# A TWO-LEVEL MORPHOLOGICAL ANALYZER FOR TURKISH LANGUAGE

**by**
**Hülya ÇETİN İÇER**

**September, 2004**
**İZMİR**

# A TWO-LEVEL MORPHOLOGICAL ANALYZER FOR TURKISH LANGUAGE

**A Thesis Submitted to the**
**Graduate School of Natural and Applied Sciences of**
**Dokuz Eylül University**
**In Partial Fulfillment of the Requirements for**
**the Degree of Master of Science in Computer Engineering**

**by**
**Hülya ÇETİN İÇER**

**September, 2004**
**İZMİR**

**M.Sc THESIS EXAMINATION RESULT FORM**

We certify that we have read the thesis, entitled **“A TWO-LEVEL MORPHOLOGICAL ANALYZER FOR TURKISH LANGUAGE”** completed by **HÜLYA ÇETİN İÇER** under supervision of **ASSIST PROF. DR. ADİL ALPKOÇAK** and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

................................................................................

**Assist. Prof. Dr. Adil ALPKOÇAK**

**Supervisor**

| | |
|---|---|
| ................................................................................ | ................................................................................ |
| **Prof. Dr. Tatyana YAKHNO** | **Prof. Dr. Bahar KARAOĞLAN** |
| Committee Member | **Committee Member** |

Approved by the
Graduate School of Natural and Applied Sciences

Prof. Dr. Cahit HELVACI
Director

# ACKNOWLEDGMENTS

I would like to thank my supervisor Assist. Prof. Dr. Adil Alpkoçak for guidance, suggestions. He gave me great ideas and concentrate about this thesis.

I also thank to the Committee Members for their efforts and advices.

I am grateful to my family, my father Necdet ÇETİN, my mother Ferhunde ÇETİN and my brother Cem ÇETİN for their infinite moral support and help throughout my life.

And I preserved my special thank to my husband Oğuz Kaan İÇER because of supporting and motivating me on each step of thesis.

Hülya Çetin İÇER

# ABSTRACT

In this study, a morphological analyzer tool is developed for Turkish language based on two-level model of morphology. The tool analyses surface forms and returns all alternations of stems, suffixes and their types by using the two-level rules, dictionary and morpheme order rules based on nominal and verbal model of the Turkish language. The project also represents a visual interface to help analyzing and debugging process. All alternations of results and the steps of processes are shown as tree structures in XML format as well as all required Turkish rule definitions, words and suffixes.

**Keywords:** morphology, morphotactics, morphophonemics, two-level description of morphology, natural language processing, Turkish morphology

# ÖZET

Bu tez çalışmasında Türkçe sözcükleri iki düzeyli model kullanılarak biçimbilimsel çözümleyebilen bir araç geliştirilmiştir. Araç, girilen kelimenin olası tüm gövdelerini, tüm eklerini ve bunların türlerini bulur. Uygulama temek olarak ikidüzeyli biçimbilimsel kuralları, sözlüğü ve eklerin sıralanışını ifade eden kuralları kullanır. Eklerin sıralanışını ifade eden kurallar Türkçe'nin isim ve fiil modeline dayanmaktadır. Bu tez kelimeleri biçimbilimsel olarak çözümleyebilen ve bu çözümlemenin adımlarını izlemeye olanak veren bir görsel arabirimle desteklenmiştir. Uygulamanın kullandığı tüm veriler yanında, çözümleme sonuçları ve bu sonuçlara ulaşırken izlenen adımlar da XML formatında saklanmıştır.

**Anahtar Sözcükler:** biçimbilim, biçimdizim, biçimbirim değişmeleri, iki düzeyli biçimbilimsel model, doğal dil işleme, Türkçe biçimbilim

# CONTENTS

# Chapter Two
# MORPHOLOGICAL ANALYSIS

# Chapter Three
# TURKISH MORPHOLOGY

# Chapter Five
# SOFTWARE DESIGN AND IMPLEMENTATION

# LIST OF TABLES

# LIST OF FIGURES

**Page**

# CHAPTER ONE
# INTRODUCTION

Turkish is an agglutinative language and belongs to Altaic languages group. The number of words in these languages is much more than the number of words in the vocabularies. Word structures can grow to an unmanageable size so Turkish morphology is very complex more over there are many exceptional cases in Turkish.

Turkish has been quite popular in linguistics literature but there have been very few computational studies in the past. The most important methods are Hankamer's Keçi Project for Turkish (Hankamer, 1986), PC KIMMO for Finnish (Antworth, 1990) and Ample for Quechua (Weber et al., 1988). The most popular analyzer is PC Kimmo. It uses the root driven approach. Because of this, PC KIMMO analyzer can be used for Turkish.

This thesis presents an implementation of a morphological analyzer for Turkish. This project aims to reach the stem of a word and all suffixes and determine the types of stem and suffixes. This project implementation is based on PC KIMMO structure. Turkish rule definitions in this project have been taken from Oflazer's project. (Oflazer, 1993)

PC KIMMO analyzer runs under MS-DOS based platforms and UNIX systems in general. Our thesis is developed by Borland C++ Builder Version 6.0. It aims to develop a new visual tool for analyzing words and debugging its processes. While users debugging its processes they understand the wrong rules and right rules or wrong and right root and suffixes. Additional two rules are developed to analyze the morpheme order according to nominal and verbal model of the Turkish language.

This project consists of two parts: a library to analyze words and an example application that uses this library. The library is a reusable software tool for analyses of Turkish text. The application is developed for testing purposes. This library analyzes the surface form of the word and returns stem and all suffixes and the types of them. All data are stored in XML documents. All data are stored in the text documents for PC-KIMMO project. Reading and understanding text documents is difficult than XML documents. Because of this XML documents are used in this project to store data.

Some Turkish letters can not be used as original letters in PC-KIMMO project. For example “S” is used instead of Turkish letter “ş” or “C” is used instead of Turkish letter “ç”. In this project all Turkish letters are used as original letters. All letters that can not be used in PC-KIMMO are the following: “O” as “ö”, “U” as “ü”, “S” as “ş”, “C” as ç”, “I” as “ı”, “G” as “ğ”.

## 1.1 Review of Related Works

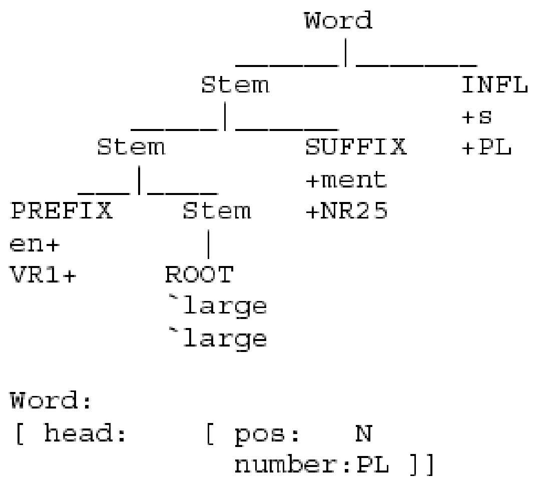
PC KIMMO is an implementation of the two-level model of morphology. Koskenniemi’s model is “two-level” that a word is represented as a direct, letter for letter correspondence between lexical and surface form. (Antworth, 1995)

Example:

| | |
|---|---|
| Surface Form: | e k m e ğ 0 i m |
| Lexical Form: | e k m e k + H m |

PC KIMMO has two main functions: *generator* and *recognizer*. Surface form is an input to recognizer function and returns a lexical form. Lexical form is the input to generator function, which returns surface form.

PC KIMMO version 1 is produced in 1990. It is written in C and ran on the personal computers, Macintosh and UNIX. Version 1 could not directly determine the part of speech of a word. Example: PC KIMMO could tokenize the word “enlargements” into the sequence of morphemes “en-large-ment-s”. It can gloss each morpheme but it could not determine entire word was a plural noun. PC KIMMO version 2 is produced to correct this deficiency in 1993 and a word grammar is added to version 2. The word grammar provides parse trees and feature structures. Version 2 returns the input's word parse tree and feature structure shown in Figure 1.1. (Antworth, 1995)



```
                         Word
                   ______|_______
                Stem              INFL
          _____|______            +s
      Stem           SUFFIX       +PL
   ___|____          +ment
PREFIX    Stem       +NR25
en+        |
VR1+      ROOT
          `large
          `large

Word:
[ head:     [ pos:    N
              number:PL ]]
```

**Figure 1.1 Parse tree and feature structure for word "enlargements"**

## 1.2 Thesis Organization

This thesis includes four chapters except introduction chapter and conclusion. The thesis is organized as follows:

Chapter two gives information about morphological analysis. Firstly, morphology is explained. And then it describes two-level model of morphology in detail. And finally it gives general information about finite state machines, regular expressions, formal languages and regular languages.

Chapter three gives information about Turkish morphology. Firstly, the specifications of Turkish language are explained. And then Morphophonemic's and Morphotactics of Turkish are explained.

Chapter four gives rule definitions of Turkish language. These Turkish rule definitions are used in this project.

Chapter five gives information about this project, its properties and implementation details. It describes how analyzer works and the user interfaces.

# CHAPTER TWO
# MORPHOLOGICAL ANALYSIS

## 2.1 Morphology

In general, morphology is the study of word structure, or meaningful components of words. The smallest meaningful components are called morphemes. Morphology is also interested in how morphemes can be combined to form words.

The first question is what meaning bearing units are. We can say that “kuşlar” has two units. One of them is main meaning of word. In this example “kuş” is the main meaning of the word. These morphemes are called stems; other morphemes are also called as affixes. In this example “lar” is an affix.

The second question is how morphemes can be combined to form words. There are two kinds of processes to combine morphemes to form words: *inflection* and *derivation*. So morphology is generally divided into two types.

### 2.1.1 Inflectional Morphology

Inflectional morphology covers the variant forms of nouns, verbs, etc. Inflectional process is adding grammatical affixes to word stem. It doesn’t change the class of stem. It changes in:

Person (like first, second, etc.)
Tense (like present, future, etc.)
Number (singular or plural)
Gender (Male, female or neuter)

Adding a plural affix ("lar") to a noun stem is an inflectional process.

| Stem | | Affix | | Word |
|---|---|---|---|---|
| kuş | + | -lar | = | kuşlar |
| bird | + | -s | = | birds |

Here "kuş" and "kuşlar" are in the same class which is noun.

English nouns have only two kinds of inflection that are plural and possessive. But in Turkish, there are more kinds of inflection.

### 2.1.2 Derivational Morphology

Derivational morphology is the formation of a new word. This process is simply an affix addition to a word stem. It may change the class of the stem in some cases. After derivation, the resulting class may be different from the stem. For example, in the word "kalemlik", the affix "-lik" is a derivational morpheme. It changes the meaning of the word while it doesn't change the class of stem. Because, "kalem" and "kalemlik" are both noun.

| Stem | | Affix | | Word |
|---|---|---|---|---|
| kalem | + | -lik | = | kalemlik |
| Noun | | Affix | | Noun |

But in the next example, the affix "-iş" is a derivational morpheme in the word "geliş". It changes both the meaning of the word and class of the stem. Because "geliş" is a noun while "gel" is a verb.

| Stem | | Affix | | Word |
|---|---|---|---|---|
| gel | + | -iş | = | geliş |
| Verb | | Affix | | Noun |

A very common way of derivation in English is the formation of new nouns from verbs or adjectives. This process is called *nominalization*.

| <u>Stem</u> | | <u>Affix</u> | <u>Word</u> |
|---|---|---|---|
| computerize | -ation | computerization | |
| Verb | | Affix | Noun |

In Turkish there are many kinds of derivation.

## 2.2 Two-Level Model of Morphology

Two-level morphology is a general computational model for word-form recognition and generation. It is used to analyze the morphology of languages. Kimmo Koskenniemi is a Finnish computer scientist who developed a model for two-level morphology in his Ph.D. thesis in 1983. It is called KIMMO system. It was a major breakthrough in the field of morphological parsing. (Antworth, 1995, pp.2).

Two-level morphology was the first general model. According to Koskenniemi’s studies two-level morphology is based on three ideas:

- Rules are the symbol-to-symbol constraints and rules are applied in parallel.
- The constraints can refer to the lexical and surface context or to both contexts at the same time.
- Lexical lookup and morphological analysis are performed in tandem.

Koskenniemi's model is "two-level" in the sense that a word is represented as a direct, letter-for-letter correspondence between its lexical or underlying form and its surface form. “The lexical level denotes the structure of the functional components of a word while the surface level denotes the standard orthographic realization of the word with the given lexical structure.” (Oflazer, 1993, p.2)

For example, the word *"ekmeğim"* is given in this two-level representation (where "+" is a morpheme boundary symbol and "0" is a null character):

| | |
|---|---|
| Lexical form: | e k m e k + H m |
| Intermediate form: | e k m e ğ 0 i m |
| Surface form: | e k m e ğ i m |

Surface form is an input to recognizer function and returns a lexical form. Lexical form is the input to generator function which returns surface form.

KIMMO parser has two main components. The one of them is rule component and the other one is lexical component or lexicon. Rules component consist of two-level rules. The lexicon lists the all morphemes in their lexical form. All morphemes consist stems and affixes.

The main components of KIMMO are shown in Figure 2.1. (Antworth, 1995, pp.2). KIMMO has two processing functions: *generator* and *recognizer*.

In this thesis, a recognizer function has been implemented to part the word into stem and affixes. But the generator function is not implemented in this thesis.

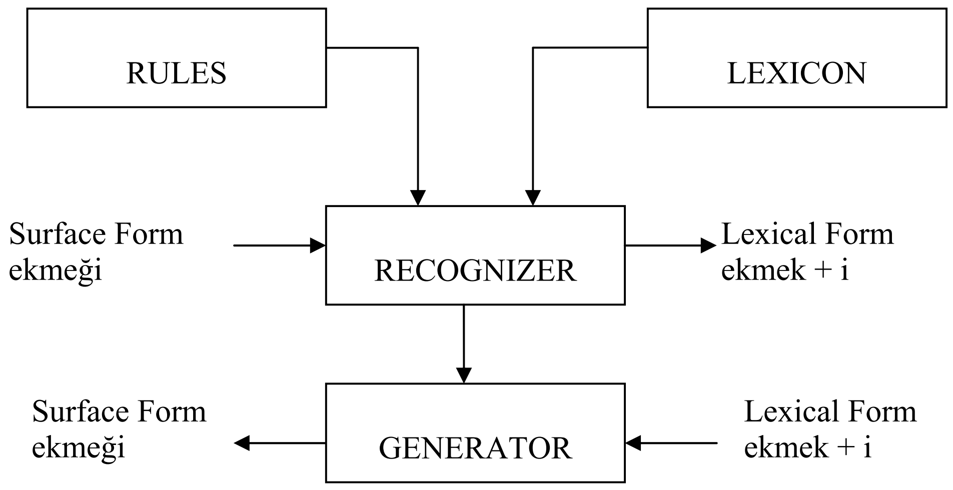


**Figure 2.1 Main components of Karttunen's KIMMO parser**

### 2.2.1 History of Two-Level Morphology

Twenty years ago there was no general language-independent method about morphological analysis. There were some simple cut-and-paste programs to analyze strings in particular languages. These programs were not reversible. Generative phonologists who lived in that time described morphological alternations by means of ordered rewrite rules, but it was not understood how such rules could be used for analysis. (Oflazer, 1993)

Koskenniemi defined formalism for two-level rules in 1983. The semantics of two-level rules were well defined. Koskenniemi and practitioners had to compile rules by hand into finite-state transducers because there was no rule compiler available at that time. So complex rules take hours of effort to compile and test. (Karttunen, 2001)

The first two-level rule compiler was written in InterLisp by Koskenniemi and Karttunen in 1985-87. They used Kaplan's implementation of the finite-state calculus. Lauri Karttunen, Todd Yampol and Kenneth R. Beesley developed the current C-version of the compiler, which is based on Karttunen's 1989 Common Lisp implementation in consultation with Kaplan at Xerox PARC in 1991-92.(Karttunen, 2001)

In 2002, Kemal Oflazer described a full two-level morphological description of Turkish word structures. The phonetic rules of contemporary Turkish have been encoded using 22 two-level rules. These rules cover almost all the special cases, and exceptions about Turkish words. In our study, Oflazer's rules are applied.

### 2.2.2 The Complexity of Two-Level Morphology

"The use of finite-state machinery in the 'two-level' model by Kimmo Koskenniemi gives it the appearance of computational efficiency, but closer examination shows the model doesn't guarantee efficient processing." (Barton, 1986, pp 1).

