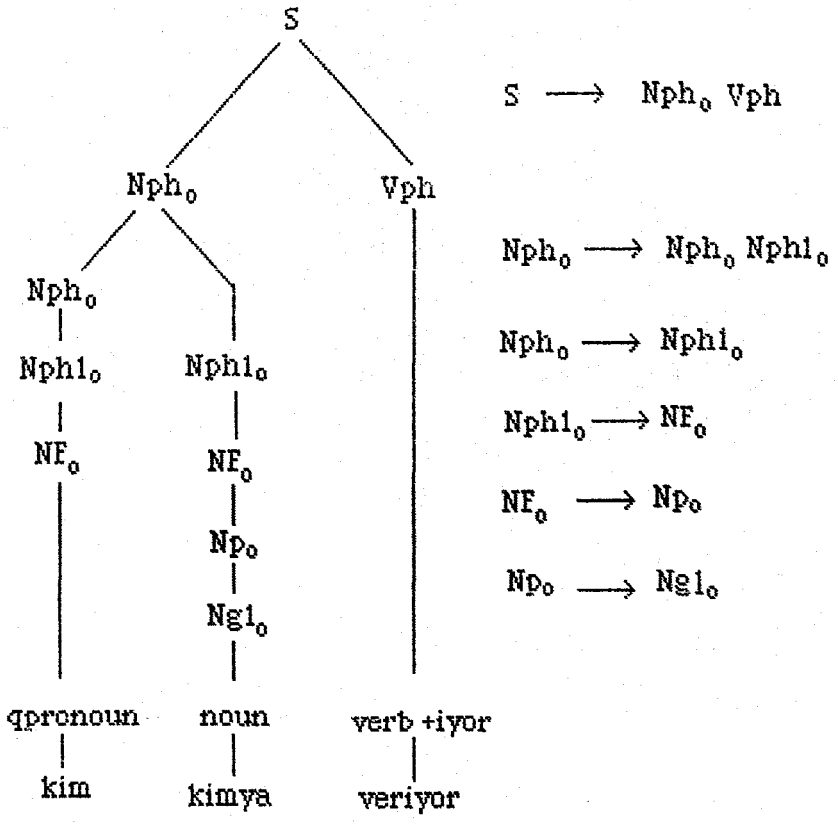


Figure 4.3. Parse tree for *kim kimya veriyor* 'who is giving chemistry'

Semantically related words are grouped on the same branch of the tree. Levels correspond to the modification levels described in the previous sections. As another example consider the analysis of the following sentence which yields the parse tree shown in Figure 4.4. Notice that each branch of the tree shows different types of modification.

hangi hoca kaç iyi kimya ve fizik notu veriyor.
which instructor how-many good chemistry and physics grade(s) is-giving

which instructor is giving how many good chemistry and physics grades

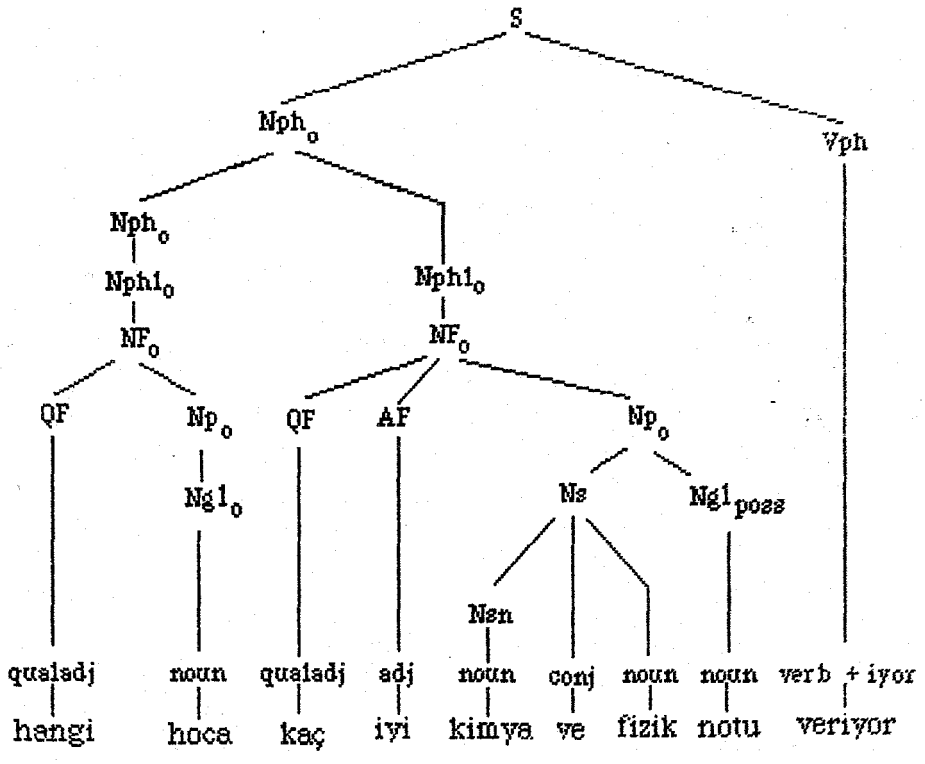


Figure 4.4. Parse tree of *hangi hoca kaç iyi kimya ve fizik notu veriyor* 'which instructor is giving how many chemistry and physics grade(s)'

It begins parsing by expanding **S** into **Nph** and **Vph**. **Vph** is matched to *ver+iyor* as explained in the previous example. The noun phrase **Nph** is replaced by **Nph Nph1**, and **Nph1** is rewritten by **NF**. **NF** is in turn replaced by **CS QF AF Np** where **Np** parses the noun group **Ng1**, together with their relevant noun qualifiers **Ns**, which must immediately precede the noun group. **AF** and **QF** parse a group of one or more adjective qualifiers and a quantity qualifier respectively. **CS** is skipped since it is optional and no comparative sentence exists. Similarly the last two constituents of the sentence make another noun phrase **Nph1** which is obtained by replacing the remaining **Nph** by **Nph1**.

As our parser proceeds from left to right, the verb of the embedded sentence can be easily recognized since it has a special form provided by the suffix *-en/-dik*. The embedded noun phrase whose verb is either in adjective function, denoted as **AFS**, or in noun function, denoted as **NF**, is parsed and the embedded noun phrase is considered as the modifier of the head noun. Figure 4.5. gives an example where the embedded noun phrase is *Ahmete ders veren* 'who teaches Ahmet' is the modifier of the noun *hocaları* 'instructors'.

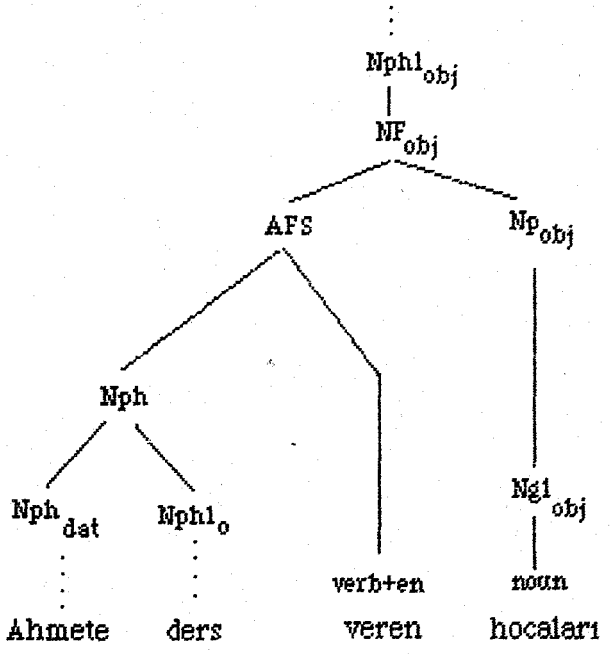


Figure 4.5. Parse tree for the embedded adjective clause *Ahmete ders veren hocaları* 'the instructors who teach Ahmet'

The following three examples given in Figures 4.6. 4.7. and 4.8. show the parse tree constructions for different types of sentences. The first example includes three modification levels that our parser distinguishes i.e. possessive relation, adjective modifiers and possessive compounds. Consider the following sentence which involves all of them,

iyi kimya hocalarının derslerini göster
good chemistry instructors courses show
show the courses of the good chemistry instructors

The noun modifier *kimya* 'chemistry' modifies the noun group that consists of the single noun *hoca* 'instructor', the possessive compound is parsed in **Npposs**. They form a noun form **NF** with the simple adjective modifier *iyi* 'good'. The modified noun phrase **Nphlgen** with the genitive property on the head noun is further combined with the noun phrase **Nphlposs+obj** which parsed the noun *derslerini* 'courses', into **Nphlobj**. **Nphlobj** constitutes part of the **Nphobj** which is in turn part of the the major sentence.

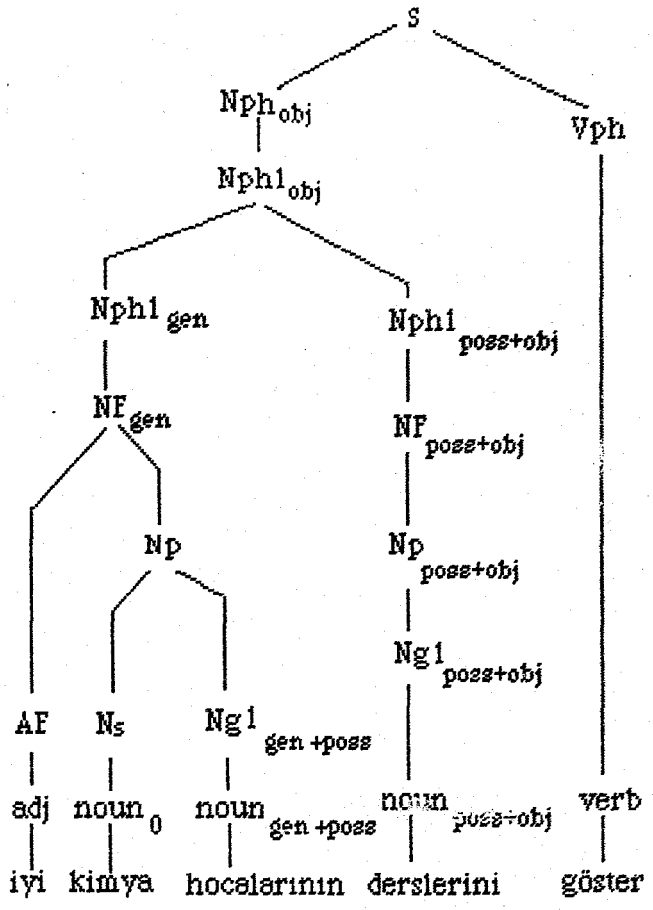


Figure 4.6. Parse tree for *iyi kimya hocalarının derslerini göster* 'show the courses of the good chemistry instructors'

In Figure 4.7. there is an example of a sentence that consists of an embedded noun phrase which includes a comparative sentence.

Ahmetin notundan fazla not alan öğrencileri göster
Ahmet's grade greater-than grade who-get students show

show the students who get a grade greater than Ahmet's grade

The comparative noun phrase *Ahmetin notundan fazla* 'greater than Ahmet's grade' is parsed in **CS**. It is the modifier of the noun *not* 'grade'. They form a noun phrase which then in turn forms an embedded clause with the adjective in verb form **ASF**, *alan* 'who gets'. The embedded clause is, in fact, the modifier for the noun *öğrenci*, and they form the noun phrase **Nph1**.

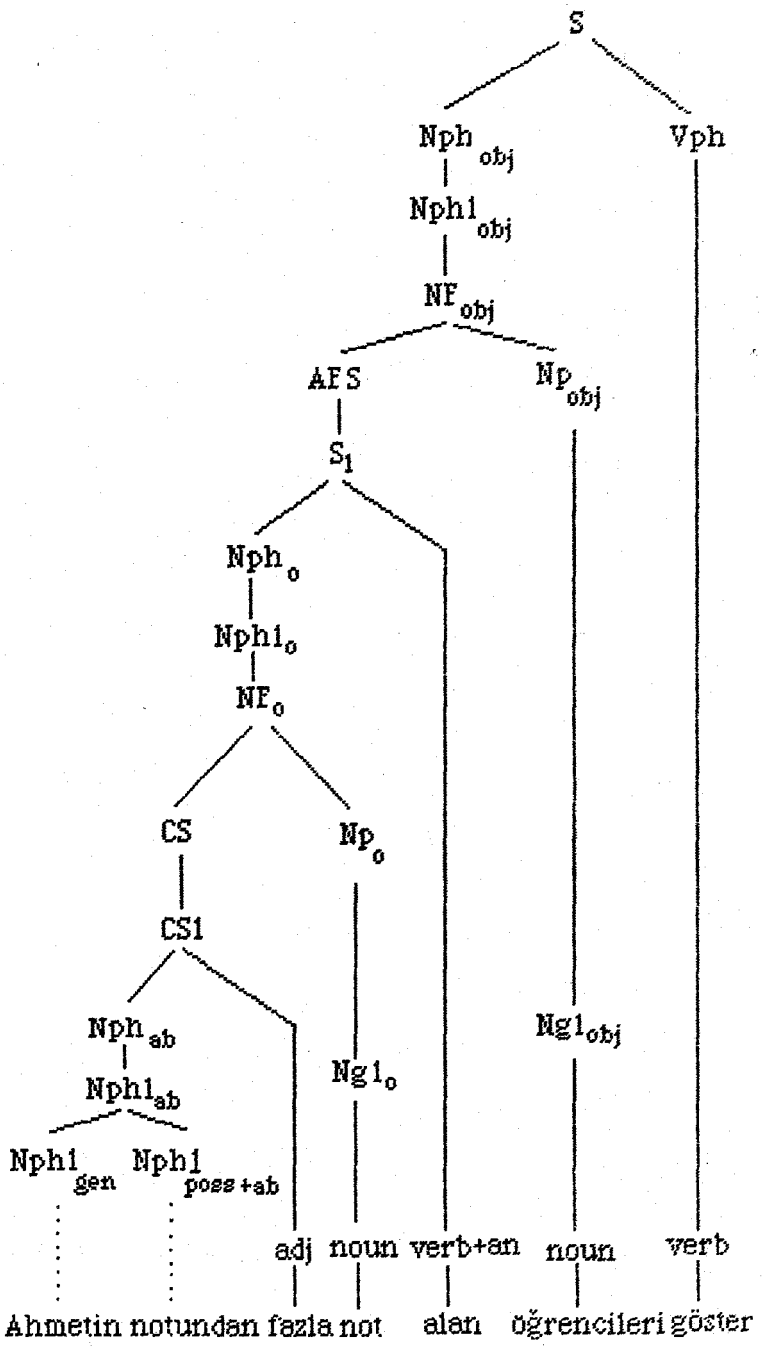


Figure 4.7. Parse tree for *Ahmetin notundan fazla not alan ögrencileri göster* 'show the students who get a grade greater than Ahmet's grade'

In Figure 4.8, the last example shows the parse tree of the sentence that consists of a group of noun modifiers and a question adjective .

En çok kaç matematik fizik ve kimya notu var
At most how many mathematics physics and chemistry grades are-there

At most how many mathematics physics and chemistry grades are there

In this example a sequence of noun modifiers is parsed in **Ns** by expanding it with the option **Nsn**, and **QF** parses the question adjective *kaç* 'how many' together with intensifier compound *en çok* 'at most'.

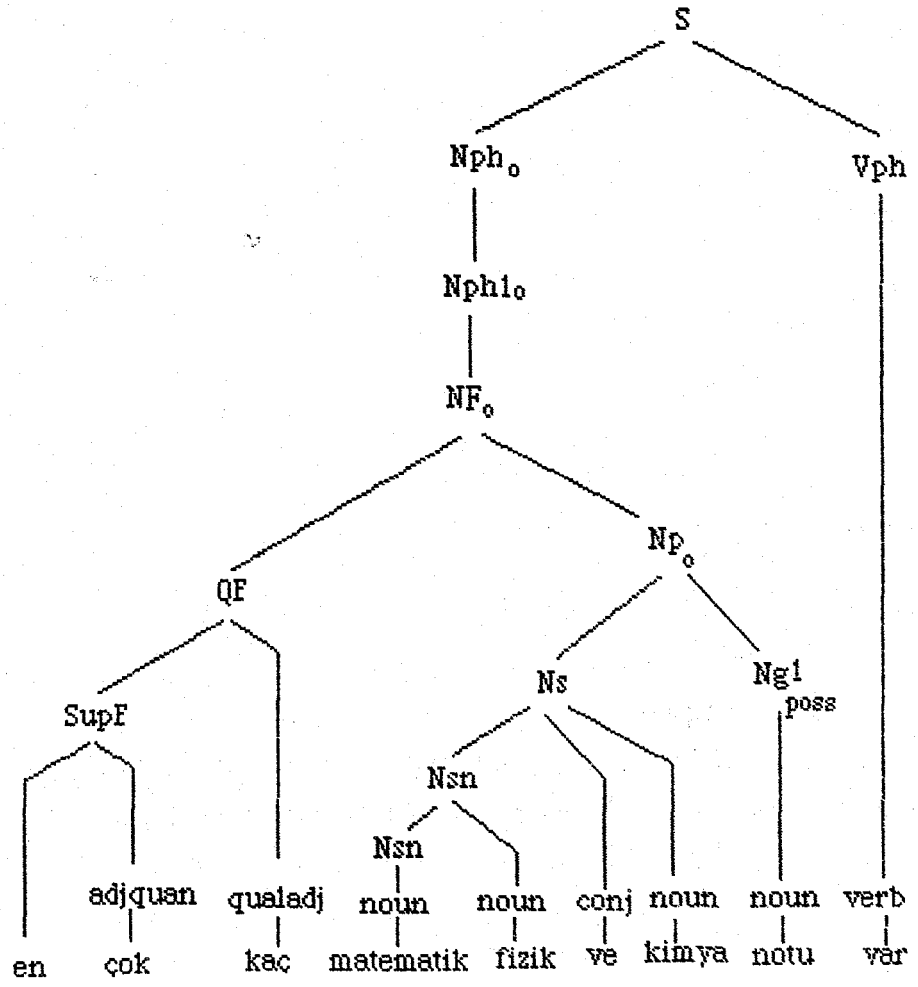


Figure 4.8. Parse tree for *en çok kaç matematik fizik ve kimya notu var* 'At most, how many mathematics physics and chemistry grade(s) are there'

V. QUERY UNDERSTANDING AND DECLARATIVE QUERY GENERATION

The previous chapter described a formalism for different types of NLI queries which is generally based on the modification relation between constituents. Considering the big difference between natural language sentences and expressions in a declarative query language, it is quite difficult to directly translate the NLI query into its declarative equivalent. Therefore, these queries will be represented in a meaning representation language, namely Wallace's D&Q. In the first two sections of this chapter, we briefly summarize the formal D&Q notation as given by Wallace and some syntactical changes made to it. The next section concentrates on the interpretation of different modification relations discussed in the previous chapter in D&Q. Finally, conversion to a database dependent declarative language is discussed.

5.1. Wallace's D & Q Notation

In most of the early natural language database interfaces, parsing and meaning representation are handled by two different modules. In our model meaning representation is built up while parsing natural language queries. An extended form of D&Q meaning representation language is found to be appropriate to represent formal queries. D&Q is powerful enough to represent every valid query and has a feature that is not normally available in declarative query languages but necessary in representing the meaning of natural language queries. They are the quantifier hierarchy feature, formal determiners.

D&Q representation divides sentences into referring phrases, and qualifying phrases which are called Descriptions and Qualifiers, in other words D and Q respectively. The simplest case for a description in Wallace's notation is a simple constant, such as AHMET, CMPE100.

On the other hand, qualifiers are predicate calculus formulas where the predicate represents the relation name, and predicate terms represent selections. A selection is a formula composed of an attribute name, a comparison operator, and a value. For instance, a qualifier for the relation *student* may look like

student (name = 'Ahmet', dept = 'chemistry')

After having transformed the qualifier into the following Prolog goal

student(_ , 'Ahmet', 'chemistry', _).

the later can be evaluated on a database of Prolog facts with each fact representing a tuple of the relation.

To return a value from a simple qualifier , a variable can be used. For instance

student (name = **X**, dept = 'chemistry')

will unify **X** to the names of the students whose department is *chemistry*.

To represent each alternative meaning qualifiers can be combined with logical operators *not*, *&* (equivalent of "and") and *or*. For instance

the name of the student whose department is ***not*** *chemistry* and *the CMPE100 course and the students whose name is Ahmet* will be represented in D&Q respectively as

not (student (name = X,dept = ' chemistry')),

course (code = 'CMPE100') & student (name = 'Ahmet')

Other than being a simple constant, a description can also include a variable. Its syntax is

determiner-count-**qual**(‹ variable ›,‹qualifier›)

The determiner can be *the*, *any* or *what* depending on whether the tuple in question is defined or not. Count is an integer number referring to the number of tuples. *Any* or *what* refer to any group of tuples with one or more elements, where in case of *what* referents are undetermined. *The* creates a definite description which refers to a unique group of tuples.

For instance

the student number of the student named Ahmet

will be expressed as

the-1-qual(X, student(num = X, name = Ahmet)).

This description refers to the tuple in the relation *student* where name has the value Ahmet.

A description can be joined with another description as well as with a qualifier. Selections of a qualifier may contain a variable, so the selection becomes a function of that variable. Descriptions and qualifiers can then be joined via these variables to represent complex queries. The syntax of two possible combinations is given below.

A qualifier can be defined as

‹description› **is qual** (‹ variable ›,‹qualifier›)

and the definition of the description is

‹description› **is funct**(‹ variable ›, ‹description›)

5.2. Changes Made in the Original Syntax of D&Q

The D&Q syntax given in Appendix B.1. is used for meaning representation of natural language queries. In addition to that, D&Q formulas are matched against the database to retrieve tuples satisfying it. In our model, the syntax of D&Q will be used to express the meaning of the natural language queries. Then, the formula will be converted into a declarative database language. It is necessary to make some changes in the original syntax of D&Q so that the syntax of the representation language is closer to the syntax of the declarative database languages:

1) We allow the usage of an attribute more than once in a qualifier, i.e.

gradelist(grade =X, grade >5)

because we need such a construction. Consider the phrase

Grades greater than 5

which can be represented in our syntax as

any-N-qual (X, gradelist(grade=X, grade >5))

The equivalent in Wallace's notation is gradelist(grade > 5) which is transformed to a Prolog clause. However, it is difficult to convert such a construction into a declarative database query since it does not include any variable.

2) A selection in Wallace's notation has a strict form, which is

‹attribute› ‹ comparison › ‹ variable›

Therefore a selection like *grade > 5* where the last component is not a variable as in the above example, or *X >5*, where first component is a variable, are not allowed. In our model we change the D&Q syntax such that it accepts the above structures. That is, the syntax will accept either constant values or variables for either side of the comparison operator.

3) Consider the *student* relation. Suppose we want to list student names and student numbers. This is not possible in D&Q as described by Wallace unless we construct two descriptions on the same relation with different return values, i.e.

what-N-qual(X,student(name=X))
what-N-qual(Y,student(num=Y)).

We may combine these two descriptions into a single one by modifying the structure of the description. We can add a field **display field,** as we call it, which contains a list of variables whose values will be displayed. In our new syntax the above description will look like

what-N-qual(X,[X,Y],student(name=X,num=Y)).

whose equivalent in a declarative database language is

SELECT student.name , student. num

4) In the interpretation of the formal parameters, *any* and *what* refer to any group of tuples with one or more elements, where in case of *what* referents are undetermined. The difference between *what* and *any* is not important since they have the same semantic. The intention in using two different determiners is for efficiency reasons in implementing the evaluation of the formula against the database. Since our study does not involve the evaluation of the formula, only *the* and *what* are used in our notation to refer to definite and indefinite tuples.

5.3. Internal Query Generation

After having briefly introduced the syntax of D&Q and the changes we made to it, we discuss in the following section how different modifications that we introduced in the previous chapter can be represented in extended D&Q.

5.3.1. Basic Algorithm and Simple Sentences

Remember that a sentence is treated as a sequence of related words where words are classified in different categories. In interpreting the query, our language processing components categorize words in three different groups. The query in natural language is then viewed as a collection of constituents of the following three types in a syntactically correct order. The natural language query for an application will comprise the following :

- names that can map on entity/relationships or attribute names. **Entity, relationships and attribute names** are words that appear in the conceptual database schema and their synonyms.

- **content words**; these are words that mean something on their own. Possibly a proper name which is a value of some relation or one of its attributes e.g. *Ahmet, CMPE100*

- **function words**; these words have special meanings in the sentence. For example a word whose interpretation is a mathematical function such as *ortalama*

'average' *sayı* 'count'; question pronouns or question adjectives such as *kaç* 'how many', *hangi* 'which'.

In top-down parsing as described in the previous chapter, the major sentence S is extended by continually replacing the right-hand side by a possible left-hand side. Query generation is just the reverse process of parsing. In query generation, each noun phrase is interpreted as a reference to an entity, a description which involves the entity name as the predicate of its qualifier. Interactions between each noun phrase of the query is interpreted either by extending the current description or by combining it with an existing description. The algorithm is as follows:

Find the word (head noun),

If it is an entity name or relationship name

then create its description ,

collect all its relevant internal modifiers,

interpret these modifiers by extending the description

otherwise create the description of the entity or relationship that involves that word

into the memory

Interpret the interaction of the current description and the previous ones if there exist any

To exhibit how a natural language query can be represented in D&Q on the database described in Appendix C, consider the following simple query whose parse tree is given in Figure 5.1.

kimya derslerini göster

show the chemistry courses

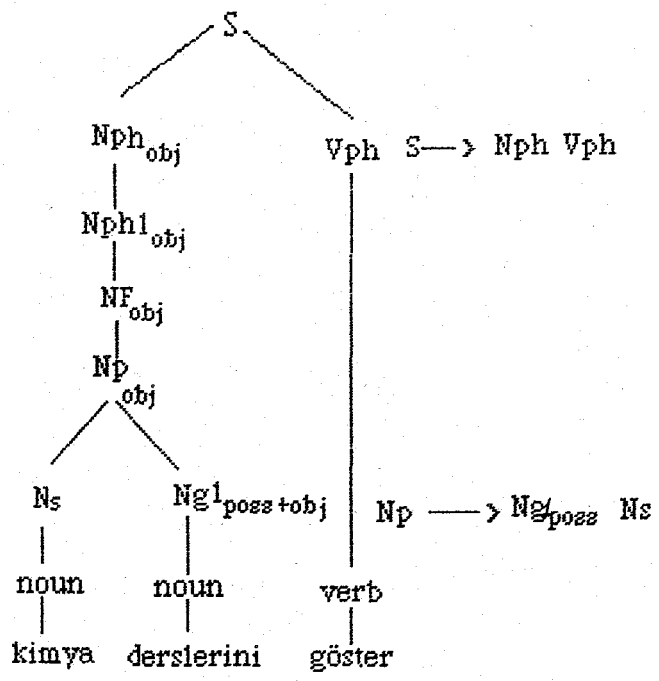


Figure 5.1. Parse tree for *kimya derslerini göster* 'show the chemistry courses'

After having parsed the word *ders* 'course', which is found to be a **entity name**, the query generator part of the language processor will create the following description whose qualifier is *ders*

the-N-qual (X,[], ders ()).