In two level systems the general problem is extensive backtracking process. NULL characters are used to insert and delete process. If NULL characters are excluded, problems are NP-complete in the worst case. If NULL characters are completely unrestricted, the problem is harder.

The next subsection presents how two level rules are used.

### 2.2.3 Two-Level Rules

Two-level model is defined as a set of correspondences between lexical and surface representation. There is a similarity between two-level rules and the rules of standard generative phonology. There is a difference in several crucial ways at the same time.

Rule1 is an example of a generative rule:

Rule1 a --> c / ___ d

Rule2 is an example of the analogous two-level rule: Rule2 a : c => ___ d

Their meanings and notation are different. Two level rules are declarative and bidirectional. They state that certain correspondences hold between a lexical (that is, underlying) form and its surface form. Lexical form represents a simple concatenation of morphemes making up a word and surface form represents the spelling of the word. Figure 2.2 shows an example of lexical, intermediate and surface tapes.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Lexical Form | | k | i | t | a | p | + | H | m | |
| Intermediate Form | | k | i | t | a | b | 0 | ı | m | |
| Surface Form | | k | i | t | a | b | ı | m | | |

**Figure 2.2 Example of lexical, intermediate and surface tapes**

Rule2 states that lexical “a” corresponds to surface “c” before “d”; it is not changed into “c”, and it still exists after the rule is applied. Because two-level rules express a correspondence rather than rewrite symbols, they apply in parallel rather than sequentially. Thus no intermediate levels of representation are created as artifacts of a rewriting process. Only the lexical and surface levels are allowed.

The two-level rules deal with each word as a correspondence between its lexical representation (LR) and its surface representation (SR).

For example:

| | |
|---|---|
| Lexical Representation: | a b a d |
| Surface Representation: | a b c d |

PC-KIMMO uses the notation “lexical character: surface character”, for instance “a:a”, “b:b” or “k:ğ”. There are two types of correspondences. One of them is default correspondences like a:a and the other one is special correspondences like “k:ğ” and “ç:c”. The all of the default and special correspondences make up the set of feasible pairs. All feasible pairs must be explicitly declared in the description.

Generative rules have three main characteristics:

- They are transformational rules. They convert or rewrite one symbol into another symbol. Rule Rule1 states that “a” becomes (is changed into) “c” when it precedes “d”. After rule Rule1 rewrites “a” as “c”, “a” no longer exists.

- Sequentially applied generative rules convert underlying forms to surface forms via any number of intermediate levels of representation; that is, the application of each rule results in the creation of a new intermediate level of representation.

- Generative rules are unidirectional. They can only convert underlying form to surface form, not vice versa.

### 2.2.4 Two-Level Rule Notation

A two-level rule is made up of three parts:

1. Correspondence,
2. Rule operator,
3. Environment or context.

#### 2.2.4.1 Correspondence

The correspondence "a : c" is the first part of the rule Rule2. Correspondence is a pair of lexical and surface characters. Correspondence has the same meaning with correspondence pair. The first part of rule Rule2 is the correspondence "a : c". It specifies a lexical "a" that corresponds to a surface "c".

If the lexical and surface characters of a correspondence pair are identical, the correspondence can be written as a single character. "__d" is the short notational form of "__ d : d". Rule3 is the full form of Rule2. So Rule2 is equivalent to Rule3.

Rule3 a : c => ___ d:d

Rule4 a:c => d:d ___

Rule3 and Rule4 are different from each other because of notation "___". In this notation, "___d" means any character is accepted before character "d". This notation "d___" means any character can be after d.

There are two types of correspondences: Default correspondence and special correspondence. They will be explained in subsection 2.2.5.2.2.

### 2.2.4.2 Rule Operator

The rule operator “=>” is the second part of Rule2. There are four operators: =>, <=, <=>, /<=. These operators are shaped like an arrow. Rule operators determine the relationship between the correspondence and the environment.

Semantics of the rule operators:

“=>” means “only but not always”
“<=“ means “always but not only”
“<=>” means “always and only”
“/<=” means “never”

The rule operator specifies the logical relation between the correspondence and the environment of a two-level rule.

Four different rule types are used to represent the phonetic restrictions:

a:b => LC __RC
a:b <= LC __RC
a:b <=> LC __RC
a:b /<= LC __RC

Here, LC means left context and RC means right context.

### 2.2.4.3 Environment or Context

The third part of the Rule2 is the environment or context, written as “__d”. As in standard phonological notation, an underline, called an environment line, denotes the position of the correspondence in the environment.

### 2.2.5 Rule Types

There are four types of rule:

- The Context Restriction Rule: a:b => LC __RC

Rule2 a : c => ___ d

Rule2 is written with the rule operator "=>". The "=>" operator means the correspondence only occurs in the environment. Rule2 states that lexical "a" corresponds to surface "c" only preceding "d", but not necessarily always in that environment. Thus other realizations of lexical "a" may be found in that context, including "a:a". The "=>" operator means context does not necessarily imply the correspondence. It means that the "=>" rule is an optional rule. Rule2 would be used if the occurrence of "a" and "c" freely varies before "d". If the surface input form is "abcd" recognizer will produce both lexical form "abcd" and "abad". To state it negatively, Rule2 prohibits the occurrence of the correspondence "a:c" everywhere except preceding "d".

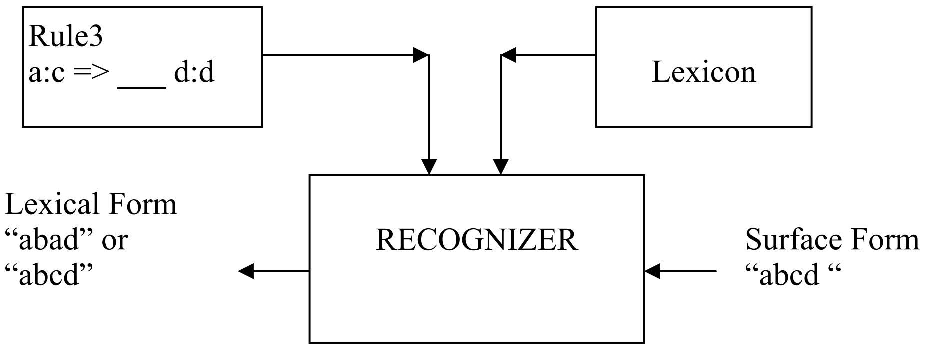


**Figure 2.3 Example of context restriction rule**

Example Rule "g:ğ => _ +:0 (X:0) VOWEL" (Oflazer, 1993)

When a word ending with "g" and certain suffixes are added then the "g" may become "ğ".

| | |
|---|---|
| Surface form: | dialoğa |
| Intermediate form: | dialoğ00a |
| Lexical form: | dialog+yA |

- The Surface Coercion Rule: a:b <= LC __RC

The "<=" operator means the correspondence always occurs in the environment. Rule4 states that lexical "a" always corresponds to surface "c" proceeding "d", but not necessarily only in that environment. The "<=" operator is approximately equivalent to an obligatory rule in generative phonology. It means that the context implies the correspondence, but the correspondence doesn't necessarily imply the context. To state it negatively, if "a:¬c" (where "¬c" means the logical negation of "c") means the correspondence of lexical "a" to surface not-c (that is, anything except "c"), then Rule4 prohibits the occurrence of "a:¬c" in the specified context.

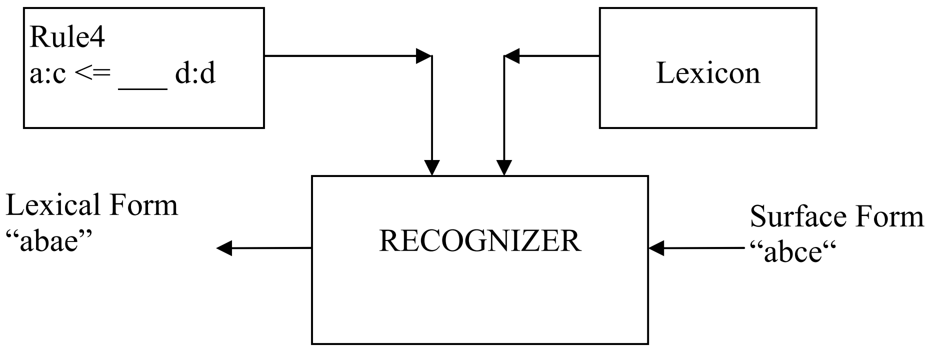


**Figure 2.4 Example of surface coercion rule**

There is no example rule for this operator in Turkish.

- The Composite Rule a:b <=> LC __RC

Rule5 a : c <=> ___ d:d

The “<=>” operator means the correspondence always and only occurs in the environment. The “<=>” operator is the combination of the operators “<=” and “=>”. Rule5 states that lexical “a” corresponds to surface “c” always and only preceding “d”. If this operator is used when a correspondence obligatory occurs in a given environment and in no other environment and the correspondence is allowed if and only if it is found in the specified context.

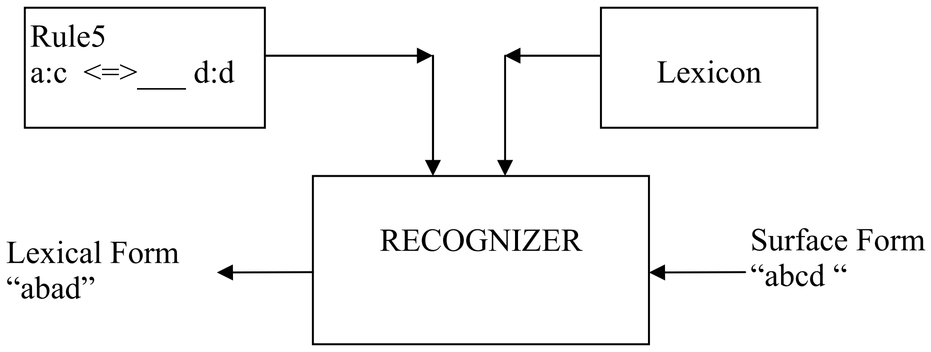


**Figure 2.5 Example of composite rule**

Example Rule “H:0 <=> VOWEL:VOWEL (':') +:0 _” (Oflazer, 1993)

If the last character of the stem is a vowel and the first character of the morpheme it is affixed to stem is “H” vowel then “H” vowel is deleted.

Example:

| | |
|---|---|
| Surface form: | masam |
| Intermediate form: | masa00m |
| Lexical form: | masa+Hm |

- The Exclusion Rule a:b /<= LC __RC

The "/<=" operator means the correspondence never occurs in the environment. This operator forbids the specified correspondence from occurring in the specified context. This operator explains "exceptions". Lexical "a" cannot correspond to surface "c" preceding "d:e". As the operator symbol suggests, the "/<=" operator is similar to the "<=" operator in that it does not prohibit the correspondence from occurring in other environments.

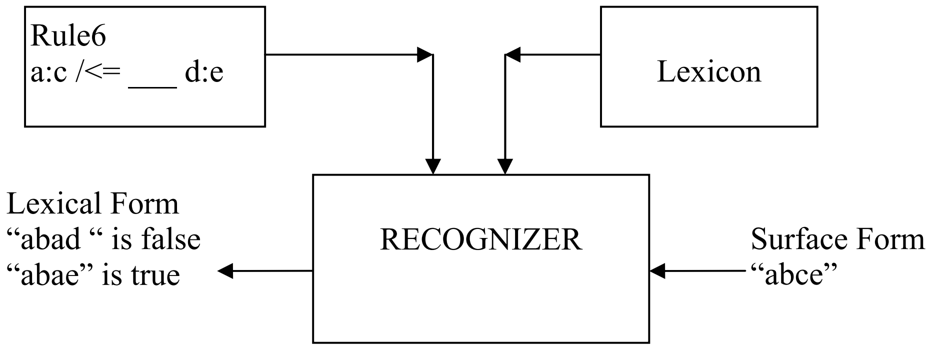


**Figure 2.6 Example of exclusion rule**

Example rule "g:ğ /<= n_" (Oflazer, 1993)

If foreign words ending with "g" and "g" is preceded by another consonant then it doesn't become "ğ". This consonant may be "n".

Example:

| | |
|---|---|
| Surface form: | brifingim |
| Intermediate form: | brigfing0im |
| Lexical form: | brifing+Hm |

The diagnostic properties of the four rule types is shown in the Table 2.1 (Antworth, 1995, pp. 5)

**Table 2.1 Diagnostic properties of the four rule types**

| Rules | Is t:c allowed preceding i ? | Is preceding I the only environment in which t:c is allowed ? | Must t always correspond to c before i ? |
|---|---|---|---|
| t:c => __i | yes | yes | no |
| t:c <= __i | yes | no | yes |
| t:c ⇔ __i | yes | yes | yes |
| t:c /<= __i | no | - | - |

### 2.2.5.1 Complex Environments

Complex environments contain optional elements, repeated elements and alternative elements. These are elements:

1. " ‘ " Symbol:

" ’ " is a stress mark. "As and example we will use a vowel reduction rule, which states that a vowel followed by some number of consonants followed by stress (indicated by ‘) is reduced to schwa (e). "(Antworth, 1995, pp.6)

For example: (Antworth, 1995, pp. 5)

| | | |
|---|---|---|
| LR: | bab’a | bamb’a |
| SR: | beb’a | bemb’a |

- " ( " and " ) " Symbols:

Parenthesis indicates an optional element.

Rule a : c => __d(d)’

This rule requires either one or two "d" characters.

Rule a : c => (d)(d)’

This rule requires either zero, one or two “d” characters.

- “*” Symbol:

An asterisk indicates zero or more instances of an element.

Rule a : c → __c*’
This rule requires either zero, one or more “c” characters.

Rule a:c → __ cc*’

This rule requires either one or more “c” characters.

- “|” Symbol:

Vertical bar indicates disjunctive between expressions.

- “[“ and “]” Symbols:

The square brackets delimit the disjunctive expressions from the rest of the environment.

| | |
|---|---|
| Rule1 | a:e => __C’ |
| Rule2 | a:e => ‘__ |

These two rules use the “=>” operator. This operator allows the correspondence to occur only in the specified environment. “a:e” occurs only in a pretonic syllable in Rule1 and in a tonic syllable in Rule2. So the two rules conflict with each other. This type of rule conflict is called an environment conflict. If we collapse there two rules into one then this conflict can be resolved like this:

Rule3 a:e => [ __ C’ | ‘ __ ]

This rule means the “a:e” correspondence is permitted only in either pretonic or tonic position.

### 2.2.5.2 Rules Component

#### 2.2.5.2.1 Alphabetic Characters

Alphabet characters are used in lexical and surface forms. Alphabet characters include all characters and special symbols. The NULL and BOUNDARY symbols are also considered as alphabetic characters. Alphabet doesn’t include ANY symbol and subset names.

There are special symbols to write rules like ANY, NULL, BOUNDARY symbol. These special symbols explained below.

- “@” is an ANY symbol, not ANY character. ANY symbol is said to be a "wildcard" character. ANY symbols indicate for any alphabetic character in feasible pairs.

  Example:

  |  |  |
  |---|---|
  | Feasible Pairs: | {a:a, b:b, c:c, d:d, d:e, e:e} |
  | Rule: | a:b => __d:@ |

  For this rule, “a” corresponds to “b” before any feasible pair whose lexical character is “d”. “d:@” means d:d and d:e. Because “@” means for lexical character “d” is “d” and “e”. “@:i” is simplified to “.:i”. And also “i:@” is simplified to “i:.”.

- The ANY symbol can also be used on the lexical side of a correspondence or on the surface side of a correspondence or both of them. These usage alternatives are “a:@”, “@:a”, “@:@”.  “@:@” means all feasible pairs.

- “0” (zero) is a NULL symbol. NULL symbol written as zero. There must be an equal number of characters in both lexical and surface forms. Each lexical character must map to exactly one surface character, and each surface character must map to exactly one lexical character.

  If necessary, analyzer inserts morpheme boundary character with NULL symbol.

| | |
|---|---|
| Lexical Representation: | b ı ç a k + ı |
| Surface Representation: | b ı ç a ğ 0 ı |

  Recognizer function implemented in this project doesn’t show 0’s (zero) on output form and lexical form. Here recognizer inserts 0 (zero) in surface form to symbolize “+” as morpheme boundary.

  Recognizer can delete or insert characters with NULL symbol. We can do almost anything with zero. The correspondence “H:0” represents the deletion of “H”, while “0:H” represents the insertion of x.

| | |
|---|---|
| Lexical Representation: | m a s a + H m |
| Surface Representation: | m a s a 0 0 m |

  “Without zero, two-level phonology would be limited to the most trivial phonological processes; with zero, the two-level model has the expressive power to handle complex phonological or morphological phenomena. “ (Antworth, 1995, pp. 6)

- “+” is a morpheme boundary. This symbol is used only in a lexical form. Morpheme boundary corresponds to a surface “0” (zero).