Next the parser proceeds to parse the preceding word *kimya*, 'chemistry', which is a noun modifier for the word *ders*, using

Np —> Ns Ng1 poss+obj

kimya can simply be mapped onto a particular attribute of the entity *ders* and the qualifier *ders* will be extended with the addition of the selection *ad = kimya* ' name = chemistry'

the -1-qual (X,[], ders (ad = kimya)).

Finally, the verb *göster* 'show' will update the description by changing the determiner *the* to *what*. In addition to that, the identifying attribute of the entity, key field, is bound to the description variable X which is in the display field. The final representation of our query will be expressed in D & Q as follows;

what-N-qual (X,[X], ders(kod=X,ad=kimya)).

While updating the qualifier *ders* it is necessary to select the relevant attribute of the file *ders*.To deal with the problem of attribute selection, each word in the lexicon has **class knowledge** associated with it. That is to say, our parser and query generator distinguish between words not only syntactically but **semantically** as well. We use four different classes in our example database; **person, item, number,** and **location**. For instance *hoca* 'instructor' and *öğrenci* 'student' belong to class **person**, whereas *ders* belongs to **item** and *not* 'grade' belongs to **number**. Associating a domain value to a question pronoun simplifies ambiguities as well. Consider the following two sentences,

kim kaç alıyor 'who is taking how many'

kim ne alıyor 'who is taking what'

It is possible to distinguish between the meanings of these sentences since *kaç* 'how many' and *ne* 'what' belong to classes **number** and **item** respectively. Similarly *kim* 'who' can be interpreted as *hangi öğrenci* 'which student' since öğrenci belongs to class **person** to which *kim* also belongs.

The conceptual database schema has a class a value associated with each attribute of a relation. For example the attributes of the entity *ders* 'course' and their corresponding class values are as follows;

kod	'code'	number
ad	'name'	item
bölüm	'department'	location
hocano	'instructor's id'	number

The interaction between relation name and noun modifiers is worked out on the basis of class knowledge. The class value of the word *kimya* 'chemistry' is **item**, and it can only match with the second attribute, hence *kimya* is taken as the value of attribute *ad* of the entity *ders*. When the class value matches more than one attribute class, the first attribute name is selected. The user may be asked to clarify the situation in order not to make a wrong decision.

All the internal modifiers extend the current description. The following paragraphs explain the interpretations of the three types of internal modifiers, i.e. adjective modifiers, question adjectives and comparative sentences.

Consider the following sentence which includes the adjective qualifier *iyi* 'good' and a question adjective *hangi* 'which'.

hangi iyi kimya dersleri var 'which good chemistry courses are there'

After having interpreted the noun modifier *kimya* the following description has been created;

the -1-qual (X,[], ders (ad = kimya)).

Remember that the meaning of *iyi ders* 'good course' is defined in the semantic definitions dictionary as a *ders* 'course' with *hocano = 1234* 'instructor's id = 1234', the parser will refer to the special dictionary to get the meaning of *iyi* 'good' as the modifier of *ders* and it will extend the description with includes the qualifier *ders* by inserting the selection *hocano = 1234* which then yields the following description,

the-1-qual (X, [] , ders (ad = kimya, hocano = 1234))

Function words are treated in the same way. Different parts of the description are modified depending on the semantic of the function word. To illustrate that, consider the same sentence and the following description of *ders*;

the-1-qual (X, [] , ders (ad = kimya, hocano = 1234))

The question adjective *hangi* 'which' modifying the word *ders* 'course' will replace the determiner *the* and the count *1* with *what* and *N* respectively and insert the key field variable X of *ders* into the display field to yield

what-N-qual (X, [X] , ders (kod=X, ad = kimya, hocano = 1234))

On the other hand, the interpretation of the question pronoun *kaç* ' how many' in the noun phrase *kaç ders* ' how many course(s)' will be to insert the "count" function **cnt** into the description to yield

what-N-qual (X, [**cnt(X)**] , ders (kod=X, ad = kimya, hocano = 1234))

Mathematical functions that can be used in the representation and their symbols are **cnt, tot, max, min, avg** for count, total, maximum, minimum and average respectively.

Comparative sentences comprise a word with comparative meaning and a noun phrase. Therefore, comparative sentences are treated as a pair consisting of a value and a comparison operator. The word at the end of the comparative sentence determines the

operator. As a simple example consider comparative sentence *5den çok* 'greater than 5' in the phrase *5den çok notlar* 'grades greater than 5'. Assume that *karne* 'stud_course', which includes *not* 'grade' as one its attributes, has the following description;

the-1-qual (X, [] , karne ())

Its interpretation is to add the selection *not* > *5* into the qualifier *karne* 'stud_course' which then yields

the-1-qual (X, [] , karne (not > 5))

The comparison operator requires identical elements on both sides. Notice that *not* 'grade' has a class value **number** which can easily be compared with the numeric value *5*. A tricky case may appear when the comparative sentence modifies a noun from a different class. In such cases, the count is taken into account simply by applying the mathematical function **cnt**.

5.3.2. Relations

To handle relations between two noun phrases, it is necessary to combine descriptions. Fortunately, descriptions can be combined in D&Q with **is funct** or **is qual**.

If there are two noun phrases with interpretation Desc1 and Desc2 having the formats, the-N2-qual(X1,Ds1,Qual1) and the-N2-qual(X2,Ds2,Qual2) respectively, then the overall interpretation will be

the-N2-qual(X1,Ds1,Qual1) **is qual**(**Y**,the-N2-qual(X2,Ds2,**Qual2**))

One of the selections of the qualifier **Qual2** contains the variable **Y** assigned to the field with which the entities associated with **Qual1** and **Qual2** can be joined.

It is quite possible to have the relationship file to serve in a sense to connect the other two entity files together. Assuming that Desc1 and Desc2 are the two basic descriptions, and Desc3 is the *relationships* file, the general interpretation will be in the form

Desc1 **is qual** (**Y**, Desc3 **is qual** (**Z**, Desc2))

where interaction between two descriptions is provided by two variables **X** and **Z**.

There are two possible relations that may occur between two noun phrases. Two noun phrases are either related over the verb or they are in a possessive relation.

For the case of **relation over the verb** two noun phrases may modify each other over the verb. The type of the modification is distinguished from the verb. The relation that we express in D&Q is ***subject-object*** relation. It is expressed as the **join** of two entities.

Consider the natural language query

hangi hoca CMPE100 dersini veriyor

'which instructor is giving the course CMPE100'.

As diagrammed in Figure 3.2. *ders* 'course' with the case marker *objective* is the ***object*** and *hoca* 'instructor' with case marker Ø is the ***subject*** of the sentence. The verb of the phrase *ver* 'give' can take *ders* and *hoca* as its *object* and *subject*, respectitively. We can express this relation by joining the entity files *hoca* and *ders*. *Hoca* becomes the description of *ders* which can be interpreted with **is qual** qualifiers.

what-1-qual(Y,[Y],hoca(hocano=Y)) **is qual(A**,

the-1-qual(X,[],ders(ad=CMPE100,hocano=**A**)))

For the case of **possessive relation**, consider the entities *ders* 'course' and *hoca* instructor' and the possessive relation

hocanın dersi 'course of the instructor'.

This relation will be expressed as

the-1-qual(Y,[],hoca (hocano=Y))) **is qual(A**,

the-1-qual(X,[],ders(kod=X,hocano=**A**)))

The order of the relation is important in expressing possessive relations, because the noun on the right is qualified by its predecessor. When the relation, *dersin hocası* 'the instructor of the course' is to be formalized, the expression would look like

the-1-qual(X,[],ders(hocano=X)) **is qual (A,**

the-1-qual(**A**,[],ders(hocano=**A**)))

In our example database *ders* 'student' and *öğrenci* 'student' are two entity files that may be related over the relationship file *karne* 'stud_course'. The D&Q interpretation of the possessive relation

dersin öğrencisi 'the student of the course'

is thus

the-1-qual(Y,[], ders(kod= Y)) **is qual(C,**

the-1-qual(Z,[], karne(num=Z,kod=**C**)) **is qual(B,**

the-1-qual(**B**,[],öğrenci(num=**B**))))

For example

öğrencinin dersinin hocası 'the instructor of the course of the student'

the nested possessive relation can be expressed as follows keeping possessives in correct order,

the-1-qual(X,[],öğrenci(num=X)) **is qual(A,**

the-1-qual(Y,[], karne(num=**A**,kod=Y)) **is qual(B,**

the-1-qual(Z,[], ders(kod= **B**,hocano=Z)) **is qual(C,**

the-1-qual(C,[],hoca (hocano=**C**)))))

5.3.3. Attribute Names of Entities

In the previous examples, all constituents were entity names. They can easily be combined without any complication. However, when an attribute is encountered it can be represented in the formal description of the entity if the attribute name is unique in the database. Our generator keeps the attribute name in the memory until a possessive

relation is encountered with the attribute name appearing as one of the constituents. Consider the following sentence

derslerin adlarını göster 'show the names of the courses'

After having parsed the word *ders* 'course', the possessive relation between *ders* and *ad* 'name' can be interpreted. The query generator defines the description of *ders* as follows,

the-1-qual(X,[],ders(ad=X))

However, in interpreting the query

derslerin notlarını göster 'show the grades of the courses'

not 'grade' is not an attribute of the relation of *karne* 'stud_course', and it is unique.

The query generator defines the description of *karne* as

the-1-qual(X,[],karne (not=X))

To interpret the relation *karne* may be joined with *ders*. Having the *karne* description with *not* we can now express the relation between *karne* and *ders* as the following

the-N-qual(Z,[],ders(kod=Z) is qual (Y,

what-N-qual(Y,[X],karne (kod=Y,not=X))

5.3.4. Content Words and Question Pronouns

Remember that a content word is a proper name which is a value of some relation or one of its attributes e.g. *Ahmet, CMPE100*. The way of handling content words which function as noun modifiers has been explained before. More sophisticated cases may occur, for instance, a content word may be the first constituent of the possessive relation. An example of this is

CMPE100un hocası 'the instructor of CMPE100'

The description of *hoca* 'instructor' is already in the database. When the parser needs to relate the word *CMPE100* having the possessive property with the word *hoca*, it should determine the appropriate relation name for the latter. The dictionary is looked up for this. The dictionary contains the following information about the functional relationships between possible objects of the verb *ver* 'give a course',

(ver ders object)
(ver not object)
(ver öğrenci dative)
(ver hoca subject)

where the second constituent appears as the subject with case marker Ø. *CMPE100* belongs to the same domain as *ders* 'course' which is in *subject-object* relationship with *hoca*. Thus, *hoca* can be interpreted as the subject of the action *ver* whose object is *CMPE100*. After creating the following description for *ders*

the-1-qual(X,[], ders())

the resolution of the possessive relation is reduced to a simple combination of two relations *hoca* and *ders*.

Question pronouns are handled exactly in the same way. Consider the following sentence with the question pronoun *kim* 'who',

kim CMPE100 alıyor 'who is taking CMPE100'

The pronoun *kim* will be understood as the subject of the verb *al* 'take' after refering to our semantic relationships dictionary. *Öğrenci* 'student' which is the subject of the verb, will replace *kim*. Hence the above query will be reduced to the simple relation of *öğrenci ders*. The only difference is that *öğrenci* has to be displayed.

Similarly the following query

kim ne alıyor 'who is taking what'

will be treated as *öğrenci ders* relation where both constituents should be displayed.

The other two constructs that may be interpreted by our formal query generator are two forms of participles. They are treated as possessive relations. In fact

hocanın verdiği ders 'the course given by the instructor', and

ders veren hoca 'the instructor who gives a course'

have the same semantics as

hocanın dersi 'the course of the instructor', and

dersin hocası 'instructor of the course'

The order of the subject-object relation, which is the most important semantic concept, is also preserved in possessive relations.