- “#” is used as word boundary symbol. “#” indicates a word boundary, either initial or final. It can only correspond to another boundary (like “#:#”).

| | |
|---|---|
| Lexical Representation: | d o l a p + ı # |
| Surface Representation: | d o l a b 0 ı # |

Recognizer doesn’t show “#” symbol on input and output form.

#### 2.2.5.2.2 Feasible Pairs

A feasible pair is a specific correspondence between a lexical alphabetic character and a surface alphabetic character. The set of all correspondence is called the set of feasible pairs. Each feasible pair must be declared in rules environment.

- Default Correspondence

Some of correspondences are called default correspondences which lexical and surface side are identical like “a:a”, “b:b” or “c:c”. But “a:b” is not a default correspondence because “a” and “b” are not identical. Normally default correspondences are not included in each rule. Generally default correspondence can be written in one state table. A table of default correspondences has only one state and each transition is back to state one. Default corresponds must include "@:@" as a column header.

- Special Correspondence

If a correspondence is not default then it is called special correspondence. Generally special correspondence can be written in separate tables. Subsets can be used in special correspondence. "@:@" indicate special correspondences like "a:c" not “a:a”.

#### 2.2.5.2.3 Subsets

A subset name defines a set of alphabet character. These set of characters indicate the character classes. These character classes are defined in SUBSET statements in the rules file.

Example: V is a set of vowels: V = {a, e, ı, i, o, ö, u, ü}
C is a set of consonants: C = {b, c, d, f, g, ğ, h, k …z}

| | | |
|---|---|---|
| SUBSET | S1 | a e |
| SUBSET | S2 | c y z |
| SUBSET | S3 | d ı |

Rule a:c => __d:d

This rule can be written as "Rule S1:S2 => __S3:S3" or "Rule S1:S2 => __ S3". So this means that using the correspondence "S1:S2" as a column header in a rule does not implicitly declare as feasible pairs all correspondences that match.

### 2.2.6 Implementing Two-Level Rules as Finite State Machines

How two-level rules work, how they can be implemented as finite state machines and how the four types of two-level rules can be translated into finite state tables are presented in this subsection.

#### 2.2.6.1 How Two-Level Rules Work

A two-level description contains rules. These rules must also contain a set of default correspondences, such as "a:a", "b:b", and so on. The sum of the special and default correspondences is called feasible pairs. The total set of valid correspondences or feasible pairs that can be used in the description.

The recognizer implemented in this thesis requires an input in surface form and it outputs lexical form of given word. Now, let us see how two-level rules work in an example:

| | |
|---|---|
| Rule1 | a:c => __ d |
| Surface form | abcd |
| Feasible Pairs | {a:a, b:b, c:c, d:d, a:c } |

Recognizer begins with the first character of surface form. Firstly it looks in feasible pairs for "a:c". If this correspondence is not exists in feasible pairs then recognizer skips this correspondence. If this correspondence is exists in feasible pairs then the recognizer analyze it.

Step 1: Recognizer finds "a" as surface character in feasible pair. There is only one correspondence "a:a". So "a" is not converted and to any other character. Then recognizer moves on to the second character of the input word.  (LR: Lexical Representation, SR: Surface Representation)

```
SR:     a     b     c     d
              |
Rule:         |
              |
LR:     a
```

Step 2: Recognizer analyzes "b" as surface character with same operation like step 1.

```
SR:     a     b     c     d
              |     |
Rule:         |     |
              |     |
LR:     a     b
```

Step 3: Recognizer analyzes “c” as surface character with same operation like step 1. But in this case there is a different situation. Because there are two alternatives for “c” as surface character in feasible pairs. Alternatives are “a:c” and “c:c”. Recognizer selects one alternative and moves the next character. When recognizer reaches the final character it decides this alternative correct or not. Sometimes recognizer reaches the next character to decide if these alternatives are true or false. If it is false recognizer goes back and tries the second alternative for character “c”.

For first alternative:

```
SR:     a     b     c     d
              |     |     |
Rule:         |     |     1
              |     |     |
LR:     a     b     a
```

Recognizer moves the second character for “d”. There is only one pair for “d” as surface character in feasible pair that is “d:d”. Thus, the first alternative “a:c“ is true because Rule1 means that lexical “a” is realized as surface “c” only (but not always) in the environment preceding “d:d”.  This satisfies the environment of the Rule1 and exits Rule1. Since there are no more characters in the lexical form, the recognizer outputs the lexical form “abad”. However the recognizer is not done yet. It will continue backtracking and try to apply other alternatives.

```
SR:     a     b     c     d
              |     |     |     |
Rule:         |     |     1     |
              |     |     |     |
LR:     a     b     a     d
```

Recognizer also applies the second alternative “c:c”.

```
SR:     a     b     c     d
              |     |     |
Rule:         |     |     |
                 |       |       |
LR:      a       b       c
```

And finally recognizer reaches the final character of surface form. Recognizer finds the “d” character in feasible pairs as surface character again. There is only one alternative in feasible pair that is “d:d”. So recognizer applies this alternative. Since there are no more characters in the lexical form, the recognizer outputs the lexical form “abcd”. Now recognizer is done.

```
SR:      a       b       c       d
                 |       |       |       |
Rule:            |       |       |       |
                 |       |       |       |
LR:      a       b       c       d
```

The procedure is essentially the same when two-level rules are used in generation mode. In this situation lexical form is input and the corresponding surface forms are output.

### 2.2.6.2 How Finite State Machines Work

“The basically mechanical procedure for applying two-level rules makes it possible to implement the two-level model on a computer by using a formal language device called a finite state machine.” (Antworth, 1995, pp.11). Finite State Automaton (FSA) is the simplest finite state machine. It generates the well-formed strings of a regular language. Regular language is a type of formal language.

“A Finite State Transducer (FST) is like an FSA except that it simultaneously operates on two input strings. It recognizes whether the two strings are valid correspondences of each other.” (Antworth, 1995, pp.12).

Two level rules can be implemented as FST, the only difference being that the column headers are pairs of symbols, such as “a:a” and “b:b” or “b:c”. State-transition tables are occurred after compiling rules. An automaton is represented with state-transition table. The state-transition table indicates the start state, final and non-final states and transitions between each state.

Here, we show an example:

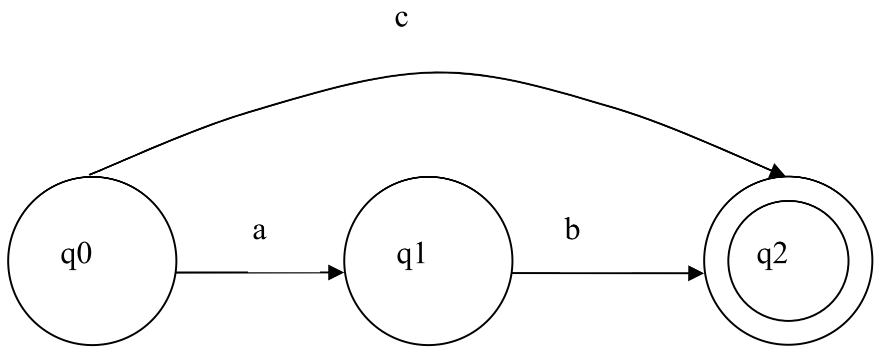


**Figure 2.7 State diagram of an example automaton - I**

**Table 2.2 State transition table of an example automaton - I**

| | Input | | |
|---|---|---|---|
| State | a | b | c |
| 0 . | 1 | 0 | 2 |
| 1 . | 0 | 2 | 0 |
| 2 : | 0 | 0 | 0 |

The graph of automaton is represented in Figure 2.7 as Table 2.2. State 0 is initial state. State 2 is a final state and marked with ":" of symbol. The "." indicates non-final state and ":" indicates final states. "0" indicates an illegal or missing transition. We can read the first row as "if we are in state 0 and we see the input 'a' then we must go to state 1 or if we see the input 'b' then we must go to state 0 or if we see the input 'c' we must go to state 2".

An FSA operates only on a single input string and a finite state transducer (FST) operates on two input strings simultaneously. For example, assume the first input string to an FST is from language L1 above, and the second input string is from language L2. Here is an example correspondence of two strings:

L1: abbb

L2: accb

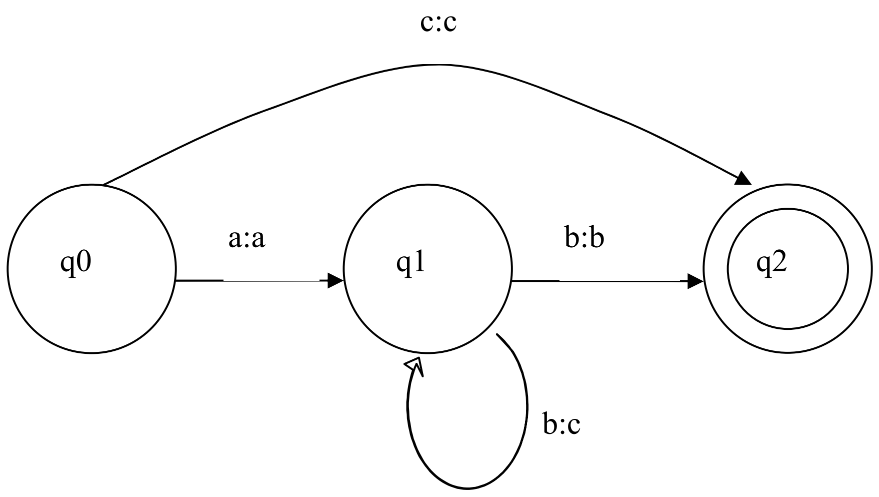


**Figure 2.8 State diagram of an example automaton - II**

Figure 2.8 shows the FST in diagram form. Note that the only difference from an FSA is that the arcs are labeled with a correspondence pairs consisting of a symbol from each of the input languages.

This FST can also be represented as tables like Table 2.3.

**Table 2.3 State transition table of an example automaton - II**

| | Input | | | |
|---|---|---|---|---|
| State | a<br>a | b<br>b | b<br>c | c<br>c |
| 0 . | 1 | 0 | 0 | 2 |
| 1 . | 0 | 2 | 1 | 0 |
| 2 : | 0 | 0 | 0 | 0 |

The upper or lexical language specifies the string "abbb" and the lower or surface language specifies the string "accb". However, note that a two-level rule does not specify the grammar of a full language.

I will explain each rule type as a finite state machine in detail.

#### 2.2.6.2.1 Rule Types as a Finite State Machine

- A “=>” Rule as a Finite State Machine

If rule is “a:c => __d” then we can draw this state diagram to represent this rule.

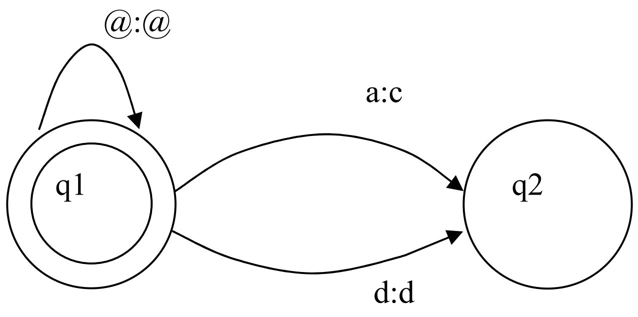


**Figure 2.9 State diagram for rule “a:c => __d”**

The column header “@:@” does not match for all feasible pairs. The “@:@” arc (where @ is the ANY symbol) allows any pairs in feasible pairs to pass successfully through the FST except “a:c” and “d:d”. Every feasible pair must belong to one and only one column header. This FST can also be represented as state transition table like Table 2.4.

**Table 2.4 State transition table for rule “a:c => __d”**

| | Input | | |
|---|---|---|---|
| State | a<br>c | d<br>d | @<br>@ |
| 1 : | 2 | 1 | 1 |
| 2 . | 0 | 1 | 0 |

Default correspondences of the system must be existed in a FST.

**Table 2.5 State transition table of default correspondences for rule “a:c => __d”**

| | Input | | | | |
|---|---|---|---|---|---|
| State | a | b | c | d | @ |
| | a | b | c | d | @ |
| 1 : | 1 | 1 | 1 | 1 | 1 |

Default corresponds must include "@:@" as a column header. "@:@" indicate special correspondences like "a:c". If the correspondence "@:@" is not exist then the FST would fail for special correspondence like "a:c". Because all the rules apply in parallel in a two-level description. The correspondence “a:c” is exists in Table 2.4 but this correspondence is not exists in Table 2.5. But the correspondence "@:@" is occur in Table 2.5 so this doesn’t fail.

State tables specify where correspondences must fail. Table 2.4 and Table 2.5 will work together to generate the lexical form of given surface form. Table 2.4 fails when anything but "d:d" follows "a:c".

- A “<=” Rule as a Finite State Machine

If rule is “a:c <= __d” then we can draw this state diagram to represent this rule.

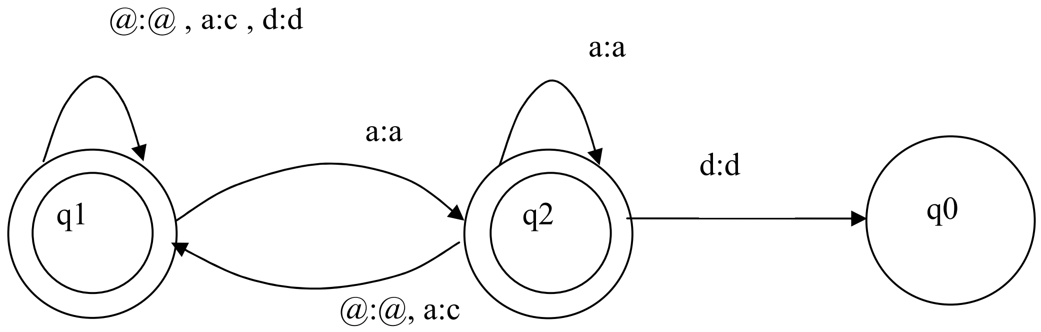


**Figure 2.10 State diagram for rule “a:c <= __d”**

This FST can also be represented as state transition table like Table 2.6.

Table 2.6 State transition table of default correspondences for rule “a:c <= __d”

| | Input | | | |
|---|---|---|---|---|
| State | a | a | d | @ |
| | c | a | d | @ |
| 1 : | 1 | 2 | 1 | 1 |
| 2 : | 1 | 2 | 0 | 1 |

In this state transition table we can see that the zero in the “d:d” column indicates that the input has failed. State 1 and state 2 are final states. State zero is a non-final state.

- A “<=>” Rule as a Finite State Machine

If rule is “a:c <=> __d” then we can draw this state diagram to represent this rule.

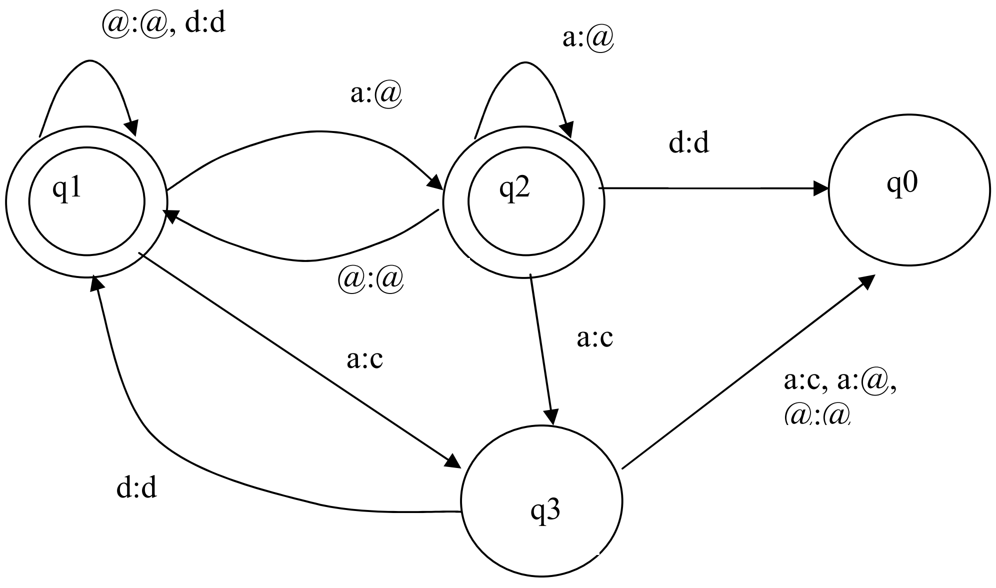


**Figure 2.11 State diagram for rule “a:c <=> __d”**

This FST can also be represented as state transition table like Table 2.7.