In handling noun relative clauses, the referent of the participle is found in semantic relationship dictionary. For instance

hocanın verdiğini '(that is) given by the instructor'

is converted to

hocanın verdiği ders 'the course given by the instructor'

since *verdiğini* has an case marker *objective* and the first item with *object* function found in semantic relationships dictionary is *ders* 'course' for the verb *ver* 'give'. The interpretation of the latter is already discussed.

5.3.5. More Specific Cases

Consider the following sentence

kimya ve fizik derslerini göster 'show the chemistry and physics courses'.

Although *kimya* ve *fizik* 'chemistry and physics' modify the head noun *ders* 'course' with the conjunctive *ve* between them, no reference to *ders* can be found satisfying *kimya* and *fizik* at the same time. We can express this in terms of two different modifiers connected with &, where the symbol & denotes the connective "and" to yield

what-N-qual(X,[X],ders (kod=X, ad=kimya)) & what-N-qual(Y,[Y],ders(kod=Y, ad=fizik))

Alternatively, kimya and fizik can be considered as alternative values of the function which contains *ad* 'name' as its constant parameter. In this case, the two descriptions *kimya and fizik* will be combined as kimya & fizik which in turn is combined with *ders* description via **is funct** to give

kimya & fizik is funct(Y,what-N-qual(X,ders (kod=X,ad=Y)))

Sentences may contain noun phrases with case value dative (-e hali) or case value ablative (-den hali). They have to be converted into a canonical form which contains the noun phrase as its object or subject. This conversion is accomplished by consulting the semantic relationships dictionary.

In the sentence,

Hocadan ders alan öğrenciler 'students who take courses *from the instructor*'

the word *hoca* 'instructor' with the case marker ablative modifies the verb *al* 'take' whose object is *ders* 'course'. Subject-object relationship between *hoca* and *ders* is defined in the dictionary under the verb *ver* 'give'. Therefore *hocadan* is transformed to *hocanın verdiği*. The new form of the above sentence is

hocanın verdiği ders*i* alan öğrenciler 'students who take *the course given by the instructor*'

and it is reduced to two simple relations, and which can be represented as explained previously.

5.4. A Full Example of Representation Process

Let us illustrate query understanding with the example whose parse tree is shown in Figure 5.2. Numbers show the sequence in which descriptions are created or modified. After having parsed the noun ders 'course', the description of *ders* **(1)**

the-1-qual (X,[], ders ())

is created. The noun qualifier *CMPE100* will be understood as the value of the attribute *ad* 'name' for *ders* and the description will be updated **(2)** as

the-1-qual (X,[], ders (ad=CMPE100))

Parsing *hoca* 'instructor' will produce its description **(3)** as

the-1-qual (Y,[], hoca ())

and the question adjective *hangi* 'which' will modify *hoca* by changing the determiner *the* to *what* and it will put the *key field of hoca*, *hocano* 'instructor's id', into the display field**(4)**. Thus the description will be

what-N-qual (Y,[Y], hoca (hocano=Y))

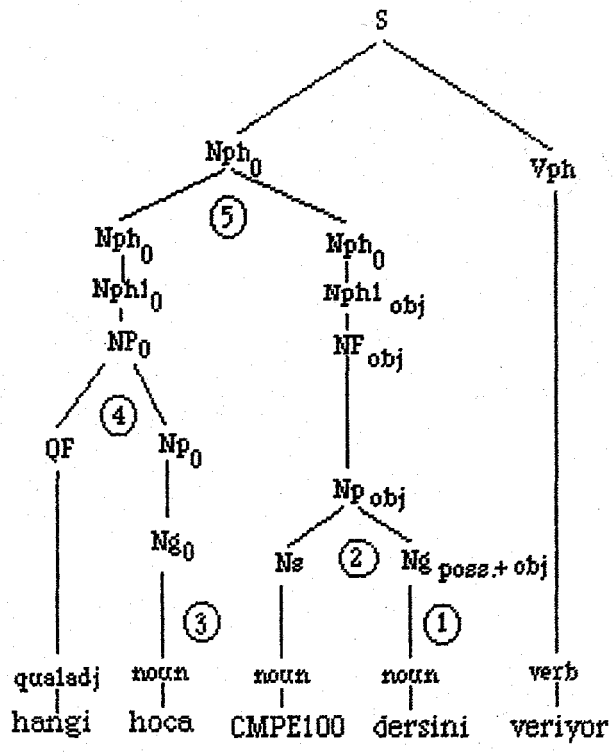


Figure 5.2. The parse tree for *hangi hoca CMPE100 dersini veriyor* 'which instructor is giving the course CMPE100'

The last step in translation **(5)** is to set the relation between *hoca* and *ders*. They will be combined with **is qual** to yield,

what-1-qual(Y,[Y],hoca(hocano=Y)) **is qual(A**,

the-1-qual(X,[],ders(ad=CMPE100,hocano=**A**)))

5.5. Conversion to a Declarative Language

Using the previously described interpretations, the analysis of the Turkish query

hangi hocalar CMPE100 dersini veriyor

would give the following D&Q interpretation

what-1-qual(Y,[Y],hoca(hocano=Y)) **is qual(A,**

the-1-qual(X,[],ders(ad=CMPE100,hocano=**A**)))

The next step is to convert this representation to an expression in a declarative query language. The language used is SQL. Its structure consists of a set of display values and file names and a list of logical expressions connected with reserved words SELECT, FROM and WHERE. Display values are in the form of

filename . fieldname

and logical expressions are

file name.fieldname comparison-operator file name.fieldname / constant

The converter relates each D&Q formula to a set of display values and a list of logical expressions. D&Q expressions are transformed into the declarative language by applying a set of rules. There are mainly two different groups of rules, each converting a different part of the whole expression; one for qualifiers and one for descriptions. The former converts selections in qualifiers into a list of logical expressions where each expression is of the form given above. For instance

... ders(.., ad=CMPE100)..

is converted to ders.ad = CMPE100

For selections containing a variable the value of the variable is substituted if it is bound, otherwise it is bound to this value. In converting

hoca (hocano = Y)

Y will get its value as *hoca.hocano* to be used for later references.

Description converter deals with display values. After having all the selections in the qualifier converted, values of variables in the display field are added into the set of display values for description whose determiner is *what*, i.e. *hoca.hocano* in our example.

Combined descriptions are converted by adding a join expression into the list of logical expressions. For instance **is qual** is converted by adding X=Y, more specifically

ders.hocano = hoca.hocano

Our example query will then look like

SELECT hocano
FROM hoca
WHERE
ders.ad= CMPE100
hoca.hocano = ders.hocano

The declarative query obtained after the first step is still in user's view. It will then be converted to actual fields specified in the domain database mapping table. This transformation is a simple mapping. Using the entries given below,

(hoca , hocano, instructor, instr_no)	hoca . hocano --> instructor . instr_no
(ders, ad, course, c_name)	ders. ad --> course . c_name
(ders, hocano, course, instr_no)	ders. hocano --> course . instr_no

Database dependent representation of our example query is then,

SELECT instr_no
FROM instructor
WHERE
course.c_name = CMPE100
instructor.instr_no = course.instr_no

VI. IMPLEMENTATION

Our model has been implemented on a IBM compatible PC AT with 640 KB memory. Turbo Prolog 2.0 [20,22] is used as the implementation language. The system runs as five different modules, the morphological parser, the syntactic parser , the query generator, the spelling corrector and the translator.

Modules are combined into a single, stand-alone project MYTEZ using the modular programming feature of Turbo Prolog. Modules communicate with each other using the predicates in global predicates and global database sections. All of the five modules above have been implemented.

Below we give a brief overview of the clauses used in implementing five modules, and discuss the main issues in design together with the knowledge source storage.

6.1. Morphological Parser

The morphological parser contains one rule for each node of the decision tree given in Figure 4.2. The module is called with the clause **C_W** which gets the noun to be parsed and returns its properties. There can be four properties associated with a noun. We use a parameter for each type of property rather than a single parameter with different values for each possible combination of the properties. These properties are its type, its domain, its genitive property and its case value, represented as **T, D, Fp**, and **Fh**, respectively. Values for these parameters are strings consisting of one or two characters. For instance **T** can be bound to "n" for a common noun, to "p" for a pronoun, to "qp" for a question pronoun, or **Fh** can have the values "ø","o","t","l","f","t1" denoting different case properties such as subject, object, dative, locative, ablative and genitive, respectively

The structure of the predicate **C_W** is

C_W(W, RW, D, T, Fp, Fh)

The input parameter **W** supplies the noun to be parsed to **C_W**. It calls the root node clause **c_n** to check if the ending of the noun in question corresponds to one of the possible endings of suffixes.

Each level of the tree is represented by a group of more than one clause with the same name, differentiated with their first constant parameters. Parameters correspond to the possible end letters, e.g. **i,u,n,r,a,** or **e** for the first level. For instance, the clause for the first level is **c_nf** and it has the following structure:

```
c_nf("i", To, L, RW, D, Typ, Fp, "o"):-
        ch_rest("y", To, L, RW, D, Typ, Fp).
```

This clause of the **c_nf** is satisfied when the last letter of the noun is "**i**" , and it will bound the last parameter **Fh** to "**o**", the objective case. Clauses corresponding to other levels are **ch_rest, ch_rest1, ch_rest2,** and **ch_rest3**. The predicate, **ch_rest2** bounds **Fp** to "**t2**" if the noun has the possessive property. The first parameter of the last clause is unbound, and it does dictionary look up at that level when no suffix elimination is no longer possible .

Dictionary look up for a noun is done with predicate **ch_word**. It checks for the plurality , and calls the predicate **find_noun** which returns its type **Typ**, its domain **D**, and the canonical form **RW** of the word for synonyms.

6.2. Syntactic Parser

Our syntactic parser is written as a set of production rules. Each rule accepts a list of words, parses head words in the list, and returns the remaining part of the list together with the information gained at this part of the parsing process.

The verb phrase in our major rule, for instance, is written in Prolog as

```
Vph([V|L] ,[V] , L) :- verb(V), asserta(v( 0,[] ,V, [] )).
```

The value of **V** is returned together with the remainder of the sentence **L** provided that it is a verb. Information is propagated in the parsing process. Either they are passed as values for the parameters or they are kept in the memory. In order to handle the relation over the verb, a database fact which has the form,

v(type, object , verb, subject)

is inserted for each verb parsed in the sentence. This is done by the clause **Vph**.

Each rule contains parameters to hold the necessary information at that level as well. The predicates which parse a noun group must return the information about the noun group together with its functions. Consider the predicate **Ng**, for instance. It has the form

Ng(L, NDL, Conj, T, Fp, Fh, RNgL)

where **L** is the only input parameter containing the current input string, and **NDL** is a list which returns a group of one or more nouns parsed in that clause. Each noun is kept in the list as a noun-domain pair in the form **e(noun,domain)**. The word group has its functions; its type **T**, its possessive property **Fp**, and its case value **Fh**. **Conj** is bound to the conjunctive if the group consists of more than one noun. In addition to these, the last output parameter **RNgL** returns the remaining part of the input string. Assuming that the clause **Ng** is called with **L** bound to *Ahmetin dersini ve hocasını* 'the course and the instructor of Ahmet' output parameters will get the following bindings :

NDL ≡ [e(ders,"i"),e(hoca,"p")]
Conj ≡ "ve"
T ≡ "n" (common noun)
Fp ≡ "t2" (existence of possessive property bounds Fp to t2)
Fh ≡ "" (no genitive property or case marker)

There exists a separate clause to parse the last two levels of modifiers, namely premodifiers and noun modifiers. Similar to the clauses that parse nouns, they have an input list **L** and an output list **RL** containing the sentence being parsed, two parameters **QL** to return the modifiers parsed, and **Conj** a possible conjunctive between them. The structures of **AF** and **QF** are

AF(I, T, L, QL, Conj, Fq, RL) **QF(_, L, QL, Fq, RL)**

In addition to these parameters, some clauses include input parameters used to distinguish between different types. For instance, the parameter **T** in **AF** is bound to 0 or to 1 depending on whether the adjective expected is a simple one or it is in verb form.

respectively. Since different subcategories of modifiers are parsed using the same clause there also exists an output parameter **Fq** in which the type of the parsed modifier is returned.