**Table 2.7 State transition table of default correspondences for rule "a:c <=> __d"**

| | Input | | | |
|---|---|---|---|---|
| State | a<br>c | a<br>@ | d<br>d | @<br>@ |
| 1 : | 3 | 2 | 1 | 1 |
| 2 : | 3 | 2 | 0 | 1 |
| 3 . | 0 | 0 | 1 | 0 |

This state transition table is a combination of the "=>" and "<=" tables.

- A "/<=" Rule as a Finite State Machine

If rule is "a:c /<= __d:b" then we can draw this state diagram to represent following rule.

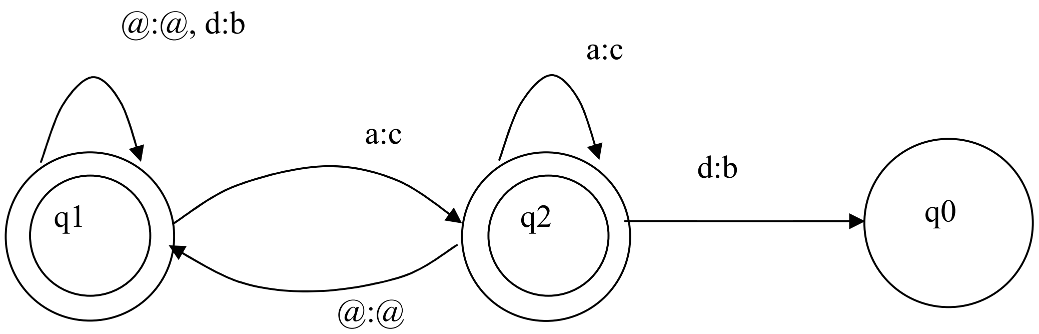


**Figure 2.12 State diagram for rule "a:c /<= __d:b"**

This rule type shares properties of the "<=" type rule. This FST can also be represented as state transition table like Table 2.8.

**Table 2.8 State transition table of default correspondences for rule "a:c /<= __d:b"**

| | Input | | |
|---|---|---|---|
| State | a<br>c | d<br>b | @<br>@ |
| 1 : | 2 | 1 | 1 |
| 2 : | 2 | 0 | 1 |

#### 2.2.6.2.2 Regular Expressions and Automata

A regular expression is a string that describes a whole set of strings according to certain syntax rules. A string is a sequence of symbols or it is any sequence of alphanumeric characters. Alphanumeric characters include letters, numbers, tabs, spaces and punctuation. Regular expression is a formula for matching strings that follow some pattern. Many text editors and utilities to search text in information retrieval applications, word-processing applications and etc use these expressions. Regular expressions are supported by class libraries such as scripting tools such as awk, grep, sed, and increasingly in interactive development environments such as Microsoft's Visual C++. It is used also in UNIX and UNIX-like utilities.

Regular expressions are made up of normal characters and metacharacters. Normal characters include upper and lower case letters and digits. The metacharacters have special meanings and are described in detail below.

- Regular expressions are case sensitive so lowercase /a/ is different from uppercase /A/. The string /exam/ will not match /Exam/ according to this rule. Square brackets can be used to solve this problem. The pattern /eE/ matches patterns containing e or E.

- Dash "-" is used in square brackets to specifies any one character in a range. The pattern /[1-3]/ means one of the characters 1,2 or 3.

- Caret ^ is used to match the start of the line.

- Question mark /?/ is used to preceding character or nothing.

- Kleene * is used to match zero or more occurrences of the immediately previous character or regular expression.

- Kleene + is used to specify one or more of the previous character.

- Period character "." is used to match any single character.
- $ is used to match the end of a line.

#### 2.2.6.2.3 Finite State Automaton

A finite state machine (FSM) or finite state automaton (FSA) is an abstract machine that has only a finite, consonant amount of memory. Finite state automata can be represented using a state diagram or state transition table. An FSA is composed of states and directed transition arcs. At least there must be existed an initial state, a final state and an arc between them. Each state has transitions to states. There is a input string that determines which transition is followed. Finite state machines are studied in automata theory. An automaton is a self-operating machine.

There two kinds of automata, one of them is deterministic and the other is non-deterministic. In non-deterministic finite state automaton, each state there might several possible choices for the next state as in Figure 2.14. So there can be more than one transition from a given state for a given possible input. In deterministic automaton, for each state there is at most one transition for each possible input as shown Figure 2.13.

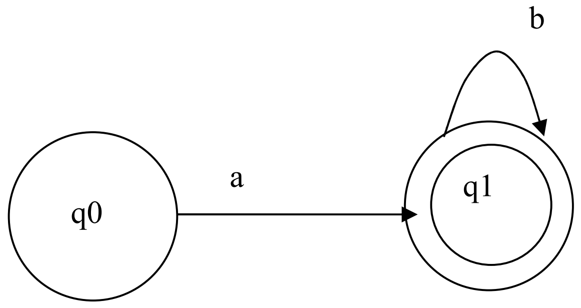


**Figure 2.13 State diagram for a deterministic finite state automaton**

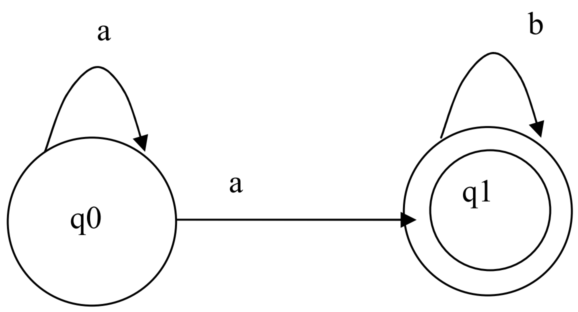


**Figure 2.14 State diagram for a non-deterministic finite state automaton**

If current state is q0 and input is “a” character then there are two choices for next state for Figure 2.14. One choice is q0 and the other one is q1. But if current state is q0 and the input is “a” character then there is only one choice for Figure 2.13. This is the difference between deterministic and non-deterministic finite state automaton.

### 2.2.6.2.4 State Transition Table

A state transition table is used to describe the transition function. It is used to represent an automaton. States are indicated horizontally, and events are read vertically. A state transition table represents the start state and the accepting states. State transitions and actions are represented in the form of action/new-state.

Final states are represented by “:” symbol and non-final states are represented by “.” symbol. Final states mean accepted states.

**Table 2.9 State transition table for deterministic finite state automaton that as shown Figure 2.13.**

| State | Input | Next State |
|---|---|---|
| q0. | a | q1 |
| q1: | b | q1 |

**Table 2.10 State transition table for non-deterministic finite state automaton that as shown Figure 2.14.**

| State | Input | Next State |
|---|---|---|
| q0. | a | {q0, q1} |
| q1: | b | q1 |

All the possible inputs to the machine are enumerated across the columns of the table. All the possible states are enumerated across the rows.

NFA (Non-Deterministic Finite State Automaton) is a non-deterministic then a new input may cause the machine to be in more than one state. In this case, parentheses {} are used with the list of all legal states in the parentheses like Table 2.13.

If you want, it is possible to draw a state diagram from the state transition table. We can use these steps to do it. Firstly draw the circles to represent the states given then for each state, draw an arrow from the source states to the destination states. Finally determine start state and accept states.

### 2.2.6.2.5 Formal Languages

The origin of regular expressions lies in automata theory and formal language theory. These theories are part of theoretical computer science. These fields study models of computation and ways to describe and classify formal languages. An automaton implicitly defines a formal language. A formal language is nothing but a set of strings. Each string composed of symbols from an alphabet. A formal language is a set of finite length words over some finite alphabet. This description is used in mathematics, logic and computer science.

A formal language can be specified in variety of ways such as:

1. Some formal grammar produce strings, (Chomsky hierarchy)
2. Regular expression produce strings,

3. Some automaton accepted strings (like Turing machine or finite state automaton)

From a set of related YES / NO questions those ones for which the answer is YES, (decision problem)

### 2.2.6.2.6 Regular Languages and FSA's

Regular language is a type of formal language. Regular languages can be characterized as languages defined by regular expressions. If the set of all languages that are regular, then the class of languages called regular languages. A language is regular if it is accepted by some DFA (Deterministic Finite State Automaton), NFA (Non-Deterministic Finite State Automaton), regular expression or regular grammar.

A single language is a set of strings over a finite alphabet and is there for countable. A regular language may have an infinite number of strings. The strings of a regular language can be enumerated, written down for length 0, length 1, length 2 and so forth.

Regular language is the language associated to a regular grammar. A grammar G=( N,T, P, σ ) in which every production is of the form:

$A \rightarrow a$ or $A \rightarrow aB$ or $A \rightarrow \lambda$, where $A, B \in N$, $a \in T$.

Regular languages over an alphabet T have the following properties:

(λ = 'empty string', αβ = 'concatenation of α and β', α^n = 'α concatenated with itself n times'):

$\emptyset$, $\{ \lambda \}$, and $\{ a \}$ are regular languages for all $a \in T$.

If L1 and L2 are regular languages over T the following languages also are regular:

L1 U L2 = { α | α Є L1 or α Є L2 },

L1L2 = { αβ | α Є L1, β Є L2},

L1^* = { α1 … αn | αk Є L1, n Є N },

T^* - L1 = { α Є T^* | α Є L1 },

L1 п L2 = { α | α Є L1 and α Є L2 }.

Regular languages coincide with the languages accepted by non-deterministic finite-state automata. Every non-deterministic finite state automaton is equivalent to some deterministic finite state automaton. A language L is regular if and only if there exists a finite-state automaton that accepts precisely the strings in L.

Regular languages are closed under operations: concatenation, union, intersection, complementation, difference, reversal, Kleene star, substitution, homomorphism and any finite combination of these operations.

# CHAPTER THREE
# TURKISH MORPHOLOGY

## 3.1 Turkish Language

Turkish is an agglutinative language like Finnish, Hungarian. It belongs to the southwestern group of Turkic family. Turkic languages are in the Uralic-Altaic language family. In agglutinative languages, words formed by combined root words and morphemes. Word structures can grow by addition of morphemes. Morphemes added to a stem can convert the word from nominal to a verbal structure or vice-versa.

Turkish has a very productive morphology. There is a root and several suffixes are combined to this root. It is possible to produce a very high number of words from the same root with suffixes. The lexicon size may grow to unmanageable size.

A popular example of a Turkish word formation is:

OSMANLILAŞTIRAMAYABİLECEKLERİMİZDENMİŞSİNİZCESİNE

This can be broken down into morphemes:

OSMAN+LI+LAŞ+TIR+AMA+YABİL+ECEK+LER+İMİZ+DEN+MİŞ+SİNİZ +CESİNE

In this example, one word in Turkish corresponds to a full sentence in English. This example can be translated into English as “as if you were of those whom we might consider not converting into an Ottoman”. In English, words contain only a small number of affixes or none at all.

There are 29 letters in Turkish language. The eight of them are vowels and twenty-one of them are consonants.

Vowel letters: {a, e, ı, i, o, ö, u, ü}
Consonant letters: {b, c, ç, d, f, g, ğ, h, j, k, l, m, n, p, r, s, ş, t, v, y, z}

The number of vowels is more than many languages. Vowels of Turkish can be classified in three groups according to their articulatory properties:

- Front and back,
- Round and unrounded,
- High or low

We can partition the vowels as below in detail:

- Back vowels: {a, ı, o, u}
- Front vowels: {e, i, ö, ü}
- Front unrounded vowels: {e, i}
- Front rounded vowels: {ö, ü}
- Back unrounded vowels: {a, ı}
- Back rounded vowels: {o, u}
- High vowels: {ı, i u, ü}
- Low unrounded vowels: {a, e}

### 3.1.1 Morphophonemic’s

Turkish word formation uses a number of phonetic harmony rules. When a suffix is appended to a stem vowels and consonants change in certain ways.

### 2.1.1.1 Vowel Harmony

Vowel harmony is the best-known morphophonemic process in Turkish. It is most interesting and distinctive feature. Vowel harmony is a left-to-right process. It operates sequentially from syllable to syllable. Vowel harmony processes force certain vowels in suffixes agree with the last vowel in the stems or roots they are being affixed to. When vowels are affixed to a stem, they change according to the vowel harmony rules. The first vowel in the suffix changes according to the last vowel of the stem. Vowel harmony consists of two assimilations:

- Palatal assimilation (It is called in Turkish as “Büyük Ünlü Uyumu”)

This is called “major vowel harmony” . This vowel harmony is common to almost Turkic languages. This assimilation is about front/back feature of the language. Back vowels are the set of {a, ı, o, u} and the front vowels are the set of {e, i, ö, ü}.

If the vowels of the following morphemes are back then the vowel of the first morpheme in a word is back.

For example:

| | |
|---|---|
| Surface Form: | askılar |
| Intermediate Form: | askı0lar |
| Lexical Form: | askı+lAr |

“lAr” is a plural suffix. “A” is resolved as “a” or “e” in general. But in this example “A” is resolved as “a” because the vowels of the stem are back vowels.

If the vowels of the following morphemes are front then the vowel of the first morpheme in a word is front.

For example:

| | |
|---|---|
| Surface Form: | evler |

| | |
|---|---|
| Intermediate Form: | ev0ler |
| Lexical Form: | ev+lAr |

In this example “A” is resolved as “e” because the vowel of the stem is front vowel.

If the last vowel is a long vowel then “A” is realized as an “e”. Long vowels are “â, û, ô”. These vowels are in words of French origin in general.

For example:

| | |
|---|---|
| Surface Form: | saatler |
| Intermediate Form: | saat 0ler |
| Lexical Form: | saât+lAr |

| | |
|---|---|
| Surface Form: | goller |
| Intermediate Form: | gol0ler |
| Lexical Form: | gôl+lAr |

| | |
|---|---|
| Surface Form: | usuller |
| Intermediate Form: | usul0ler |
| Lexical Form: | usûl+lAr |

- Labial assimilation (It is called in Turkish “Küçük Ünlü Uyumu”)

  This is called “minor vowel harmony”. This assimilation is about rounded/unrounded feature of the language. There are four alternatives about this assimilation:

  - “H” is resolved as “ı,i,u,ü” in general .“H” is resolved as “ü” in this example because the last vowel in the stem is a front-unrounded vowel.

    For example:

    | | |
    |---|---|
    | Surface Form: | çölün |

| | |
|---|---|
| Intermediate Form: | çöl0ün |
| Lexical Form: | çöl+Hn |

- “H” is resolved as “ü” if the last vowel in the stem is a long “û” or “ô” as defined below.

For example:

| | |
|---|---|
| Surface Form: | golün |
| Intermediate Form: | gol0ün |
| Lexical Form: | gôl+Hn |

For example:

| | |
|---|---|
| Surface Form: | usulün |
| Intermediate Form: | usul0ün |
| Lexical Form: | usûl+Hn |

- “H” is resolved as “ı” if the last vowel of the stem is a back-unrounded vowel.

For example:

| | |
|---|---|
| Surface Form: | topalın |
| Intermediate Form: | topal0ın |
| Lexical Form: | topal+Hn |

- “H” is resolved as “i” if the last vowel in the stem is a front-unrounded vowel.

For example:

| | |
|---|---|
| Surface Form: | defterim |
| Intermediate Form: | defter0im |
| Lexical Form: | defter+Hm |

“H” is resolved as “i” if the last vowel in the stem is a long “â” also.
For example:

| | |
|---|---|
| Surface Form: | saatim |
| Intermediate Form: | saat0im |
| Lexical Form: | saât+Hm |

There are some two-level rules for vowel harmony. These rules are:

- A:a => [VOWEL:BACKV | Q:0] ':' * CONS * @:0 *  +:0 *  _
- A:e => [VOWEL:FRONTV | E:0 | %:a | &:u | ^:o] ':' * CONS * @:0 * +:0 * _
- H:ı => [VOWEL:BKUNRV | Q:0] ':' * CONS * +:0 * @:0 * _
- H:i => [VOWEL:FRUNRV | E:0 | FRUNRV:0 +:0 |  %:a] ':' * CONS * +:0 * @:0 * _
- H:u => VOWEL:BKROV ':' * CONS * +:0 * @:0 * _
- H:ü =>[VOWEL:FRROV   |  &:u | ^:o] ':' * CONS * +:0 * @:0 * _

These rules will be explained in chapter four.