Reminding that our parser is implemented as a transition network parser, let us illustrate our basic approach in handling rules on the following two rules

A $\longrightarrow$ **B C**

A $\longrightarrow$ **B D**

B can be implemented as

```
B(input,information-gained,remainder):-
        parsing_B(input,rest),
        before_b(rest,remainder).
```

with two **before_b** clauses,

```
before_b(input,remainder):-
        c(input,remainder).
before_b(input,remainder):-
        d(input,remainder).
```

In this approach, we do not throw out the information gained in parsing **B** in case **C** fails, but use it to handle rules where there are many candidate right-hand sides having common parts for the same left-hand side. It is used for grouping successive words and for parsing premodifiers. As an example of the implementation of the syntactic parser as a transition network consider the main rule **NF** which is in the form,

$$\mathbf{NF}_{(caseM \mid gen)+(poss)} \longrightarrow$$

$$\mathbf{pronoun}_{(caseM \mid gen)+(poss)} \mid \mathbf{qpronoun}_{(caseM \mid gen)+(poss)} \mid$$

$$\mathbf{Ng2}_{(caseM \mid gen)} \mid$$

$$(\mathbf{ASF} \mid \mathbf{AFS} \mid (\mathbf{CS})\ (\mathbf{QF})\ (\mathbf{AF}))\ \mathbf{Np}_{(caseM \mid gen)+(poss)}$$

Remember that the rule **NF** parses four different types of nounlike words, namely pronouns, question pronouns, proper nouns, and nouns. Note that a sequence of proper nouns and nouns are allowed and a sequence of premodifiers may only precede

nouns. Thus, four options exist for the rule **NF**, instead of using four rules for each type of nounlike words we implement **NF** as a single rule.

```
NF(L,VL,DL,Conj,T,Fp,Fh,RL):-
        Ng(L,NDL,Conj,T,Fp,Fh,RNgL),
        :
        b_Np(RNgL,DL,T,Fp,RNpL),
        b_NfNp(RNpL,DL,T,Fp,Fh,RL).
```

It consists of the predicate **Ng** to parse a nounlike word or a sequence of nounlike words and two more clauses **b_Np** and **b_NfNp** which are satisfied according to the type of nounlike word parsed in the predicate **Ng**.

```
Ng([W| L],DL,Conj,T,Fp,Fh,RL):-
        C_W(W,RW,D,T,Fp,Fh),
        b_Ng(T,L,NDL,Conj,Fp,Fh,NRL),
        apwad(e(RW,D),NDL,DL).
```

The first clause **C_W** is the clause in the morphological parser which returns all functional and domain information about the input noun in **W**. After having parsed the nounlike word there may be different possibilities depending on the value returned in **T**. Our program includes different **b_Ng** predicates for each possibility differentiated by the value of **T**. For instance, a noun can be followed by a conjunctive and by another noun group having similar functional properties. The clause that satisfy this is the following,

```
b_Ng(T, [Wc,W|L], DL, Conj, Fp, Fh, NRL):-
        conj(Wc),
        C_W(W, RW, D, T, Fp, Fh),
        Ngn(T, L, NDL, Fp, Fh, RL),
        apwad(e(RW,D), NDL, DL).
```

Notice that, this time, **C_W** is called with determined **T**, **Fp** and **Fh** values. In case the clause **conj(Wc)** is satisfied, **C_W** must also be satisfied so that two successive nouns of the same type are separated with a conjunctive. Theoretically, an infinite number of noun qualifiers can be parsed by recursively calling the clause **Ngn**. The final noun list is obtained by appending the currently parsed noun to the list of nouns returned by

Ngn. However, the existence of a noun is not necessary in order to satisfy the clause. It returns an empty list and bounds **RL** to the input list when no noun is found. The following two clauses handle this situation

```
Ngn(T, [W|L], DL, Fp, Fh , RL):-
        C_W(W, RW, D, T, Fp, Fh),
        Ngn(T, L, NDL, Fp, Fh, RL),
        apwad(e(RW,D), NDL, DL).
Ngn(_, L, [], _, _, L).
```

We use the same principle for the grouping of other words, e.g. adjectives and comparatives. The list of clauses which handles a sequence of adjectives is

```
AF(_,0,[W|L],QL,Conj,"a",RL):-
        find_adj(W,RW),
        b_AF("s","",L,AQL,Conj,RL),
        append(RW,AQL,QL).
        b_AF("s",F,[Wc,W|L],QL,Wc,RL):-
         conj(Wc),
         find_adj(W,RW),
         ADF(1,"s",F,L,AQL,RL),
         append(RW,AQL,QL).
ADF(0,"s",F,[W|L],QL,RL):-
         find_adj(W,RW),
         ADF(0,"s",F,L,AQL,RL),
         append(RW,AQL,QL).
```

The clause **AF** parses the first adjective and calls the clause **b_AF** parses the conjunctive and the second adjective whose existence is mandatory. The sequence of indefinite number of adjectives are parsed in the clause **ADF** which recursively calls itself.

The clause **Ng** which parses a single noun or a group of nouns corresponds to **Ng1** and **Ng2** depending on the value of T, and **Ngn** corresponds to **Ngn1** and **Ngn2**.

Referring back to the clause **Ng**, a pronoun or a question pronoun, denoted by "**p**" and "**qp**" respectively, might be parsed as well in **Ng**. Pronouns are not allowed to be connected with a conjunction at this level, and therefore **b_Ng** clauses corresponding to these cases are

b_Ng("p",L,[],"",_,_,L).

b_Ng("qp",L,[],"",_,_,L).

After having found the noun group by using the rule **Ng**, we have to check the rest of the sentence for noun modifiers and premodifiers. Parsing of possible noun modifiers and premodifiers are implemented in two predicates, namely **b_Np** and **b_NfNp**, respectively. Each call includes the list of head nouns in **DL**. This is because when a modifier is found. It will modify the head nouns in the same clause.

The clause **b_Np** is called to parse a group of noun modifiers when the input parsed in the clause **Ng** is a noun, it has the form

b_Np(L, DL, "n", "t2", RL):-

Ns("n", L, QNL, Conj, RL),

nqquantify(DL, QNL, RL).

The clause **Ns** has a structure similar to **AF**.

The clause **b_NfNp** which parses the premodifiers of a noun, it corresponds to the following option in the rule **NF**,

$$\mathbf{NF}_{\mathbf{(caseM/gen)+(poss)}} \longrightarrow \mathbf{\{ASF \mid AFS \mid (CS)\ \ (QF)\ \ (AF)\}\ Np}_{\mathbf{\{caseM \mid gen)+(poss)}}.$$

The structure of the rule **b_NfNp** is as follows

```
b_NfNp(L,DL,_,_,_,RL):-
        AF(0,_,L,AQL,ConjA,Fp,RAFL),
        quantify(Fp1,DL,AQL,ConjA),
        QF(0,RAFL,QQL,Fq1,QRL)
        quantify(Fq1,DL,QQL,""),
        CS(0,QRL,DL,RL).
```

The **AF** is written as

```
AF(_,0,[W|L],QL,Conj,"a",RL):-
        find_adj(W,RW),
        b_AF("s","",L,AQL,Conj,RL),
        append(RW,AQL,QL).
```

The list of simple adjectives can be collected in the predicate **b_AF**. Notice that **AF** is optional in the rule and optionality for the rules is provided with an extra parameter, which is the the first parameter of each clause. Addition of two more clauses to the end of **AF** predicates list does this. They are

```
AF(0,0,L,[],"","",L).
AF(1,0,L,[],"","",L):-fail.
```

AF is satisfied when the first parameter is 0, that is , it is skipped by returning the whole input list. It **fails** when the first parameter is 1.

Adjective clauses **AFS** are implemented by adding two more clauses for **AF**. When an adjective in verb form is parsed with the clause **c_adjv**, the verb is considered as the main verb of the coming noun phrase and it is parsed with **Nph**. The structure of **AFS** which is used to parse participle constructed with the suffix *-en*, is,

```
AF(_,1,[W|L],QL,Conj,"1",RL):-
        c_adjv(W,RW,"1"),
        Nph(RAL,[RW],_,RL).
```

Actually **NF** is called from **Nph1** to parse the noun group and its modifiers. The clause **Nph1** is defined as follows

```
Nph1("v",L,VL,DL,Conj,OldDL,Fp,Fh,RL):-
        NF(L,VL,DL,Conj,Fp,Fh,IRL),
        b_Nph1("v",IRL,VL,DL,Conj,Fp,Fh,RL).
```

Noun groups parsed at lower levels modify either another head noun or the verb at the **Nph1** level depending on the information carried in **Fp** and **Fh** parameters.

The predicate **b_Nph1** is written for the case where the currently parsed noun phrase has the possessive property "t2" on it. A noun phrase with the genitive property is expected. The clause **b_Nph1** is defined as

```
b_Nph1("v",L,VL,DL,Conj,"t2",Fh,RL):-
        verbquantify(Fh,VL,DL,Conj),
        Nph1("v",L,VL,DL,NConj,DL,Fp,"t1",RL),
        genquantify(DL,NDL,Nconj).
```

The first predicate **verbquantify** qualifies the verb in **VL** according to the function in **Fh**. Next, it attempts to parse a noun with the genitive property. In case of success, possessive relation between two nouns is represented by combining their descriptions in **genquantify**.

Finally two noun phrases with a conjunctive in between is parsed with the rule

```
b_Nph1("v",[Wc|L],VL,DL,Conj,Fp,Fh,RL):-
        verbquantify(Fh,VL,DL,Conj),
        conj(Wc),
        Nph1("v",L,VL,DL,NConj,DL,Fp,Fh,RL).
```

The clause **verbquantify** modifies the verb by adding the modifier into the database predicate **v(0,[] ,V, [])**, if the noun has case marker *objective* or Ø. The modification is implemented in two steps. The structure of the clause **verbquantify** is

```
verbquantify(Fh,[V|L],DL,Conj):-
        add_case_verb(N,Fh,V,DL).
```

The clause **add_case_verb(N,Fh,V,DL)** retracts the existing database predicate for the verb **V**, and calls the clause **append_case** to insert the new form of the database predicate. There are separate clauses for different possibilities rising from the type of the verb, its voice and the function of the modifier. The structure of **add_case_verb** is

```
add_case_verb(N,Fh,V,DL):-
        retract(v(N,Co,V,Sub)),
        append_case(N,Fh,DL,V,Co,Sub).
```

The clause **append_case** which handles the noun with case marker Ø on it (denoted with "") modifying the main verb in passive voice is

```
append_case(0,'',DL,V,Co,Sub):-
        passive(V),
        assert(v(N,DL,V,Sub)).
```

6.3. Meaning Representation and Internal Query Generator

Before we discuss clauses used in meaning representation, let us see how descriptions are stored in the memory. Descriptions and qualifiers are kept as database facts. As the implementation is in Turbo Prolog, the form of all database facts must be predefined. Hence we use two domain definitions for descriptions and qualifiers, namely ***desc*** and ***qual***. There are separate domain functors for each type of descriptions and qualifiers. The domain definition for the qualifier type

relation name(list of selections)

is **q2(Rname,attrL).**

The **attrL** parameter is a list consisting of elements defined as

at(termtype symbol termtype)

where a **termtype** can be one of the following

t1(string) , t2(string,string) , var(string)

since a value or a relation name can be used. For instance the qualifier ders(kod = 1234) is implemented in Turbo Prolog as

q2(ders, [at(t1(kod),"=",t1(1234))])

The description **determiner-count-qual(variable, display list, qualifier))** has the general structure as

d2(determiner,count,variable,display list, *qual*)

The description for *ders* with the general syntax

what-1-qual (X,[X], ders(kod =X))

can be written in Prolog as

d2(what,1,var("X"),[var("X")],q2(ders,[at(t1(kod),"=",var("X")])).

The major drawback of Turbo Prolog is that free variables can not be inserted into memory. To tackle this problem, we use an extra domain functor **var** containing the variable name. We introduce two different functors as **d3** and **q4** to handle a combination of descriptions and qualifiers using **is funct** and **is qual** respectively. For example, the description **is funct (variable, description)** is implemented as

d3(*desc*, var("X"), *desc*)

and description **is qual(variable, qualifier)** is implemented as

q4(*desc*, var("X"), *qual*)

A meaning representation module consists of a clause for each main task. These are the tasks of asserting the head noun, converting content words, question pronouns to canonical forms, handling noun and adjective qualification, and creating relations.

The predicate **asq** asserts the description of the head noun into the memory. It checks if the head noun is a relation name using the user view of the conceptual data schema, and it asserts a description for each relation name. The assertion has the form

d2(the,1,var("X"),[],q2(ders,[t1(kod),"=",var("X")])

for *ders*, for example.

Conversion of content words is accomplished by the predicate **convert**. It gets as input the content word together with its domain, its case marker, its type, and the verb of the sentence and returns the relation name. The description of the relation is either modified or inserted into the memory. The program contains separate clauses for the conversion of content words, question pronouns, and adjective or nouns in verb form. They use the semantic relationships information about the verb and its possible objects to get the relation name.The clause used to convert *kimya* 'chemistry' in *kim kimya alıyor*. ' who is taking chemistry' is

```
convert("n","o",[al],[e(kimya,"i")],DL):-
        con_n_obj("o",[al],e(kimya,"i"),DL).
```

where **con_n_obj** bounds **DL** to **e(ders,"i")** due to the following fact,

ss(al,ders, "i","o")

Referent of noun clauses are determined in the clause **con_verb_obj**. It has the form

con_verb_obj("nf1", "o", e(al,"i"), N):-

find_v(1,"o", al, N).

The clause **find_v** which uses the semantic relationships dictionary. This is defined as

find_v(1,_,To,e(N,Dn)):-

ss(To,N,Dn,"a")

:

In the above example, the referent of the noun phrase *alanları* is determined and **N** is bound to *öğrenci* due to the following fact

ss(al,öğrenci, "p","a")

The three predicates used in relation handling are **f_r**, **dofr**, and **connect**. Using these three clauses, we can create all possible combination of descriptions. **f_r** determines the common fields of the two fields. The decision is based on the conceptual schema of the database. **do_fr** fetches the named descriptions and calls the clause **connect** to form a combined description.

To handle the relation over the verb when both object and subject arguments are filled in, **f_r** is called to relate these arguments. The type parameter is used to determine the order of the relation. For instance, *ders veren hoca* 'the instructor who gives a course' and *hocanın verdiği ders* 'the course taught by the instructor' have the same object and subject values, but the order of combination is different. Here are the two database facts for these noun phrases respectively,

v(1,ders,ver,hoca)

v(2,ders,ver,hoca)

The entry for *hangi hoca ders veriyor* 'which instructor is giving a course' will have the type value Ø.

The possessive relation is handled by the predicate **genquantify** which simply calls **f_r** after having converted the content words appearing in the possessive relation.

Function propagation is carried out using a different parameter in every relevant clause; but descriptions are kept in the memory rather than using a list that contains current descriptions. Although memory access takes time, it is assumed that implementation as a list will be slower since the required description may be anywhere in the list. In addition, the list implementation will make the program code larger.

6.4. Spelling Corrector

The spelling corrector mainly consists of two sets of rules, one set of rules for asserting a lexical item into the dictionary, and one for comparing the unrecognized input with the entries in the dictionary. The clause **corr** controls the spelling correction. It consists of two rules in the following form

```
corr(_,Str,Clist,W2,D):-
        str_length(Str,Length),
        L1=Length/2,
        frontstr(L1,Str,Bstr,Estr),
        ch_cr(Bstr,Estr,L1,W2,D,Clist).
corr(1,Str,_,Str,D):-
        write(" Görevini giriniz > "),
        readln(Func),nl,
        write(" Eşanlamlısı var mı ? (e/h) >"),
        readln(Answer),
        adding(Str,Func,D,Answer).
```

The parameter **Str** keeps the misspelled word. Its correct form is returned in **W2**. The parameter **Clist** contains the list of possible categories that the word in **Str** may belong to .

Assertion is handled in the clause **adding**. Separate clauses exist for different functions of the word. The structure of the clause **adding** used to add a new adjective is

```
adding(Str,"a","","h"):- assert(adj(Str)).
```

To add a noun, its class value is also required from the user. For synonyms, the noun for which the unrecognized input is the synonym asked from the user, and synonyms are added into the synonym dictionary.

Before adding a word into the dictionary, the spelling corrector tries to find a word close to the misspelled one in the dictionary. The spelling corrector is called with a list of possible categories that the word may belong to. Different clauses are called from **ch_cr** depending on the entries in the list **Clist**. There are four different clauses namely, **ch_n, ch_s, ch_p, ch_v** to correct the spelling of a noun, adjective, pronoun and verb respectively. They are similar in structure and they use the clause **comper** in which the comparison is handled. For example, the clause for correcting the misspelled adjectives, **ch_s** is defined as

```
ch_s(Bstr,Estr,L1,""):-
        adj(W1),
        frontsr(L1,W1,Bw,Ew).
        comper(L1,Bw,Bstr,Ew,Estr,W2).
```

What this clause does is simply to get an adjective from the dictionary, to split it into two parts and to compare the first and the second parts of both words. The clause **comper** handles the comparison and returns the correct form after the word found is confirmed by the user. Its structure is

```
comper(_,Bw,Bstr,Ew,Estr,W1):-
        Bw=Bstr,
        concat(Bw,Ew,W1),
        concat(Bstr,Estr,W2),
        write(W2," yerine ",W1," kullanılabilir mi ? > "),
        readln(Answer),
        Answer="e".
```

6.5. Translator

Conversion to a declarative language query is implemented using an algorithm similar to the one given in Wallace's book. It is written in Turbo Prolog , and it works for the extended D&Q syntax. The output of the query generator is used as the input for the converter. The input of the converter is a single description consisting of one or more combined descriptions related to each other through variables. Starting from the inner qualifier all the combined descriptions and qualifiers are converted to simple declarative database language queries. As Turbo Prolog does not allow free variables in the memory, the input description does not contain any free variable. In place of using a free variable X, we use a bound term **var** that contains X as its argument. Unification is handled explicitly by keeping a table for variables. Every entry in the table consists of the variable name and its value. Every time we have to unify two variables we copy the value of the "bound" variable into the value field of the "free" variable. Unification fails if either of both variables contain different values or they do not have any values.

6.6. Knowledge Source

Dictionaries and tables are held as database facts. There are different predicates for each category of words. For instance facts containing noun-like words have two arguments, the word and its type, each of which is stored as a single character string. The basic lexicon has the predicates noun,pronoun, adj, verb,conj,quanadj, e.g.

```
noun("kimya","i")
pronoun("kim","p")
adj("iyi")
verb("ver")
```

Except for qualadj's, where a third object is used to distinguish between them, it allows to use the same code for four different types of qual's.

```
qual("kaç","q"),
qual("bu","d")
```

This approach facilitates the database search tremendously by providing a self contained index at the level of functionality.

For synonym dictionary ,the two synonym words and their function are kept in the same database "fact". The first word is the canonical one which will be returned when the second is referred to, e.g.

```
syn("öğrenci","talebe","n")
```

The conceptual data schema is stored as entity and field pairs including a "marking" object to denote the key fields of each relation:

```
r("ders", "kod","i","k")
r("ders", "ad","i",_)
r("ders", "bölüm","l",_)
r("ders","hocano","n",_)
```

The semantic definitions dictionary keeps the noun and the modifier pair together with their interpretation. There is an entry for each definition. For instance, the entry for *iyi ders* is

```
sps("ders","iyi","hocano","1234","=")
```

Semantic relationships are also kept as database facts. Each fact consists of two related objects , the class value of the second object and the type of the relation. The fact representing that the verb *al* 'give' may have the word *ders* 'course' as object, is stored as

```
ss("al","ders","i","o")
```

Finally, the domain to database mapping table is also stored as a list of facts.

```
fr("ders", "kod", "course", "code")
fr("ders", "ad", "course", "c_name")
fr("ders", "kredi", "course", "credit")
fr("ders", "hocano", "course", "instr_no")
```

VII. CONCLUSION

In this thesis, we have developed a model for a portable natural language database interface system in Turkish. Our model has a two step transformation from natural language to an intermediate meaning representation language and finally to a target database language. We have distinguished two different processing phases, and separated domain dependent and independent parts of the NLI. Our design principle is to have domain independent run time modules for different processing stages and to supply the domain dependent knowledge as a knowledge base to these modules. The language processing component is a general purpose syntactic parser based on the simple principle of general categorization incorporated with the notion of modification between words. The grammar is arranged in a hierarchical structure and each level defines a type of modification. Each syntactically identified noun phrase must pass semantic checks to decide on whether it is meaningful with the act determined by the verb. A domain dependent knowledge about the semantic relationships is supplied to the model for that purpose. This knowledge is used to determine the referents of question pronouns as well. The output of the linguistic component is processed in two stages. In the first stage, a general purpose meaning representation generator is used. The meaning of the sentence is represented in the meaning representation language D&Q. The database schema is used in this process to determine the applicability of the representation in the database. The meaning representation generator is not a simple generator. It has a rule based reasoning capability. It can make analogies based on the domain dependent knowledge to understand queries formulated in user's view as well as selecting the appropriate entity or attribute names for each content word as well as less descriptive words. The D&Q expression is translated into a declarative query language by applying a set of database independent transformational rules, and domain objects are mapped onto actual database files and fields.

Considering pros and cons of two different parsing techniques, namely semantic grammar-based parser and syntactic parser, we combine syntactic and semantic analysis in our work. Application of the semantic processing to the output of the syntactic parser makes the linguistic component domain independent and avoids the disadvantages of a purely semantic parser.

The main advantage of our model is that it consists of separate general purpose run time modules. Each run time module can separately be used. For instance syntactic parser can be used in text translation. Syntactic analysis of Turkish is a very broad subject. We attempt to solve in this work only a limited part of the Turkish grammar. The syntactic parser in our work is sufficiently potent to process a large enough subset of Turkish necessary to parse queries in the NLI. It may be extended by adding more phrase rules and categories to cover a wide range of Turkish grammar. Necessary additions are verb tenses such as past tense and future tense, verb forms for all persons, and the category of adverbs. The correct sequences of noun phrases with different case marker must be incorporated into the syntactic parser[4]. In its current state, no restrictions such as subject and verb agreement and comma restriction are handled. The design of the morphological parser plays an important role in the overall design of the syntactic parser since Turkish is a suffixing language. In our design, the affix stripping approach is used in determining all combinations of inflectional suffixes provided that nouns are listed in the dictionary in drived form rather than two separate lists containing roots and derivational suffixes. Our morphological parser is sufficient for our purpose, but it must be extended to handle exceptions in word formation. Furthermore, vowel harmony is not considered since it does not change the meaning of the word.

The model uses the D&Q language of Wallace as the meaning representation language. D&Q is found to be the most suitable for our purpose since qualifier hierarchy can be easily implemented in D&Q. The syntax of the intermediate representation language is somewhat modified in order to make it closer to the syntax of the declarative database languages. The meaning representation generator can handle a wide range of simple queries. However, it must be elaborated to handle complex sentences whose interpretations are nested queries. The current processing capability can only answer questions that require a single pass over the database.

The meaning representation and parsing are concurrent processes in our design. Once a constituent is parsed its meaning is represented in D&Q, which makes the backtracking impossible. A solution to that is to separate syntactic and semantic analysis from the meaning representation process. That is, the parser will produce an output such as a parse tree. The meaning representation generator will generate a D&Q expression from the output of the parser. This is time consuming when the query is syntactically correct but its meaning can not be expressed in the database.

Portability is the most important concern nowadays in database design. Portability is achieved with the intermediate meaning representation. The domain dependent knowledge is supplied in the form of tables and tables are easy to change. Consequently, our model can be easily adapted to other domains simply by reconstructing the knowledge source.

In its current state, our system has no editor. However an editor like the knowledge acquisition component of TEAM or more generally, a component like the world editor of the KID is necessary in order to facilitate both data entry and the adaptation of the model to a new discourse.

Although our model incorporates some intelligence in question translation , it is not an intelligent database assistant in the sense of generating clarifying dialogues in order to help the user to form correct natural language queries and to correct misconceptions about the database. The system can be further developed by adding to it

(1) a menu based knowledge acquisition component,

(2) capability to work with more than one database, and

(3) more flexible and robust dialogue capabilities.

APPENDIX A. TURKISH GRAMMAR

In first part of this appendix, we give the Turkish grammar used in NLI in terms of rewrite rules. Next its representation as a transition network is given.

A.1. Turkish Grammar Used in NLI in Terms of Rewrite Rules

Notational conventions used are as follows ;

LHS --> RHS	The left-hand side is defined by the right hand side
A | B	An **A** or **B** may occur on the right-hand side
A (B | C) D	This denotes that an **A** on the RHS followed by a **B** or **C** then a **D**
On RHS	terms starting with **lowercase** letters are **dictionary look-ups** the ones with an **uppercase** letter represent another **rule** terms in ***italic*** are **constants** terms preceded by a **minus sign** are **suffixes** **parenthesis** denotes **optionality** of a term
in **Ax**	**x** is the grammatical function of a noun phrase **A**
[]	nothing

The rules :

S $\longrightarrow$ **Nph**$_{\textbf{caseM}}$ **Vph**

Vph $\longrightarrow$ **verb** |

verb + (*-iyor* | *-di*) |

var | *yok* |

(**noun** | **adj** | **verb** + *-en*) + *-dir*

Nph$_{\textbf{caseM}}$ $\longrightarrow$ **Nph**$_{\textbf{caseN}}$ **Nph1**$_{\textbf{caseM}}$ | **Nph1**$_{\textbf{caseM}}$