**3.1.1.2 Consonant Harmony**

Consonant harmony is another basic aspect of Turkish phonology. Consonants of Turkish phonology can be classified into two main groups. These are voiceless and voiced. Voiceless consonants are {“ç”, ”f”, ”h”, ”k”, ”p”, ”s”, ”ş”, ”t”}. Voiced consonants are {“b”, ”c”, ”d”, ”g”, ”ğ”, ”j”, ”l”, ”m”, ”n”, ”r”, ”v”, ”y”, ”z”}. Consonant harmony rules doesn’t formulate easily because of irregular character of borrowed and native words. There are some consonant harmony rules in Turkish:

- If the end of the word is one the voiceless consonants (“p”, ”ç”, ”t”, ”k”) then it changes to a corresponding voiced consonants (“b”, ”c”, ”d”, ”ğ”).
    - “p” changes to “b”.

      For example:

      | | |
      |---|---|
      | Surface Form: | kitabım |
      | Intermediate Form: | kitab0ım |

| | |
|---|---|
| Lexical Form: | kitap+Hm |

There are some exceptions to this rule like “soyad”, “hemoroid”, “önad”, etc.

For example: “d” doesn’t change in this example.

| | |
|---|---|
| Surface Form: | soyadın |
| Intermediate Form: | soyad0ın |
| Lexical Form: | soyad+Hn |

- “d” changes to “t”.

  For example:

| | |
|---|---|
| Surface Form: | tattık |
| Intermediate Form: | tat0tık |
| Lexical Form: | tad+DHk |

- “k” changes to “ğ”.

  For example:

| | |
|---|---|
| Surface Form: | ayağın |
| Intermediate Form: | ayağ00ın |
| Lexical Form: | ayak+nHn |

- “ç” changes to “c”.

  For example:

| | |
|---|---|
| Surface Form: | ağacın |
| Intermediate Form: | ağac0ın |
| Lexical Form: | ağaç+Hn |

There are some exceptions to this rule like “göç”, “aç”, ”iç”, etc.

For example: “ç” doesn’t change in this example.

| | |
|---|---|
| Surface Form: | açım |
| Intermediate Form: | aç0ım |
| Lexical Form: | aç+Hm |

- Let “D” indicate a suffix initial dental consonant that may resolve as either a “d or t”. “D” is resolved to a “t” is the last phoneme in the stem is resolved as one of {“ç”, ”f”, ”h”, ”k”, ”p”, ”s”, ”ş”, ”t”}. In other cases “D” is resolved as a “d”.

For Examples:

| | |
|---|---|
| Surface Form: | yulaftan |
| Intermediate Form: | yulaf0tan |
| Lexical Form: | yulaf+Dan |

| | |
|---|---|
| Surface Form: | masadan |
| Intermediate Form: | masa0dan |
| Lexical Form: | masa+Dan |

- If the last consonant of the stem is one of {“ç”, ”f”, “h”, “k”, “p”, “s”, “ş”} and if the suffix begins with the “c” then “c” is resolved as a “ç”.

For Example:

| | |
|---|---|
| Surface Form: | yaşça |
| Intermediate Form: | yaş0ça |
| Lexical Form: | yaş+cA |

There are some exceptions for this rule. These exceptions are “aç”, “iç”, “haç”, etc.

- If “k” is at the end of the stem and “k” preceded by an “n” then “k” becomes a “g”.

For Example:

| | |
|---|---|
| Surface Form: | çelenge |

| | |
|---|---|
| Intermediate Form: | çeleng00e |
| Lexical Form: | çelenk0yA |

There are some exceptions for this rule also. One of the exception word is "bank".

- If the final character of the stem is "g" and a vowel is beginning of the suffix then "g" becomes a "ğ" in foreign origin words.

For Example:

| | |
|---|---|
| Surface Form: | analoğa |
| Intermediate Form: | analoğ0a |
| Lexical Form: | analog+yA |

There are some exceptions for this rule. Some exceptions are "lig", "pedagog", etc.

- If the final character of the stem is "g" and a consonant is beginning of the suffix then "g" does not become a "ğ".

For Example:

| | |
|---|---|
| Surface Form: | bumerangım |
| Intermediate Form: | bumerang0ım |
| Lexical Form: | bumerang+Hm |

- There are very number of nominal words in Turkish. Some of these nominal words ending with "su". These words don't obey the standard inflection rules. When a suffix is starting with a vowel or a vowel-dropping consonant is affixed then a stem final "y" is inserted to stem.

For Examples:

| | |
|---|---|
| Surface Form: | akarsuyunuz |
| Intermediate Form: | akarsuy00unuz |
| Lexical Form: | akarsuY+yHnHz |

| | |
|---|---|
| Surface form: | akarsular |
| Intermediate form: | akarsu00lar |
| Lexical form: | akarsuY+lar |

- When certain suffixes are affixed last consonant is duplicated in Arabic or Persian origin words.

For Examples:

| | |
|---|---|
| Surface Form: | zammı |
| Intermediate Form: | zamm00ı |
| Lexical Form: | zam0+yH |

| | |
|---|---|
| Surface Form: | zamlar |
| Intermediate Form: | zam0lar |
| Lexical Form: | zam+lAr |

- "-sH" can be affixed to Arabic origin words. If these words ending with a vowel then drops in exception to the general rule.

For Example:

| | |
|---|---|
| Surface Form: | camii |
| Surface Form: | camisi |
| Intermediate Form: | cami0i |
| Lexical Form: | cami+sH |

There are many numbers of words that have this property. Example words are "mevki", "cami", "terfi", "zayi", "ikna", "merci", etc.

#### 3.1.1.3 Root Deformations

Turkish roots are not flexible in normally. There are some cases about various deformations. There are three exception cases:

- Root is observed in personal pronouns

For examples:

| | |
|---|---|
| Surface Form: | bana |
| Intermediate Form: | ban00a |
| Lexical Form: | ben+yA |

| | |
|---|---|
| Surface Form: | sana |
| Intermediate Form: | san00a |
| Lexical Form: | sen+yA |

If “ben” and “sen” roots take the plural suffix then their structures completely change like this:

| | |
|---|---|
| ben + lAr → biz | (not benler) |
| sen + lAr → siz | (not senler) |

- Wide vowel at the end of the stem is narrowed when the suffix “Hyor” comes after the verbs ending with the “A”.

For example:

| | |
|---|---|
| Surface Form: | kapiyor |
| Intermediate Form: | kapi0Hyor |
| Lexical Form: | kapa+Hyor |

- When a suffix is beginning with a vowel comes after some nouns, which has a vowel {I} in its last syllable, this vowel drops. This occurs generally designating parts of the human body.

For example:

| | |
|---|---|
| Surface Form: | ağzımız |
| Intermediate Form: | ağ0ız0ımız |
| Lexical Form: | ağ$ız+HmHz |

- Similar with the above rule, when the possessive suffix "Hl" is affixed to some verbs, and the last vowel of the verb is vowel "I" then this vowel drops.

  ayır + ıl → ayrıl (not ayırıl)

- If a plural suffix is affixed to a compound words then this suffix coming before the possessive suffix at the end of the stem.

  gözyaşı + lAr -> gözyaşları (not gözyaşılar)

## 3.2 Turkish Morphology

Turkish has very productive morphology. The lexicon size may grow to unmanageable size. Because of this the number of words is very high. Turkish is characterized by certain Morphophonemics, Morphotactics and syntactic features.

### 3.2.1. Morphotactics

There are two main classes for Turkish roots. These classes are nominal and verbal. These classes are important for suffixes. If a suffix can be affixed to a nominal root then this suffix cannot be affixed to a verbal root with the same semantic function. But there are some suffixes can be affixed to nouns or verbs. These paradigms will be explained in following subsection in detail. Two rules are designed for Morphotactics for Turkish in this project: *nominal order rule*, *verbal order rule*. State diagrams of these rules are shown in Figure 3.2 and Figure 3.4.

#### 3.2.1.1 Nominal Paradigm

Nominal paradigm applies to nouns and adjectives. It describes the order of the inflectional suffixes. It is shown in Figure 3.1.

| | | | | |
|---|---|---|---|---|
| Nominal Root | Plural Suffix | Possessive Suffix | Case Suffix | Relative Suffix |

**Figure 3.1 Turkish Nominal Model**

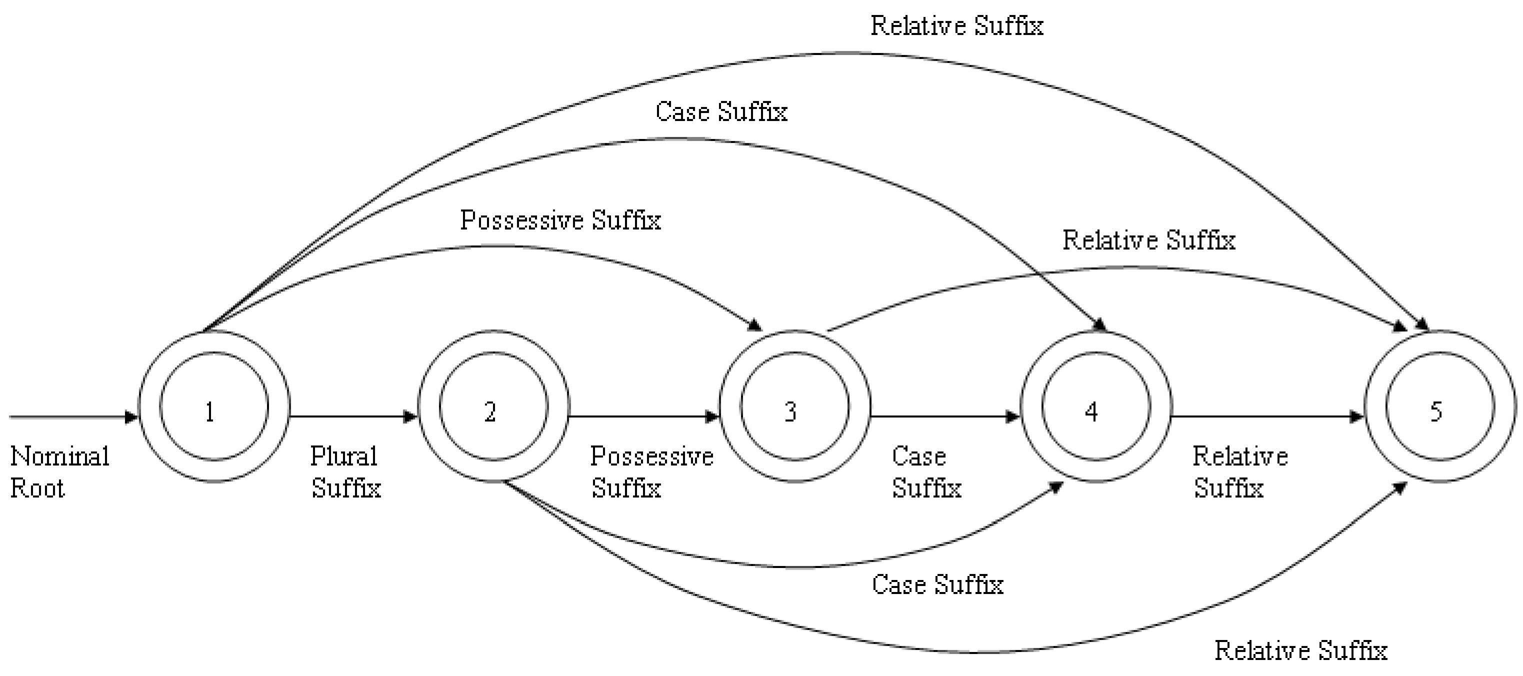


**Figure 3.2 State diagram for nominal model**

**Table 3.1 Nominal paradigm's element**

| | |
|---|---|
| Plural Suffix | -lAr |
| Possessive Suffix | -Hm, -HmHz, -Hn, - HnHz, -sH, -lArH |
| Case Suffix | -yH, -ylA, -yA, -DA, -DAn, -nHn, -nH, - nA, -nDA, -nDAn |
| Relative Suffix | -ki |

- Nominal root is the base. It may be an adjective or noun. All the elements of noun paradigm are optional, except the nominal root.

- The plural suffix can be added directly to the nominal root.

Example: kedi + lAr → kediler

- If the possessed noun is plural then plural suffix come before possessive suffix.

Example: kedi + lArH → kedileri

When the third person plural possessive suffix "-lArH" comes after a plural noun, one of the "-lArH" drops.

Example: kedi + lAr + lArH → kedileri (not kedilerleri)

Some words have third person singular possessive suffix. If possessive suffixes come after these words, possessive suffix is removed. Because possessive suffix is already occur in word structure.

Example: safrakesesi + sH → safrakesesi (not safrakesesisi)

- Case suffixes come after possessive suffix.

Example: masa + DA → masada

- The relative suffix “-ki” may be added to two type of case suffixes: One them is “-nHn” is called genitive case suffixes. The other one is “-DA” is called locative case suffixes.

  Example: kapı + nHn + ki → kapınınki

### 3.2.1.2 Verbal Paradigm

The verbal paradigm is more complicated than nominal paradigm. It is shown in Figure 3.3. Verbal paradigm’s elements are shown in Table 3.2.

| Verbal Root | Voice Suffixes | Negation Suffix | Compound Verb Suffix | Main Tense Suffix | Question Suffix | Second Tense Suffix | Person Suffix |
|---|---|---|---|---|---|---|---|

**Figure 3.3 Turkish Verbal Model**

**Table 3.2 Verbal paradigm’s elements**

| | | |
|---|---|---|
| Voice Suffixes | Reflexive | -(H)n |
| | Reciprocal | -(H)ş |
| | Causative | -DHr, -Ht, -t, -Hr, -Ar |
| | Passive | - Hl, -Hn, -n |
| Negation Suffix | -mA, -(y)AmA | |
| Compound Verb Suffix | -(y)Adur, -(y)Ayaz, -(y)Abil, - (y)Akal, -(y)Akoy, -(y)Agel, -(y)Agör, -(y)Aver | |
| Main Tense Suffix | -DH, -sA, -mHş, -(y)A, -(y)AcAk, -mAlH, -(H)r, -0, -Ar, -(H)yor, -mAktA | |
| Question Suffix | -mH | |
| Second Tense Suffix | -(y)DH, -(y)sA, -(y)mHş | |
| Person Suffix | -m, -n, -k, -nHz, -lAr, -(y)Hm, -sHn, -(y)Hz, -sHnHz, -lHm, -(y)Hn, -(y)HnHz, -sHlAr | |

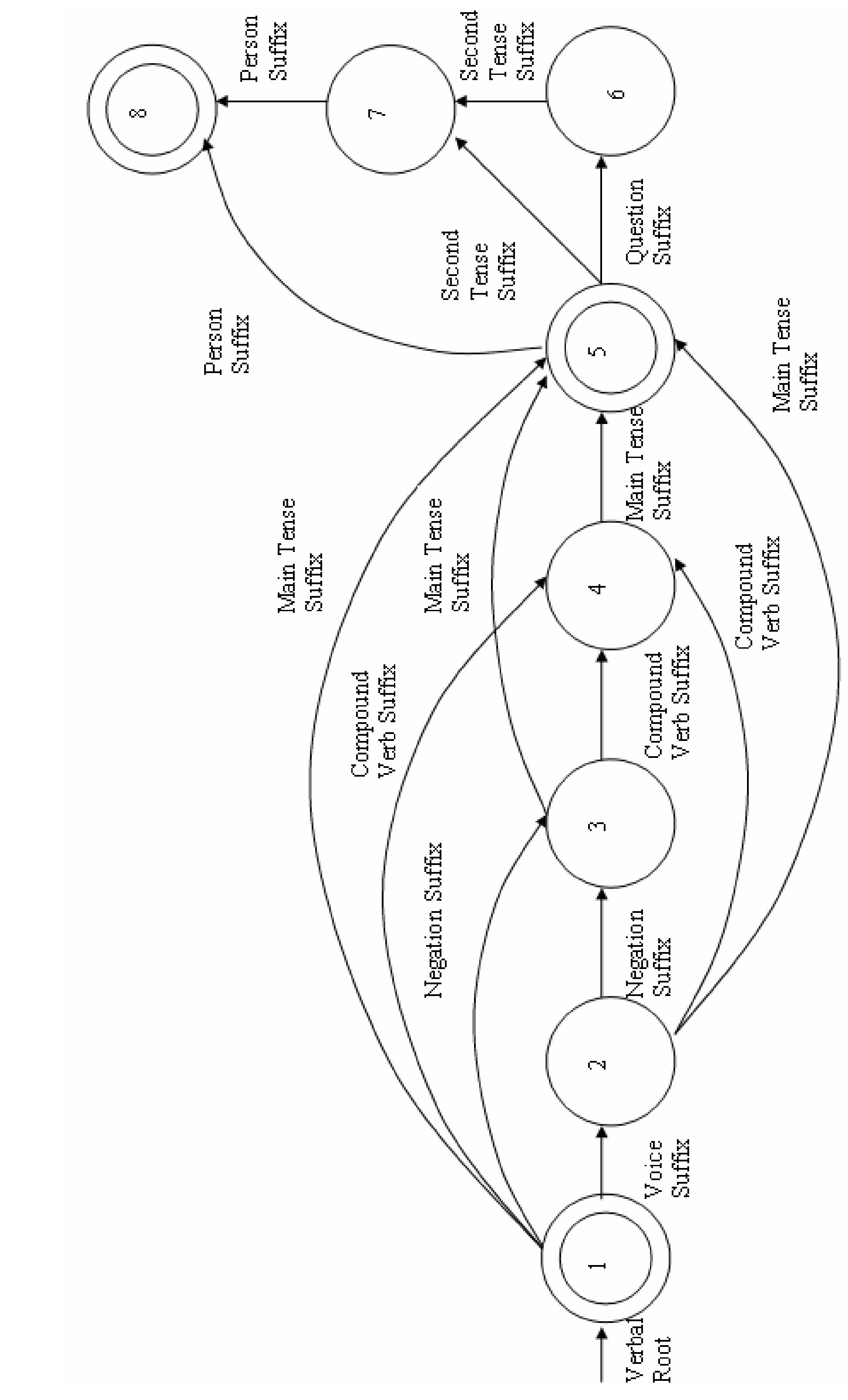


**Figure 3.4 State diagram for verbal model**

- Verbal root, the main tense suffix and the person suffix are obligatory. Others are optional.

- Voice suffixes can be divided four groups: reflexive, reciprocal, causative and passive. These suffixes can be combined. They must be appearing in the indicated order, and the reflexive and reciprocal are mutually exclusive.

  Example: “öp-mek”, “öp-üş-mek”, “öp-üş-tür-mek”, “öp-üş-tü-rül-mek”

  The causative verb suffixes can be used repeatedly. For example, “ört”, “ör-ttür”, “ört-türt”.

  The passive and reflexive forms of some verbs’ meaning are different but they have same structure. These two sentences have the same verb but their meaning is different from each other. Example word is “yıka-n-dı”.

| | |
|---|---|
| “Balkon yıkandı.” | The balkony were washed. |
| “Hülya yıkandı.” | Hülya washed herself. |

- There are two negation suffixes. These are “-mA” and “-(y)AmA”. These meaning is different from each other. The meaning of “-mA” is “not” and the meaning of “-(y)AmA” is “can/may not”.

  Example:

| | |
|---|---|
| “ye*me*m”: | I don’t eat |
| “yi*yeme*m”: | I can’t eat |

- Compound verb suffixes is used to add verbs certain additional semantics. “-(y)Agör” and “-(y)Aver” are the most frequently used. Example word is “oku-yabil-ir”.

- Main tense suffix is an obligatory. Every verb has a main tense suffix. There are nine tenses:

- Definite past: {"-DH"}
- Narrative past: {"-mHş"}
- Future: {"-(y)AcAk"}
- Aorist: {"-(H)r"," -Ar"}
- Progressive: {"-(H)yor"," -mAktA"}
- Conditional: {"-sA"}
- Optative: {"-(y)A"}
- Necessitative: {"-mAIH"}
- Imperative: {"-0"}

- There is only one question suffix that is "-mH". This suffix is written separate from the word it follows. Example word is "yaptın mı?"

- Second tense suffixes are affixed to verb stems ending with a vowel with the insertion of a Y in between. Example word is "oku-sa-ydı".

- Person suffix defines the first, second and third singular or plural persons. Example word is "gel-di-k"

#### 3.2.1.3 Verbal Nouns

There are two types of sentences for Turkish. One of them is verb sentences and the other one is noun sentences. There is a verb in the verb sentence. A verb represents an action in the sentence. There is no explicit verb in the noun sentence so there is no action. The verb "to be" in English is correspondence to the noun sentences in Turkish.

Example:

"Su içiyorum" is a verb sentence. "içtim" is a verb.
"Akıllıyım" is a noun sentence. There is no verb.

Negation suffix can be added to the verb: e.g.: “Su iç-mi-yorum”. There is no such a suffix in noun sentences. For negation process the word “değil” can be used: e.g.: “Akıllı değilim.”

Question suffix can be used noun and verb sentences: e.g.: “Su içtim mi?” is a verb sentence; “Akıllı mıyım?” is a noun sentence.

Tense information can be used in noun and verb sentences. But verb sentences use nine tense but noun sentences use three tense. These three tense are definite past, narrative past and conditional suffix. Noun sentences also use independent words: “idi”, “imiş”, “ise”. Example is “Akılllı-ydı-m”.

### 3.2.1.4 Suffix Classification

There is a root and several suffixes are combined to this root in Turkish. It is possible to produce a very high number of words from the same root with suffixes. These suffixes help us to find the right stem of a word. When we analyze the word “kalem” then we can not sure stem, because there are two solutions: “kalem” and “kale-m”. When we analyze the word “kalemler” there is only one solution: “kalem-ler”. The solution “kale-m-ler” is invalid.

There are two main classes of suffixes:

- Derivational suffixes,
- Inflectional suffixes,
    - Nominal verb suffixes,
    - Noun suffixes,
    - Tense & person verb suffixes,
    - Verb suffixes,

Derivational suffixes produce a new word. This new word has different meaning than the word derivational suffixes are affixed. They change the class of the word. For example they make nouns from verbs. Inflectional suffixes do not produce a new word like derivational suffixes. They do not change the class of the word.

Turkish suffixes are shown in Table 3.3. The abbreviations used to show suffixes in a generic way in this representation in below: (Eryiğit & Adalı, 2004)

U: {ı,i,u,ü}
C: {c,ç}
A: {a,e}
D: {d,t}
I: {ı,i}
(): The letter in parentheses can be omitted.

**Table 3.3 Turkish suffixes**

| | |
|---|---|
| Derivational suffixes | "-lUk, "-CU", "-CUk", "-lAş", "-lA", "-lAn", "-CA", "-lU", "-sUz" |
| Nominal Verb Suffixes | "-(y)Um", "-sUn", "-(y)Uz", "-lAr", "-m", "-n", "-k", "-nUz", "-DUr", "-cAsInA", "-(y)DU", "-(y)sA", "(y)mUş", "-(y)ken" |
| Noun Suffixes | "-lAr", "-(U)m", "-(U)mUz", "-(U)n", "-(U)nUz", "-(s)U", "-lArI", "-(y)U", "-nU", "(n)Un", "-(y)A", "-nA", "-DA", "-nDA", "-DAn", "-nDAn", "-(y)lA", "ki", "-(n)cA" |
| Tense & Person Verb Suffixes | "-(y)Um", "-sUn", "-(y)Uz", "-sUnUz", "-lAr", "-mUş", "-(y)AcAk", "-(U)r", "-Ar", "-(U)yor", "-mAktA", "-mAlI", "-m", "-n", "-k", "-nUz", "-DU", "-sA", "-lIm", "-(y)A", "-(y)UnUz", "-(y)Un", "-sUnlAr", "-DUr", "-(y)DU", "-(y)sA", "-(y)mUş", "-cAsInA", "-(y)ken |
| Verb Suffixes | "-m", "-zsIn", "-z", "-yIz", "-zsInIz", "-zlAr", "-mA", "-(y)AmA", "-(y)Adur", "-(y)Uver", "-(y)Agel", "-(y)Agör", "-(y)Abil", "-(y)Ayaz", "-(y)Akal", "-(y)Akoy", "-mAk", "-(y)UcU", "-(y)Up", "(y)AlI", "-DUkçA", "-(y)ArAk", "-(y)UncA", "-(y)Dan", "-yA", "-(y)An", "-(y)AcAk", "-(y)AsI", "-DUk", "-mUş", "-mAzlIk", "-mA", "-(y)Uş", "-Dan", "-DA", "-(y)lA", "-(y)A", "-mAksIzIn", "-mAdAn", "-(U)n", "-(U)ş", "-(U)l", "-Dur", "-(U)t" |

# CHAPTER FOUR
# TURKISH RULE DEFINITIONS

## 4.1 Rule Definitions for Turkish Language

This chapter describes the details of the definitions of Turkish rules. These Turkish rule definitions have been taken from Oflazer's project. (Oflazer, 1993)

### 4.1.1 Alphabetic Characters

Turkish alphabetic characters are: {b, c, ç, d, f, g, ğ, h, j, k, l, m, n, p, r, s, ş, t, v, y, z ,Z ,a ,e ,ı ,i, o, ö, u, ü, A, H, K, J, B, 9, D, $, Y, %, &, ^, ', x, q, w, E, Q, ~, +, 1, 2, 3, 4, 5, 6, 7, 8,}

| | |
|---|---|
| Null: | "0" |
| Any: | "@" |
| Boundary: | "#" |

Some capital letters are used in this representation. Their meanings are following: Z = {s}, A = {a,e}, H = {ı,i,u,ü}, J = {ç}, B = {b}, D = {d,t}, Y = {y}, % = {a}, & = {u}, ^ = {o}, . K represents a root-final lexical "k". It never becomes a surface "ğ". When certain suffixes are affixed then some vowels are deleted in the roots. These vowels are prefixed with a "$" in the lexical form. The apostrophe "'" is used to separate proper nouns from suffixes.

### 4.1.2 Feasible Pairs

Default correspondence rules are defined as feasible pairs and presented in section 4.2.1.

### 4.1.3 Subsets

A set of alphabet characters indicate the character classes. These character classes are defined in subset statements.

**CONS** is the set of consonants: {b, c, ç, d, f, g, ğ, h, j, k, l, m, n, p, r, s, ş, t, v, y, z, D, Z, Y, K, J, B, 9}

**VOWEL** is a set of lexical vowels. These are used in lexical level: {a, e, i, ı, o, ö, u, ü, A, H, %, &, ^, E, Q}

**SVOWEL** a set of surface vowels: {ı, i, o, ö, u, ü, a, e}

**BACKV** is a set of back vowels: {a, ı, u, o}

**FRONTV** is a set of front vowels: {e, i, ö, ü}

**HIGHV**: {ı, i, u, ü}

**FRUNRV** is set of front unrounded vowels: {i, e}

**FRROV** is set of front rounded vowels: {ö, ü}

**BKROV** is set of back rounded vowels: {u, o}

**BKUNRV** is set of back unrounded vowels: {a, ı}

**X** is set of some consonants. These lexical consonants used as first letter in a suffix but they may disappear on the surface form under certain conditions: {s, y, n}

**NDCONS** is a set of some consonants. These lexical consonants used as first letter in a suffix but they are always realized on the surface: {c, Z, l, d, D}

## 4.2 Two-Level Rules for Turkish

We can divide two-level rules for Turkish into two groups: *default correspondences* and *special correspondences*. There are three rules as default correspondences, twenty three rules as special correspondences. These correspondences are presented following subsections.

### 4.2.1 Default Correspondences for Turkish Language

There are 3 default correspondences for Turkish language. Feasible pairs are all these default correspondences' pair.

**RULE** "defaults"

**Table 4.1 State transition table for default correspondences - I**

| | b | c | ç | d | F | g | ğ | h | j | k | l | m | n | p | r | s | ş | t | v | y | z | Z | a | e | ı | o | ö | u | ü | @ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | b | c | ç | d | f | g | ğ | h | j | k | l | m | n | p | r | s | ş | t | v | y | z | s | a | e | ı | o | ö | u | ü | @ |
| 1: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**RULE** "defaults"

**Table 4.2 State transition table for default correspondences - II**

| | A | A | H | H | H | H | K | J | B | 9 | n | D | D | b | d | k | c | s | y | $ | a | e | ı | i | o | ö | u | ü | Y | Y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | e | ı | ı | u | ü | k | ç | b | g | 0 | t | d | p | p | t | ğ | ç | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | y |
| 1: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**RULE** "defaults"

**Table 4.3 State transition table for default correspondences -III**

| | A | % | & | ^ | ‘ | ‘ | x | q | w | E | Q | ~ | + | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | @ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | a | u | o | 0 | ‘ | x | q | w | 0 | 0 | ~ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | @ |
| 1: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

### 4.2.2 Two-Level Rules for Turkish Language

There are 23 rules for Turkish language and listed below. These rules are called as special correspondence. They have been taken from Kemal Oflazer’s project. It is called “Two-Level Description of Turkish Morphology”.

**RULE 1**: A:a => [VOWEL:BACKV | Q:0] ':' * CONS * @:0 * +:0 * _

This rule force the agreement of an A vowel to a preceding vowel in the back ness attribute.

**Table 4.4 State transition table for Rule 1**

| | A | Q | ‘ | CONS | @ | + | VOWEL | @ |
|---|---|---|---|---|---|---|---|---|
| | a | 0 | ‘ | CONS | 0 | 0 | BACKV | @ |
| 1: | 0 | 2 | 1 | 1 | 1 | 1 | 3 | 1 |
| 2: | 3 | 2 | 2 | 2 | 2 | 2 | 3 | 1 |
| 3: | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 |

For example:

| | |
|---|---|
| Surface Form: | masalar |
| Intermediate Form: | masa0lar |
| Lexical Form: | masa+lAr |

**RULE 2**: A:e => [VOWEL:FRONTV | E:0 | %:a | &:u | ^:o] ':' * CONS * @:0 * +:0 * _

This rule force the agreement of an A vowel to a preceding vowel in the back ness attribute also.

**Table 4.5 State transition table for Rule 2**

| | A | ^ | ‘ | CONS | @ | + | & | % | E | VOWEL | @ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | e | o | ‘ | CONS | 0 | 0 | u | a | 0 | FRONTV | @ |
| 1: | 0 | 2 | 1 | 1 | 1 | 1 | 3 | 4 | 5 | 6 | 1 |
| 2: | 6 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 2 | 6 | 1 |
| 3: | 6 | 2 | 3 | 3 | 3 | 3 | 3 | 4 | 3 | 6 | 1 |
| 4: | 6 | 2 | 4 | 4 | 4 | 4 | 3 | 4 | 4 | 6 | 1 |
| 5: | 6 | 2 | 5 | 5 | 5 | 5 | 3 | 4 | 5 | 6 | 1 |
| 6: | 6 | 2 | 6 | 6 | 6 | 6 | 3 | 4 | 6 | 6 | 1 |

For example:

| | |
|---|---|
| Surface Form: | meşeler |
| Intermediate Form: | meşe0ler |
| Lexical Form: | meşe+lAr |

**RULE 3**: A:0 <=> _ +:0 H:@ y o r

This rule forces the agreement of an H vowel to a preceding one in back ness and roundedness.

**Table 4.6 State transition table for Rule 3**

| | A | A | + | H | y | o | R | @ |
|---|---|---|---|---|---|---|---|---|
| | 0 | @ | 0 | @ | y | o | r | @ |
| 1: | 2 | 7 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2. | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 |
| 3. | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 |
| 4. | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 |
| 5. | 0 | 0 | 0 | 0 | 0 | 6 | 0 | 0 |
| 6. | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7: | 2 | 7 | 8 | 1 | 1 | 1 | 1 | 1 |
| 8: | 2 | 7 | 1 | 9 | 1 | 1 | 1 | 1 |
| 9: | 2 | 7 | 1 | 1 | 10 | 1 | 1 | 1 |
| 10: | 2 | 7 | 1 | 1 | 1 | 11 | 1 | 1 |
| 11: | 2 | 7 | 1 | 1 | 1 | 1 | 0 | 1 |

Example:

| | |
|---|---|
| Surface form: | selamlıyor |
| Intermediate form: | selam0l00ıyor |
| Lexical form: | selam+lA+Hyor |

**RULE 4**: H:u => VOWEL:BKROV ':' * CONS * +:0 * @:0 * _

This rule forces the agreement of an H vowel to a preceding one in back ness and roundedness also.

**Table 4.7 State transition table for Rule 4**

| | H | VOWEL | ‘ | CONS | + | @ | @ |
|---|---|---|---|---|---|---|---|
| | u | BKROV | ‘ | CONS | 0 | 0 | @ |
| 1: | 0 | 2 | 1 | 1 | 1 | 1 | 1 |
| 2: | 2 | 2 | 2 | 2 | 2 | 2 | 1 |

For example:

| | |
|---|---|
| Surface Form: | kolun |
| Intermediate Form: | kol0un |
| Lexical Form: | kol+Hn |

**RULE 5**: H:ü =>[VOWEL:FRROV | &:u | ^:o] ':' * CONS * +:0 * @:0 * _

This rule also forces the agreement of an H vowel to a preceding one in back ness and roundedness.