Nph1$_{\textbf{(caseM/gen)}}$ $\longrightarrow$ **Nph1** $_{\textbf{gen}}$ **Nph1**$_{\textbf{(caseM/gen)+poss}}$

$$Nph1_{(caseM/gen)+(poss)} \longrightarrow Nph1_{(caseM/gen)+(poss)}\ conj\ Nph1_{(caseM/gen)+(poss)}$$

$$Nph1_{(caseM\ |\ gen)+(poss)} \longrightarrow NF_{(caseM\ |\ gen)+(poss)}$$

$$NF_{(caseM\ |\ gen)+(poss)} \longrightarrow$$

$$pronoun_{(caseM\ |\ gen)+(poss)}\ |$$

$$qpronoun_{(caseM\ |\ gen)+(poss)}\ |$$

$$Ng2_{(caseM\ |\ gen)}\ |$$

$$(ASF\ |\ AFS\ |\ (CS)\ (QF)\ (AF))\ Np_{(caseM\ |\ gen)+(poss)}$$

$$Np_{(caseM\ |\ gen)} \longrightarrow Ns\ Ng1_{(caseM\ |\ gen)+poss}$$

$$Ng1_{(caseM\ |\ gen)+(poss)} \longrightarrow$$

$$noun_{(caseM\ |\ gen)+(poss)}\ |$$

$$Ngn1_{(caseM\ |\ gen)+(poss)}\ conj\ noun_{(caseM\ |\ gen)+(poss)}$$

$$Ng2_{(caseM\ |\ gen)} \longrightarrow propernoun_{(caseM\ |\ gen)}\ |$$

$$Ngn2_{(caseM\ |\ gen)}\ conj\ propernoun_{(caseM\ |\ gen)}$$

$$Ngn1_{(caseM\ |\ gen)+(poss)} \longrightarrow noun_{(caseM\ |\ gen)+(poss)}\ |$$

$$Ngn1_{(caseM\ |\ gen)+(poss)}\ noun_{(caseM\ |\ gen)+(poss)}$$

$$Ngn2_{(caseM\ |\ gen)} \longrightarrow propernoun_{(caseM\ |\ gen)}$$

$$Ngn2_{(caseM\ |\ gen)}\ propernoun_{(caseM\ |\ gen)}$$

$$Ns \longrightarrow noun_{case\varnothing}\ |\ Nsn\ conj\ noun_{case\varnothing}\ |\ []$$

$$Nsn \longrightarrow noun_{case\varnothing}\ |\ Nsn\ noun_{case\varnothing}$$

$$AF \longrightarrow adj\ |\ ADF\ conj\ AF$$

$$ADF \longrightarrow adj\ |\ ADF\ adj$$

$$ASF \longrightarrow \textit{en}\ adj\ |\ ASF\ conj\ ASF$$

$$QF \longrightarrow (SupF)\ qualadj$$

SupF $\longrightarrow$ *en* **adjquan**

CS $\longrightarrow$ **CS1** | **CS1 conj CS1**

CS1 $\longrightarrow$ $\mathbf{Nph_{abl}}$ **adjquan** |

$\mathbf{Nph_{\{dat\ |\ case\emptyset\}}}$ (*eşit* | *kadar* | *arasındaki*) |

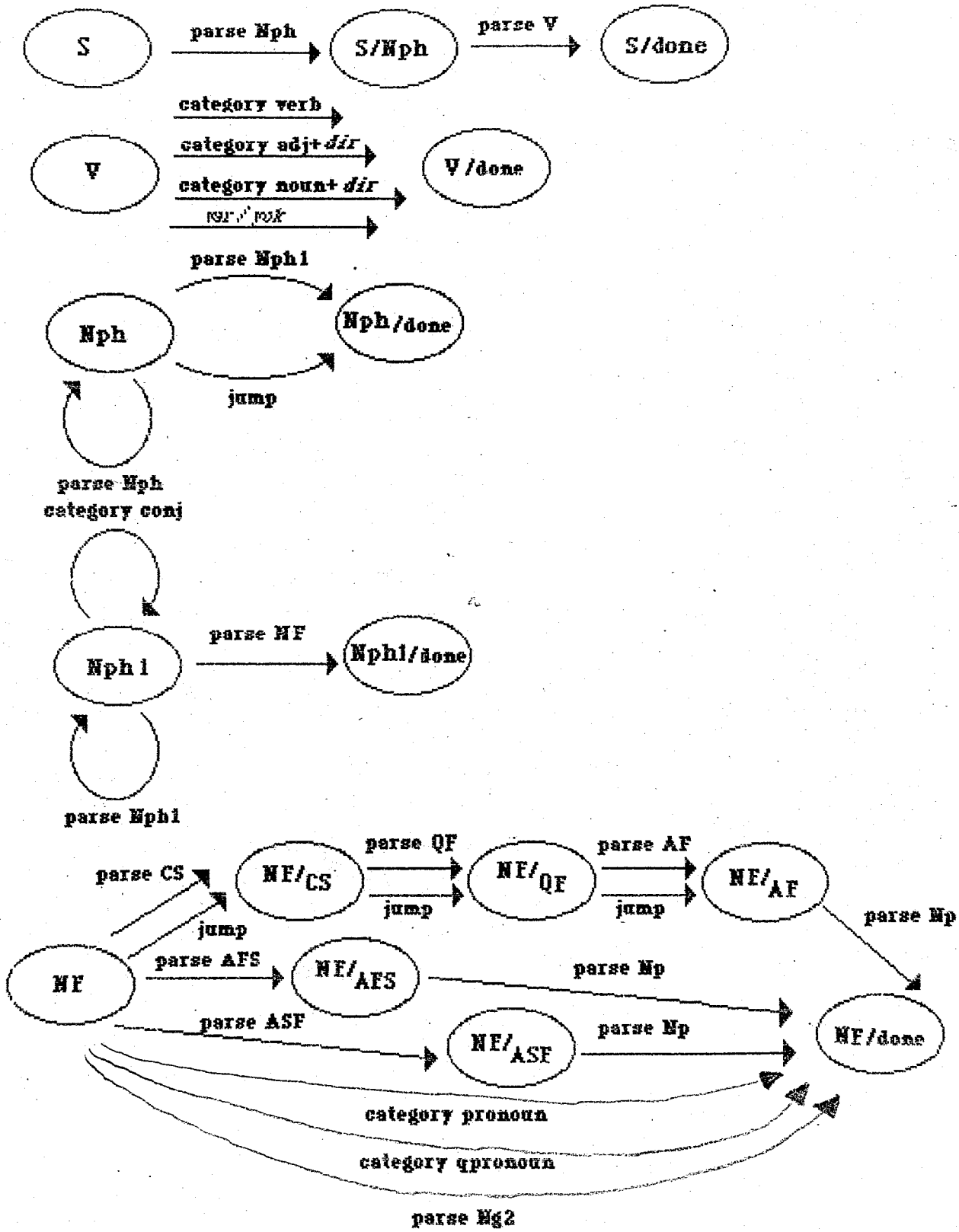
$\mathbf{Nph_{abl}}$ **adj**

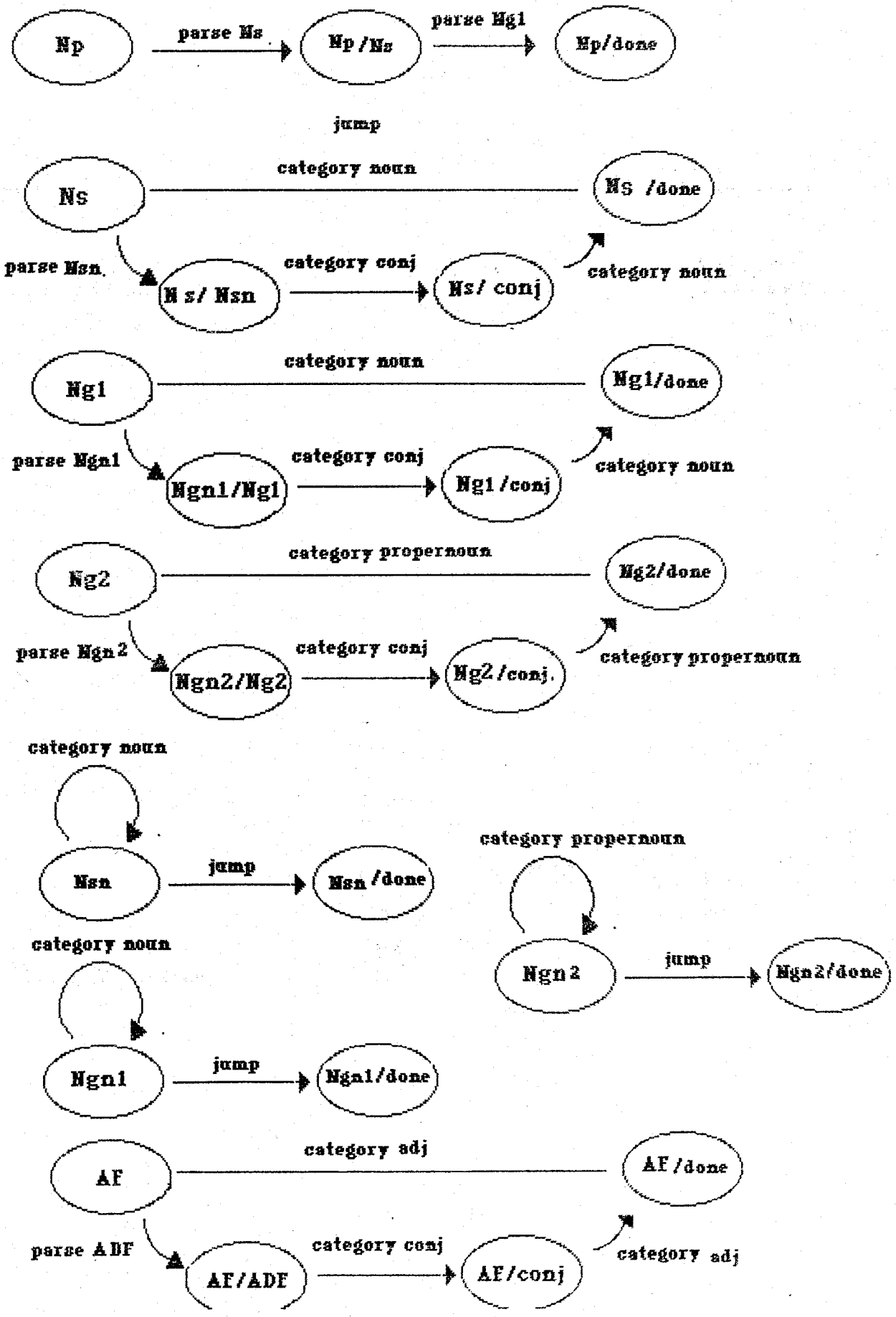
$\mathbf{Nph_{case\emptyset}}$ **noun** + *-li*

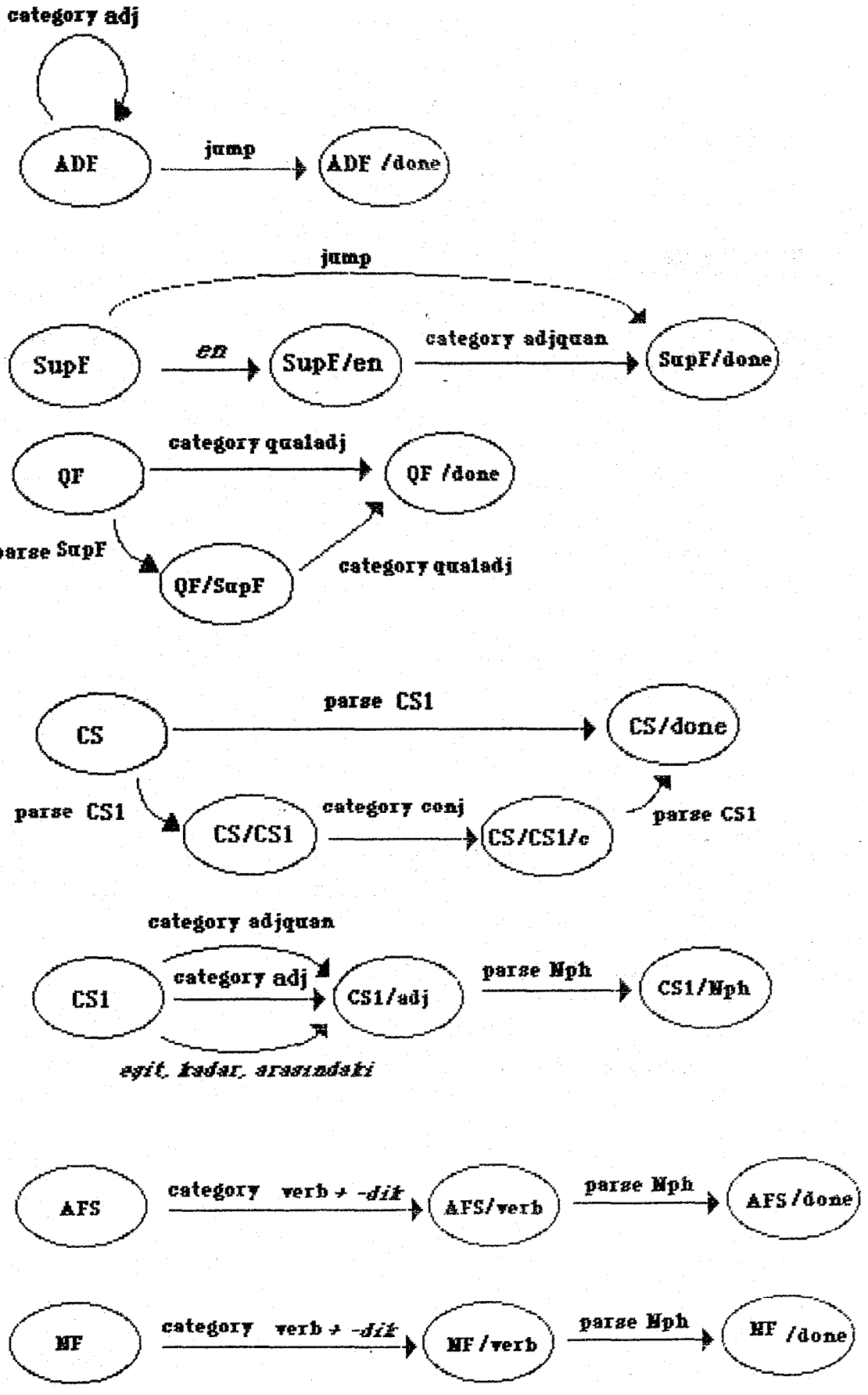
AFS $\longrightarrow$ $\mathbf{Nph_{caseM}}$ **verb** + $\mathbf{\{}$*-en* | *-dik*$\mathbf{\}_{case\emptyset}}$

NF $\longrightarrow$ $\mathbf{Nph_{caseM}}$ **verb** + $\mathbf{\{}$*-en* | *-dik*$\mathbf{\}_{caseM}}$

A.2. Representation of the Grammar as a Transition Network




Np
parse Ns
Np /Ns
parse Ng1
Np/done
jump
Ns
category noun
Ns /done
parse Nsn
N s/ Nsn
category conj
Ns/ conj
category noun
Ng1
category noun
Ng1/done
parse Ngn1
Ngn1/Ng1
category conj
Ng1 /conj
category noun
Ng2
category propernoun
Ng2/done
parse Ngn2
Ngn2/Ng2
category conj
Ng2 /conj.
category propernoun
category noun
Nsn
jump
Nsn /done
category propernoun
Ngn2
jump
Ngn2/done
category noun
Ngn1
jump
Ngn1/done
AF
category adj
AF /done
parse ADF
AF/ADF
category conj
AF/conj
category adj


category adj
ADF
jump
ADF /done
jump
SupF
en
SupF/en
category adjquan
SupF/done
QF
category qualadj
QF /done
parse SupF
QF/SupF
category qualadj
CS
parse CS1
CS/done
parse CS1
CS/CS1
category conj
CS/CS1/c
parse CS1
category adjquan
CS1
category adj
CS1/adj
parse Nph
CS1/Nph
egit, kadar, arasındaki
AFS
category verb → -dik
AFS/verb
parse Nph
AFS/done
MF
category verb → -dik
MF/verb
parse Nph
MF /done

APPENDIX B. SYNTAX OF D&Q

This appendix gives the full syntax of D&Q as it is defined in Wallace's book. Two major changes made in the syntax are listed separately.