**Table 4.8 State transition table for Rule 5**

| | H<br>ü | ^<br>o | ‘<br>‘ | CONS<br>CONS | +<br>0 | @<br>0 | &<br>u | VOWEL<br>FRROV | @<br>@ |
|---|---|---|---|---|---|---|---|---|---|
| 1: | 0 | 2 | 1 | 1 | 1 | 1 | 3 | 4 | 1 |
| 2: | 4 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 1 |
| 3: | 4 | 2 | 3 | 3 | 3 | 3 | 3 | 4 | 1 |
| 4: | 4 | 2 | 4 | 4 | 4 | 4 | 3 | 4 | 1 |

For example:

| | |
|---|---|
| Surface Form: | gölün |
| Intermediate Form: | göl0ün |
| Lexical Form: | göl+Hn |

**RULE 6**: H:ı => [VOWEL:BKUNRV | Q:0] ':' * CONS * +:0 * @:0 * _

This rule also forces the agreement of an H vowel to a preceding one in back ness and roundedness.

**Table 4.9 State transition table for Rule 6**

| | H<br>I | @<br>0 | ‘<br>‘ | CONS<br>CONS | +<br>0 | @<br>0 | VOWEL<br>BKUNRV | @<br>@ |
|---|---|---|---|---|---|---|---|---|
| 1: | 0 | 2 | 1 | 1 | 1 | 1 | 3 | 1 |
| 2: | 3 | 2 | 2 | 2 | 2 | 2 | 3 | 1 |
| 3: | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 |

For example:

| | |
|---|---|
| Surface Form: | kumarın |
| Intermediate Form: | kumar0ın |
| Lexical Form: | kumar+Hn |

**RULE 7**: H:i => [VOWEL:FRUNRV | E:0 | FRUNRV:0 +:0 | %:a] ':' * CONS * +:0 * @:0 * _

This rule also forces the agreement of an H vowel to a preceding one in back ness and roundedness.

**Table 4.10 State transition table for Rule 7**

| | H | % | ‘ | CONS | + | @ | FRUNRV | E | VOWEL | @ |
|---|---|---|---|---|---|---|---|---|---|---|
| | i | a | ‘ | CONS | 0 | 0 | 0 | 0 | FRUNRV | @ |
| 1: | 0 | 2 | 1 | 1 | 1 | 1 | 3 | 5 | 6 | 1 |
| 2: | 6 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 6 | 1 |
| 3: | 0 | 2 | 1 | 1 | 4 | 1 | 3 | 5 | 6 | 1 |
| 4: | 6 | 2 | 4 | 4 | 4 | 4 | 4 | 4 | 6 | 1 |
| 5: | 6 | 2 | 5 | 5 | 5 | 5 | 5 | 5 | 6 | 1 |
| 6: | 6 | 2 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 1 |

For example:

| | |
|---|---|
| Surface Form: | kalemim |
| Intermediate Form: | kalem0im |
| Lexical Form: | kalem+Hm |

**RULE 8**: H:0 <=> VOWEL:VOWEL (':') +:0 _

If the last character of the stem is a vowel and the first character of the morpheme it is affixed to stem is “H” vowel then “H” vowel is deleted.

**Table 4.11 State transition table for Rule 8**

| | H | H | VOWEL | ‘ | + | @ |
|---|---|---|---|---|---|---|
| | 0 | @ | VOWEL | ‘ | 0 | @ |
| 1: | 0 | 2 | 2 | 1 | 1 | 1 |
| 2: | 0 | 2 | 2 | 3 | 4 | 1 |
| 3: | 0 | 2 | 2 | 1 | 4 | 1 |
| 4: | 1 | 0 | 2 | 1 | 1 | 1 |

Example:

| | |
|---|---|
| Surface form: | kasam |
| Intermediate form: | kasa00m |
| Lexical form: | kasa+Hm |

**RULE 9**: SVOWEL:0 <=> $:0_ CONS +:0 (X:0) [A:@ | H:@] | _ +:0 H:@ y o r

If there is a vowel ellipsis phenomenon in the lexical form of the stem, it has to be deleted on the surface form. For this situation "$" is used by Oflazer on Turkish Morphology project so same symbol is used for this case. In the dictionary, some words include this "$" symbol.

**Table 4.12 State transition table for Rule 9**

| | SVOWEL | SVOWEL | $ | CONS | + | H | A | X | y | o | r | @ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | @ | 0 | CONS | 0 | @ | @ | 0 | y | o | r | @ |
| 1: | 11 | 16 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 16 | 1 | 1 |
| 2: | 7 | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 1 | 1 |
| 3: | 11 | 16 | 2 | 4 | 17 | 1 | 1 | 1 | 4 | 16 | 4 | 1 |
| 4: | 11 | 16 | 2 | 1 | 5 | 1 | 1 | 1 | 1 | 16 | 1 | 1 |
| 5: | 11 | 16 | 2 | 1 | 1 | 0 | 0 | 6 | 1 | 16 | 1 | 1 |
| 6: | 11 | 16 | 2 | 1 | 1 | 0 | 0 | 1 | 1 | 16 | 1 | 1 |
| 7. | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 8 | 0 | 8 | 0 |
| 8. | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9. | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 10 | 0 | 0 | 0 | 0 |
| 10. | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 11. | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12. | 0 | 0 | 0 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 14 | 0 | 0 | 0 |
| 14. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 0 |
| 15. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 16: | 11 | 16 | 2 | 1 | 17 | 1 | 1 | 1 | 1 | 16 | 1 | 1 |
| 17: | 11 | 16 | 2 | 1 | 1 | 18 | 1 | 1 | 1 | 16 | 1 | 1 |
| 18: | 11 | 16 | 2 | 1 | 1 | 1 | 1 | 1 | 19 | 16 | 1 | 1 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 19: | 11 | 16 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 20 | 1 | 1 |
| 20: | 11 | 16 | 2 | 1 | 17 | 1 | 1 | 1 | 1 | 16 | 0 | 1 |

Example:

| | |
|---|---|
| Surface form: | ağzımı |
| Intermediate form: | ağ00z0ım00ı |
| Lexical form: | ağ$ız+Hm+yH |

**RULE 10**: H:0 /<= VOWEL:0 +:0 _ y o r

The suffix +Hyor is indicating the present continuous tense. This suffix is verbal suffix. So "H" vowel is not deleted bur the last phoneme of the stem is vowel then this vowel is deleted.

**Table 4.13 State transition table for Rule 10**

| | VOWEL | + | H | y | o | r | @ |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | y | o | r | @ |
| 1: | 2 | 1 | 2 | 1 | 1 | 1 | 1 |
| 2: | 2 | 3 | 2 | 1 | 1 | 1 | 1 |
| 3: | 2 | 1 | 4 | 1 | 1 | 1 | 1 |
| 4: | 2 | 3 | 2 | 5 | 1 | 1 | 1 |
| 5: | 2 | 1 | 2 | 1 | 6 | 1 | 1 |
| 6: | 2 | 1 | 2 | 1 | 1 | 0 | 1 |

Example:

| | |
|---|---|
| Surface form: | döşüyor |
| Intermediate form: | döş00üyor |
| Lexical form: | döşe+Hyor |

**RULE 11**: X:0 <=> CONS (':') +:0 _ (CONS) VOWEL

If a last phoneme of the stem is a consonant and the first phoneme of the first suffix is "s", "y" or "n" then this phoneme is deleted.

**Table 4.14 State transition table for Rule 11**

| | X<br>0 | X<br>@ | CONS<br>CONS | ‘<br>‘ | +<br>0 | VOWEL<br>VOWEL | @<br>@ |
|---|---|---|---|---|---|---|---|
| 1: | 0 | 2 | 2 | 1 | 1 | 1 | 1 |
| 2: | 0 | 2 | 2 | 3 | 4 | 1 | 1 |
| 3: | 0 | 2 | 2 | 1 | 4 | 1 | 1 |
| 4: | 7 | 5 | 2 | 1 | 1 | 1 | 1 |
| 5: | 0 | 6 | 6 | 3 | 4 | 0 | 1 |
| 6: | 0 | 2 | 2 | 3 | 4 | 0 | 1 |
| 7. | 0 | 8 | 8 | 0 | 0 | 1 | 0 |
| 8. | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Examples:

| | |
|---|---|
| Surface form: | bademi |
| Intermediate form: | badem00i |
| Lexical form: | badem+yH |
| | |
| Surface form: | balonu |
| Intermediate form: | balon00u |
| Lexical form: | balon+yH |
| | |
| Surface form: | kalemi |
| Intermediate form: | kalem00i |
| Lexical form: | kalem+sH |

**RULE 12**: D:t <=> [h | @:ç | ş | @:k | @:p | @:t | f | s]  +:0 (@:0) _

“D” is known as a “d” by default. But in the following case “D” is known as “t”. Whenever it is preceded by one of the consonants in the option list across a morpheme boundary, “D” is known as “t”.

**Table 4.15 State transition table for Rule 12**

| | D<br>t | D<br>@ | s<br>s | +<br>0 | f<br>f | @<br>t | @<br>p | @<br>k | @<br>ş | ş<br>ç | @<br>h | h<br>@ | @<br>@ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1: | 0 | 1 | 2 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 2: | 0 | 1 | 2 | 3 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 3: | 2 | 0 | 2 | 4 | 4 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 4: | 2 | 0 | 2 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 5: | 2 | 1 | 2 | 1 | 6 | 1 | 6 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6: | 2 | 1 | 2 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

Example:

| | |
|---|---|
| Surface form: | dolapta |
| Intermediate form: | dolap0ta |
| Lexical form: | dolab+DA |

**RULE 13**: {b, d}:{p, t} <=> _# | _ +:0 (X:0) [CONS | c:ç]

When a word end with one of "p,t" or "p,t" are followed by a morpheme beginning with a consonant then voiced obstruents "b,d" as "p,t".

**Table 4.16 State transition table for Rule 13**

| | b<br>p | d<br>t | b<br>@ | d<br>@ | #<br># | +<br>0 | X<br>0 | CONS<br>CONS | c<br>ç | @<br>@ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1: | 2 | 2 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2. | 0 | 0 | 0 | 0 | 1 | 4 | 0 | 0 | 0 | 0 |
| 3: | 2 | 2 | 3 | 3 | 0 | 6 | 1 | 1 | 1 | 1 |
| 4. | 0 | 0 | 3 | 3 | 0 | 0 | 5 | 1 | 1 | 0 |
| 5. | 0 | 0 | 3 | 3 | 0 | 0 | 0 | 1 | 1 | 0 |
| 6: | 2 | 2 | 0 | 0 | 1 | 1 | 7 | 0 | 0 | 1 |
| 7: | 2 | 2 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |

Example:

| | |
|---|---|
| Surface form: | dolapçı |

| | |
|---|---|
| Intermediate form: | dolap0çH |
| Lexical form: | dolab+cH |

**RULE 14**: c:ç <=> [ç | ş | @:k | @:p | @:t | f | s | h ] +:0 _ [H:@ |A:@]

**Table 4.17 State transition table for Rule 14**

| | c<br>ç | c<br>@ | ç<br>ç | +<br>0 | H<br>0 | A<br>@ | ş<br>ş | @<br>k | @<br>p | @<br>t | f<br>f | s<br>s | h<br>h | @<br>@ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1: | 0 | 1 | 2 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 2: | 0 | 1 | 2 | 3 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 3: | 5 | 4 | 2 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 4: | 0 | 1 | 2 | 1 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 5. | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**RULE 15**: ç:c <=> _ +:0 (X:0) VOWEL

When "c" is the first phoneme of the suffix and "c" is the last phoneme of the stem then both "c"'s change to "ç" by mutual influence.

**Table 4.18 State transition table for Rule 15**

| | c<br>ç | ç<br>@ | +<br>0 | X<br>0 | VOWEL<br>VOWEL | @<br>@ |
|---|---|---|---|---|---|---|
| 1: | 2 | 5 | 1 | 1 | 1 | 1 |
| 2. | 0 | 0 | 3 | 0 | 0 | 0 |
| 3. | 0 | 0 | 0 | 4 | 1 | 0 |
| 4. | 0 | 0 | 0 | 0 | 1 | 0 |
| 5: | 2 | 5 | 6 | 1 | 1 | 1 |
| 6: | 2 | 5 | 6 | 1 | 1 | 1 |

Example:

| | |
|---|---|
| Surface form: | haraççı |
| Intermediate form: | haraç0çı |

| | |
|---|---|
| Lexical form: | haraç+cH |

**RULE 16**: k:ğ <=> VOWEL _ +:0 (X:0) VOWEL

When the first phoneme is morpheme is a vowel and the last phoneme is the stem is consonant "k" then "k" becomes "ğ".

**Table 4.19 State transition table for Rule 16**

| | k | k | VOWEL | + | X | @ |
|---|---|---|---|---|---|---|
| | ğ | @ | VOWEL | 0 | 0 | @ |
| 1: | 0 | 2 | 2 | 1 | 1 | 1 |
| 2: | 6 | 3 | 2 | 1 | 1 | 1 |
| 3: | 0 | 1 | 2 | 4 | 1 | 1 |
| 4: | 0 | 1 | 0 | 1 | 5 | 1 |
| 5: | 0 | 1 | 0 | 1 | 1 | 1 |
| 6. | 0 | 0 | 0 | 7 | 0 | 0 |
| 7. | 0 | 0 | 2 | 0 | 8 | 0 |
| 8. | 0 | 0 | 2 | 0 | 0 | 0 |

Examples:

| | |
|---|---|
| Surface form: | ekmeğim |
| Intermediate form: | ekmeğ0im |
| Lexical form: | ekmek+Hm |
| | |
| Surface form: | tarağın |
| Intermediate form: | tarağ0ın |
| Lexical form: | tarak+Hn |

**RULE 17**: k:g <=> n_ +:0 (X:0) VOWEL

When the first phoneme is morpheme is a vowel and the last phoneme is the stem is consonant "k" then "k" becomes "g".

**Table 4.20 State transition table for Rule 17**

| | k<br>g | k<br>@ | n<br>n | +<br>0 | VOWEL<br>VOWEL | X<br>0 | @<br>@ |
|---|---|---|---|---|---|---|---|
| 1: | 0 | 1 | 2 | 1 | 1 | 1 | 1 |
| 2. | 6 | 3 | 2 | 1 | 1 | 1 | 1 |
| 3: | 0 | 1 | 2 | 4 | 1 | 1 | 1 |
| 4: | 0 | 1 | 2 | 1 | 0 | 5 | 1 |
| 5: | 0 | 1 | 2 | 1 | 0 | 1 | 1 |
| 6. | 0 | 0 | 0 | 7 | 0 | 0 | 0 |
| 7. | 0 | 0 | 0 | 0 | 1 | 8 | 0 |
| 8. | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Examples:

| | |
|---|---|
| Surface form: | dengi |
| Intermediate form: | deng00i |
| Lexical form: | denk+yH |

| | |
|---|---|
| Surface form: | rengimiz |
| Intermediate form: | reng0im00iz |
| Lexical form: | renk+Hm+yHz |

**RULE 18**: g:ğ => _ +:0 (X:0) VOWEL

Some words has a foreign origin in Turkish. If these words ending with "g" and certain suffixes are added then the "g" becomes "ğ".

**Table 4.21 State transition table for Rule 18**

| | g<br>ğ | +<br>0 | X<br>0 | VOWEL<br>VOWEL | @<br>@ |
|---|---|---|---|---|---|
| 1: | 2 | 1 | 1 | 1 | 1 |
| 2. | 0 | 3 | 0 | 0 | 0 |
| 3. | 0 | 0 | 4 | 1 | 0 |
| 4. | 0 | 0 | 0 | 1 | 0 |

Example:

| | |
|---|---|
| Surface form: | dialoğa |
| Intermediate form: | dialoğ00a |
| Lexical form: | dialog+yA |

**RULE 19**: g:ğ /<= n_

If foreign words ending with “g” and “g” is preceded by another consonant then it doesn’t become “ğ”. This consonant may be “n”.

**Table 4.22 State transition table for Rule 19**

| | n | g | @ |
|---|---|---|---|
| | n | ğ | @ |
| 1: | 2 | 1 | 1 |
| 2: | 2 | 0 | 1 |

Example:

| | |
|---|---|
| Surface form: | bumerangım |
| Intermediate form: | bumerang0ım |
| Lexical form: | bumerang+Hm |

**RULE 20**: g:ğ /<= r_

If foreign words ending with “g” then it doesn’t become “ğ” when “g” is preceded by another consonant. This consonant may be “r”.