B.1. Syntax of Wallace's D&Q

LHS ::- RHS	The left-hand side is defined by the right-hand side.
‹Non Terminal›	On the RHS, nonterminals are put in angled brackets.
A | B	An 'A' or 'B' may occur here on the RHS.
A (B | C) D	This is an 'A' on the RHS followed by a 'B' or 'C' then a 'D'.
nil	Nothing

The full syntax of D&Q is:

QUALIFIER	::-	‹PREDICATE› | ‹PREDICATE› (‹SELECTIONS›).
QUALIFIER	::-	‹QUALIFIER› (& | OR) ‹QUALIFIER› | not ‹QUALIFIER›.
QUALIFIER	::-	‹DESCRIPTION› is qual(‹VARIABLE›, ‹QUALIFIER›).
QUALIFIER	::-	true | fail.
DESCRIPTION	::-	‹CONSTANT›.
DESCRIPTION	::-	‹DETERMINER›-‹COUNT›-qual (‹VARIABLE›, ‹QUALIFIER›).
DESCRIPTION	::-	‹DESCRIPTION› is funct (‹VARIABLE›, ‹DESCRIPTION›).
DESCRIPTION	::-	‹DESCRIPTION› (& | OR) ‹DESCRIPTION›.
SELECTION	::-	‹ATTRIBUTE› ‹COMPARISON› ‹VARIABLE› (,‹SELECTIONS›|nil).
COMPARISON	::-	= | ≠ | < | > | <= | >=.
DETERMINER	::-	the | any | what.
COUNT	::-	‹INTEGER› | ‹VARIABLE›.
CONSTANT		is a PROLOG atom or integer.
INTEGER		is a PROLOG integer.
PREDICATE		is a PROLOG atom.
ATTRIBUTE		is a PROLOG atom.
VARIABLE		is a PROLOG variable.

B.2. Extensions Made in the Syntax of D&Q

The extensions we made in the syntax of D&Q mainly concern the definitions of the DESCRIPTION and SELECTION, their new structures are as follows

DESCRIPTION ::- <DETERMINER>-<COUNT>-qual (<VARIABLE>,< **DISPLAYLIST**>, <QUALIFIER>).

SELECTION ::- **{ <ATTRIBUTE> | <VARIABLE>}** <COMPARISON> **{ <ATTRIBUTE> | <VARIABLE>}** {,<SELECTIONS> | nil }.

APPENDIX C. EXAMPLE DATABASE

In this appendix, we give the actual database definition and its representation in user's own view.

The actual files and fields in the database are as follows:

STUDENT	st_no	st_name	dept	birth

INSTRUCTOR	instr_no	instr_name	dept	office

COURSE	code	c_name	credit	instr_no

STUD_COURSE	st_no	code	grade

where the **st_no, code, instr_no, st_no,** and **code** are keyfields for **student, instructor, course, and stud_course** respectively.

The user's own view is

öğrenci(num, ad, bölüm, doğum)

hoca(hocano, ad, bölüm, ofis)

ders(kod, ad, kredi, hocano)

karne(num, kod, not)

APPENDIX D. LISTING OF THE NLI IMPLEMENTATION AND DATA FILES

This appendix gives the list of source file names that exist in the program diskette and the data files used for the implementation.

D.1. Program Listing

The program diskette contains the following source codes of the five modules :

CW.PRO : morphological parser

SYN.PRO : syntactic parser

QG.PRO : query generator

CORR.PRO : spelling corrector

SQLLIST.PRO : translator

The global predicates are listed in the file **GLOBDEF.PRO**.In addition to these source files the program diskette contains an executable file **KEYBTR.COM** which contains the following key assignments for a Turkish keyboard.

q --> ı

x --> ü

] --> ö

[--> ç

~ --> ş

? --> ğ

One has to run the keybtr.com and combine source files and globdef.pro into a stand-alone project in Turbo Prolog 2.0.

D.2. Data files

Basic Lexicon

```
noun("ortalama","n")
noun("sayı","n")
noun("toplam","n")
noun("yarı","n")
noun("kimya","i")
noun("fizik","i")
pron("o")
pron("onlar")
pron("bu")
pron("bunlar")
adj("zor")
adj("büyük")
adj("küçük")
adj("çok")
adj("iyi")
adj("kötü")
verb("ver")
verb("göster")
verb("yaz")
verb("al")
qpron("kim")
qpron("hangi")
qpron("hangisi")
qpron("kaç")
qpron("ne")
qualadj("bu","d")
qualadj("o","d")
qualadj("hangi","q")
qualadj("ne","q")
qualadj("kaç","q")
qualadj("hiçbir","h")
qualadj("hiç","h")
qualadj("bütün","h")
qualadj("her","h")
conj("ve")
conj("veya")
conj("ile")
quanadj("fazla")
quanadj("az")
quanadj("çok")
comp("büyük",">")
comp("küçük","<")
comp("eşit","=")
comp("aynı","=")
comp("kadar","=")
comp("çok",">")
comp("az","<")
```

```
comp("alçak","<")
comp("yüksek",">")
comp("iyi",">")
comp("kötü",">")
comp("düşük","<")
comp("fazla",">")
```

Conceptual Database Schema in User's View

```
r("öğrenci","num","i","k")
r("öğrenci","ad","i","")
r("öğrenci","bölüm","1","")
r("öğrenci","doğum","n","")
r("hoca","hocano","i","k")
r("hoca","ad","i","")
r("hoca","bölüm","1","")
r("hoca","ofis","1","")
r("ders","kod","i","k")
r("ders","ad","i","")
r("ders","kredi","n","")
r("ders","hocano","i","")
r("karne","num","i","k")
r("karne","kod","i","k")
r("karne","not","n","")
```

Synonym Lexicon

```
syn("num","numara","n","n")
syn("num","no","n","n")
syn("öğrenci","talebe","p","n")
syn("ad","isim","i","n")
syn("ofis","oda","1","n")
syn("not","derece","n","n")
syn("hoca","öğretmen","p","n")
```

Dictionary containing semantic definitions

```
sps("not","iyi","not","5",">=")
sps("ders","iyi","hocanum","1234","=")
sps("hoca","iyi","ad","Ahmet","=")
```

Semantic Relationships Lexicon

```
ss("al","ders","i","o")
ss("al","not","n","o")
ss("al","hoca","p","s")
ss("al","öğrenci","p","a")
ss("ver","ders","i","o")
ss("ver","not","n","o")
ss("ver","hoca","p","a")
ss("ver","öğrenci","p","t")
ss("not","ders","i","s")
ss("not","ders","i","p")
ss("ders","bölüm","i","p")
```

Domain to Database Mapping Table

```
fr("öğrenci","num","student","st_no")
fr("öğrenci","ad","student","st_name")
fr("öğrenci","bölüm","student","dept")
fr("öğrenci","doğum","student","birthday")
fr("hoca","hocano","instructor","instr_no")
fr("hoca","ad","instructor","instr_name")
fr("hoca","bölüm","instructor","dept")
fr("hoca","ofis","instructor","office")
fr("ders","kod","course","code")
fr("ders","ad","course","c_name")
fr("ders","kredi","course","credit")
fr("ders","hocano","course","instr_no")
fr("karne","num","stud_course","st_no")
fr("karne","kod","stud_course","code")
fr("karne","not","stud_course","grade")
```

BIBLIOGRAPHY

1. E. Charniak, D. McDermott, "Introduction to Artificial Intelligence," *Addison-Wesley*, 1985.

2. G. Jakobson, C. Lafond, E. Nyberg, and G. Piatetsky-Shapiro, " An Intelligent database Assistant," *IEEE EXPERT*, pp. 65-77, Summer 1986.

3. H. Ishikawa, Y. Izumida, T. Yoshino, and A.Makinouchi. "KID Designing A Knowledge-Based Natural Language Interface," *IEEE EXPERT*, pp. 57-70 ,Summer 1987.

4. HM. Meskill,"A Transformational Analysis of Turkish Syntax" *Mouton Publishers*, 1970.

5. M. Wallace, "COMMUNICATING WITH DATABASES IN NATURAL LANGUAGE," *Ellis Horwood Series in Artificial Intelligence*, 1983.

6. D. L. Waltz. "An English Language Question Answering System for a Large Relational Database," *Communications of ACM*. Vol. 21 No. 7,pp. 526-539,July 1978

7. G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, and J. Slocum, "Developing A Natural Language Interface to Complex Data," *ACM Transactions on Database Systems*, Vol. 3, No.2, pp. 104-147, October 1977.

8. B. J. Grosz, "TEAM: A Transportable Natural Language Interface System" *Proc. Conf. Applied Natural Language Interface*, pp. 39-45, 1983.

9. B. H. Thompson, F. B. Thompson "Rapidly Extendable Natural Language." *Proceedings of ACM 78 Annual Conference*, New York, pp. 173-182 ,1978

10. F. B. Thompson,P. C Lockermann, B. H. Dostert and R. Deverill "REL "RAPIDLY EXTENSIBLE LANGUAGE SYSTEM," *in Proceedings of the 24th ACM National Conference*, New York, 1969, pp 399-417..

11. M. Templeton, J. Burger "Problems in Natural Language Interface to DBMS with Examples From EUFID," *Proc. Conf. Applied Natural Language Processing*, Santa Monica, pp.3-16 February 1983

12. Artificial Intelligence Corp. "INTELLECT Query System User's Guide," 500 Fifth Ave, Waltham, Mass. 02254, 1980.

13. L. R. Harris, "The ROBOT System; Natural Language Processing Applied to Database Query," *Proceedings of ACM 78 Annual Conference*, New York, pp. 165-172, 1978.

14. S.J.Kaplan , "Designing A Portable Natural Language System" *ACM Transactions on Database Systems*, Vol. 9, No. 1, March 1985.

15. B. K.Boguraev, K.S. Jones, "How to Drive a Database Front End Using General Semantic Information," *Proc. Conf. Applied Natural Language Processing*, Santa Monica, pp. 81-88 February 1983 .

16. N. Sager, "Natural Language Information Processing," *Addison-Wesley*, 1981.

17. J. Hankamer, "Parsing Nominal Compounds in Turkish," *in Morphology as a Computational Problem UCLA Occasional Papers 7*, ed. Karen Wallace, UCLA pp.123-143, 1988.

18. J. Hankamer, "Morphological Parsing and the Lexicon", in Lexical Representation and Processing, ed. W. M. Wilson, *MIT Press*, 1988.

19. J.Hankamer, "Finite State Morphology and Left to Right Phonology", Proceedings of the West Coast Conference on Formal Linguistics, Vol. 5, Stanford University, 1986

20. Turbo Prolog Reference Manual, Version 2.0 *Borland International, Inc.*, 1988.

21. Turbo Prolog User's Guide, Version 2.0 *Borland International, Inc.*, 1988.

REFERENCES NOT CITED

1. P. C.Culicover, "Syntax," Second Edition, *Academic Press Inc.* 1982

2. T.Winograd , "Language as a Cognitive Process," Vol. 1 Syntax, *Addison-Wesley*, 1983

3. TN. Gencan "Dilbilgisi" *Kanaat Yayınları*, 1979.

4. R. Underhill "Turkish Grammar" *MIT Press*, 1976.

5. C. J. Date "An Introduction to Database Systems", Vol. I, Fourth Edition, *Addison-Wesley Company*, 1986