**Table 4.23 State transition table for Rule 20**

| | c | ç | @ |
|---|---|---|---|
| | ç | @ | @ |
| 1: | 2 | 1 | 1 |
| 2: | 2 | 0 | 1 |

Example:

| | |
|---|---|
| Surface form: | morgunuz |
| Intermediate form: | morg0unuz |
| Lexical form: | morg+HnHz |

**RULE 21**: Y:y <=> _ +:0 [X:0 | H:@]

Rule 21 and Rule 22 deal with nominal roots. These roots are ending with "su". Kemal Oflazer added a lexical "Y" to such nominal words. "Y" is realized as "0" at the end of the word if followed by a consonant, which never drops in affixation.

**Table 4.24 State transition table for Rule 21**

| | Y | Y | + | X | H | @ |
|---|---|---|---|---|---|---|
| | y | @ | 0 | 0 | @ | @ |
| 1: | 2 | 4 | 1 | 1 | 1 | 1 |
| 2. | 0 | 0 | 3 | 0 | 0 | 0 |
| 3. | 0 | 0 | 0 | 1 | 1 | 0 |
| 4: | 2 | 4 | 5 | 1 | 1 | 1 |
| 5: | 2 | 4 | 1 | 0 | 0 | 1 |

Example:

| | |
|---|---|
| Surface form: | akarsuyunuz |
| Intermediate form: | akarsuy00unuz |
| Lexical form: | akarsuY+yHnHz |

**RULE 22**: Y:0 <=> _# | _ +:0 NDCONS:@

**Table 4.25 State transition table for Rule 22**

| | Y | Y | # | + | NDCONS | @ |
|---|---|---|---|---|---|---|
| | 0 | @ | # | 0 | @ | @ |
| 1: | 2 | 3 | 1 | 1 | 1 | 1 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2. | 0 | 0 | 1 | 4 | 0 | 0 |
| 3: | 2 | 3 | 0 | 5 | 1 | 1 |
| 4. | 0 | 0 | 0 | 0 | 1 | 0 |
| 5: | 2 | 3 | 1 | 1 | 0 | 1 |

Example:

| | |
|---|---|
| Surface form: | akarsular |
| Intermediate form: | akarsu00lar |
| Lexical form: | akarsuY+lar |

**RULE 23**: ':0 <=> _# | _+:0 l

This rule remove “’” character from a lexical proper noun under certain circumstances.

**Table 4.26 State transition table for Rule 23**

| | ‘ | ‘ | # | + | 1 | @ |
|---|---|---|---|---|---|---|
| | 0 | @ | # | 0 | 1 | @ |
| 1: | 2 | 3 | 1 | 1 | 1 | 1 |
| 2. | 0 | 0 | 1 | 4 | 0 | 0 |
| 3: | 2 | 3 | 0 | 5 | 1 | 1 |
| 4. | 0 | 0 | 0 | 0 | 1 | 0 |
| 5: | 2 | 3 | 1 | 1 | 0 | 0 |

### 4.2.3 Morpheme Order Rules for Turkish Language

There are two main classes for Turkish roots: *nominal* and *verbal*. The importances of these classes are presented in section 2.2.1. Two rules are produces for these two paradigms in this project.

**Rule about nominal paradigm:**

The state diagram for the nominal model is shown in Figure 3.2. This rule is produced this model for this project. State transition table for nominal model is shown in Table 4.27.

**Table 4.27 State transition table for nominal model**

| | Plural Suffix | Possessive Suffix | Case Suffix | Relative Suffix |
|---|---|---|---|---|
| 1: | 2 | 3 | 4 | 5 |
| 2: | 0 | 3 | 4 | 5 |
| 3: | 0 | 0 | 4 | 5 |
| 4: | 0 | 0 | 0 | 5 |
| 5: | 0 | 0 | 0 | 0 |

**Rule about verbal paradigm:**

The state diagram for the verbal model is shown in Figure 3.4. This rule is produced this model for this project. State transition table for verbal model is shown in Table 4.28.

**Table 4.28 State transition table for verbal model**

| | Voice Suffixes | Negation Suffix | Compound Verb Suffix | Main Tense Suffix | Question Suffix | Second Tense Suffix | Person Suffix |
|---|---|---|---|---|---|---|---|
| 1: | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 2. | 0 | 3 | 4 | 5 | 0 | 0 | 0 |
| 3. | 0 | 0 | 4 | 5 | 0 | 0 | 0 |
| 4. | 0 | 0 | 0 | 5 | 0 | 0 | 0 |
| 5: | 0 | 0 | 0 | 0 | 6 | 7 | 8 |
| 6. | 0 | 0 | 0 | 0 | 0 | 7 | 8 |
| 7. | 0 | 0 | 0 | 0 | 0 | 0 | 8 |
| 8: | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# CHAPTER FIVE
# SOFTWARE DESIGN AND IMPLEMENTATION

This chapter describes the design and implementation of Turkish Morphological Analyzer project. The project consists of two parts: a library to analyze words and an application that uses this library. This library is a reusable software tool for analyses of Turkish text and the application is developed for testing this library.

The application and the library are developed by Borland C++ Builder Version 6.0. This library reads Turkish rule definitions and analyzes the input word according these Turkish rule definitions. All Turkish rule definitions are stored in XML files.

A drawing tool is developed for morphological analyzer of Turkish language. (Duran & Kürkçü, 2004). This tool is used to draw finite state machine. Turkish rules in “Rule.xml” documents can be opened by this tool. Rules can be seen as finite state machine and can be changed by this tool if necessary.

## 5.1 Turkish Rule Definitions

Entity relationship diagram for these tables are shown in Figure 5.1 and Figure 5.2. All are in XML files.

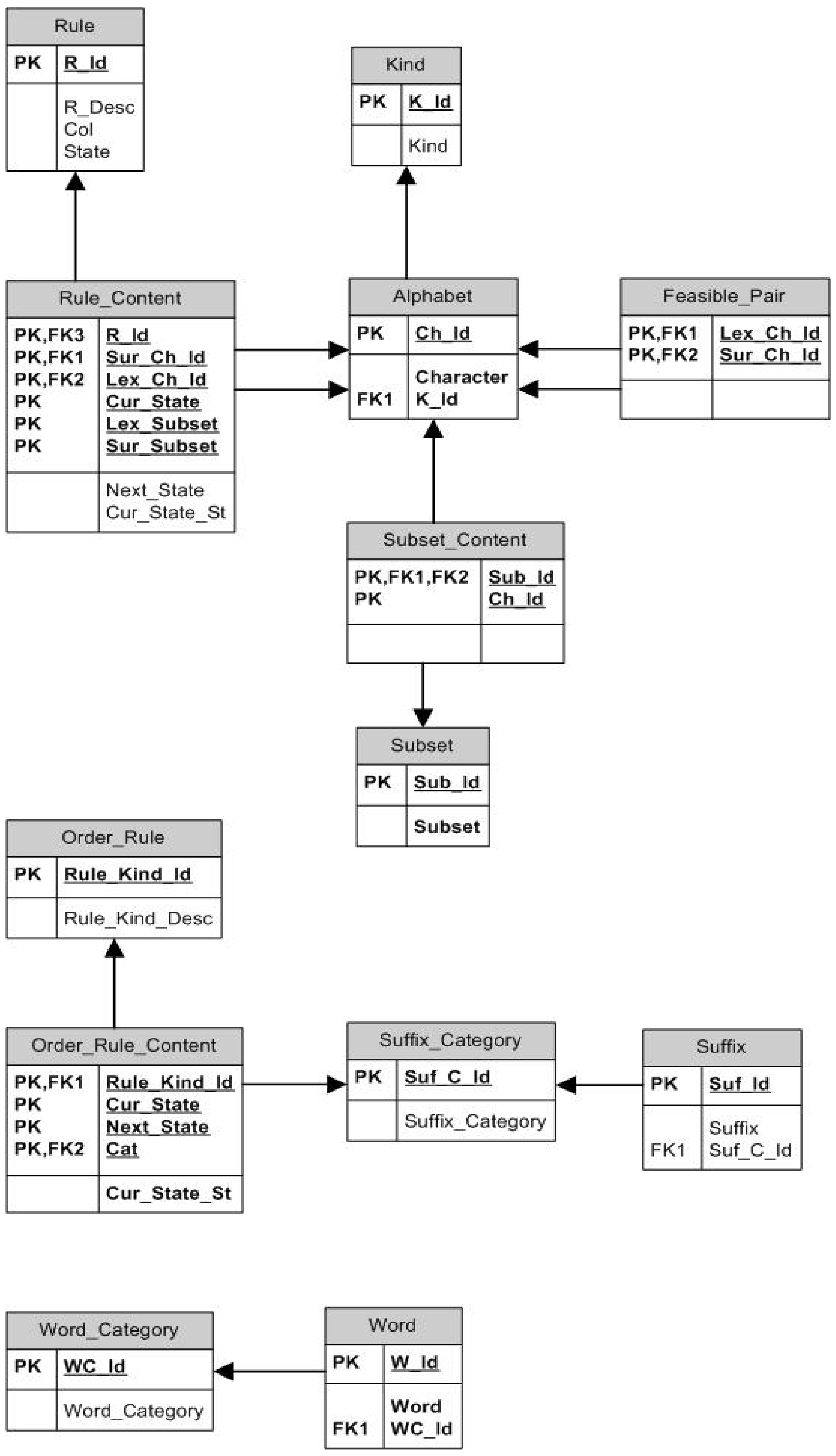


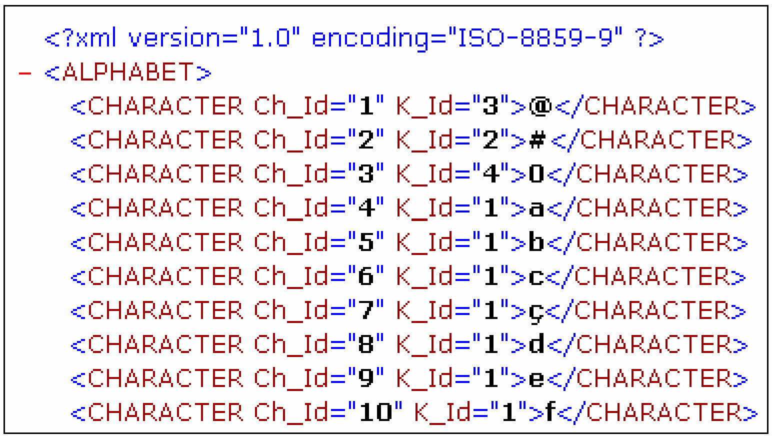
**Figure 5.1 ER Diagram**

Turkish rule definitions contain the followings:

- Alphabetic Characters

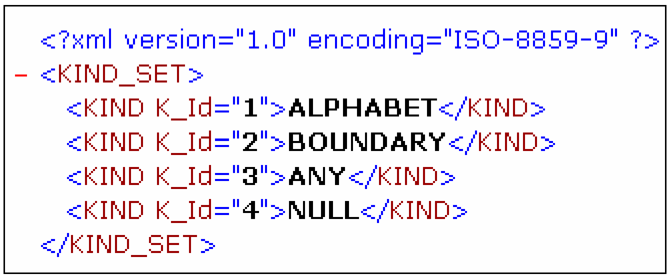
Turkish alphabetic characters are stored in “Alphabet.xml”. *Null, any* and *boundary* characters are not alphabetic character but in this project these characters are stored in alphabetic characters document. Figure 5.2 shows a part of “Alphabet.xml” file.

Null character is “0”, any character is “@” and boundary character is “#”. Types of these characters are stored in “Kind.xml”. Figure 5.3 shows all character types.



```
  <?xml version="1.0" encoding="ISO-8859-9" ?>
- <ALPHABET>
    <CHARACTER Ch_Id="1" K_Id="3">@</CHARACTER>
    <CHARACTER Ch_Id="2" K_Id="2">#</CHARACTER>
    <CHARACTER Ch_Id="3" K_Id="4">0</CHARACTER>
    <CHARACTER Ch_Id="4" K_Id="1">a</CHARACTER>
    <CHARACTER Ch_Id="5" K_Id="1">b</CHARACTER>
    <CHARACTER Ch_Id="6" K_Id="1">c</CHARACTER>
    <CHARACTER Ch_Id="7" K_Id="1">ç</CHARACTER>
    <CHARACTER Ch_Id="8" K_Id="1">d</CHARACTER>
    <CHARACTER Ch_Id="9" K_Id="1">e</CHARACTER>
    <CHARACTER Ch_Id="10" K_Id="1">f</CHARACTER>
```

**Figure 5.2 Part of characters of Turkish language**



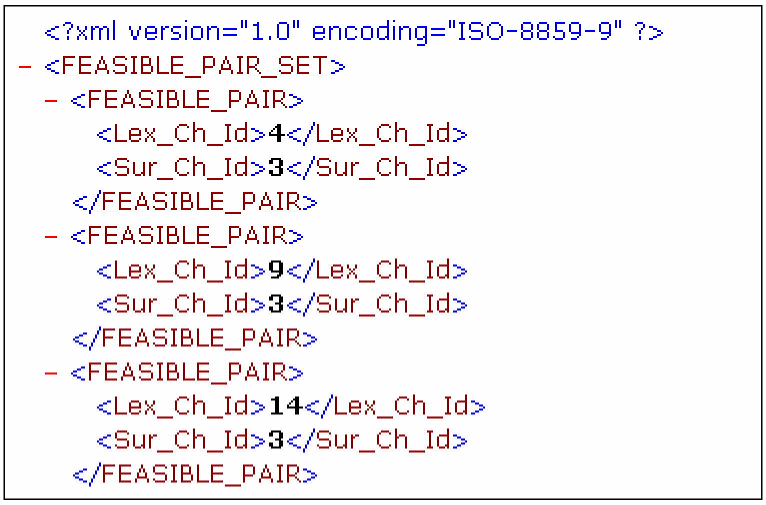
```
  <?xml version="1.0" encoding="ISO-8859-9" ?>
- <KIND_SET>
    <KIND K_Id="1">ALPHABET</KIND>
    <KIND K_Id="2">BOUNDARY</KIND>
    <KIND K_Id="3">ANY</KIND>
    <KIND K_Id="4">NULL</KIND>
  </KIND_SET>
```

**Figure 5.3 All Character types in Turkish language**

- Feasible Pairs

The all of the default and special correspondences makes up the set of feasible pairs. Turkish feasible pairs are consist of the pairs in Turkish rules. Turkish feasible pairs are stored in "Feasible_Pair.xml". Figure 5.4 shows a part of "Feasible_Pair.xml" file.

"Lex_Ch_Id = 4" and "Sur_Ch_Id = 3" is a feasible pair. It determines lexical character is "a" and surface character is "0". This pair is shown as "a:0" as shortly.

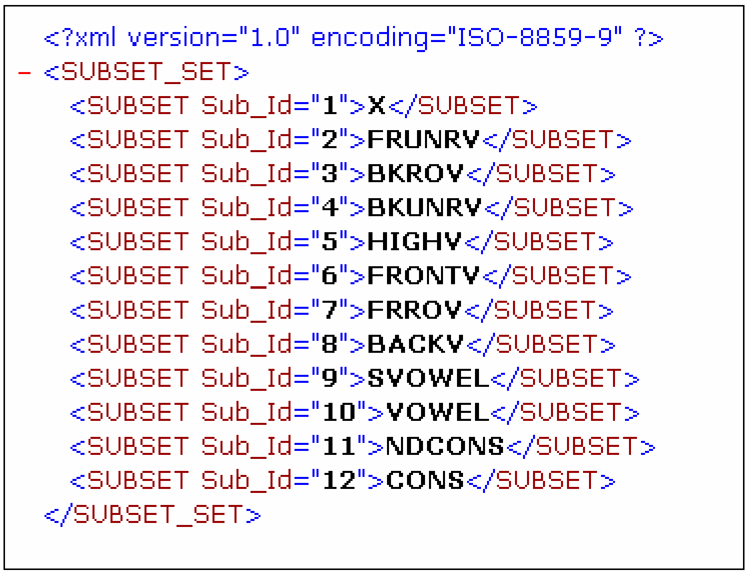


```
  <?xml version="1.0" encoding="ISO-8859-9" ?>
- <FEASIBLE_PAIR_SET>
  - <FEASIBLE_PAIR>
      <Lex_Ch_Id>4</Lex_Ch_Id>
      <Sur_Ch_Id>3</Sur_Ch_Id>
    </FEASIBLE_PAIR>
  - <FEASIBLE_PAIR>
      <Lex_Ch_Id>9</Lex_Ch_Id>
      <Sur_Ch_Id>3</Sur_Ch_Id>
    </FEASIBLE_PAIR>
  - <FEASIBLE_PAIR>
      <Lex_Ch_Id>14</Lex_Ch_Id>
      <Sur_Ch_Id>3</Sur_Ch_Id>
    </FEASIBLE_PAIR>
```

**Figure 5.4 Part of feasible pairs of Turkish language**

- Subsets

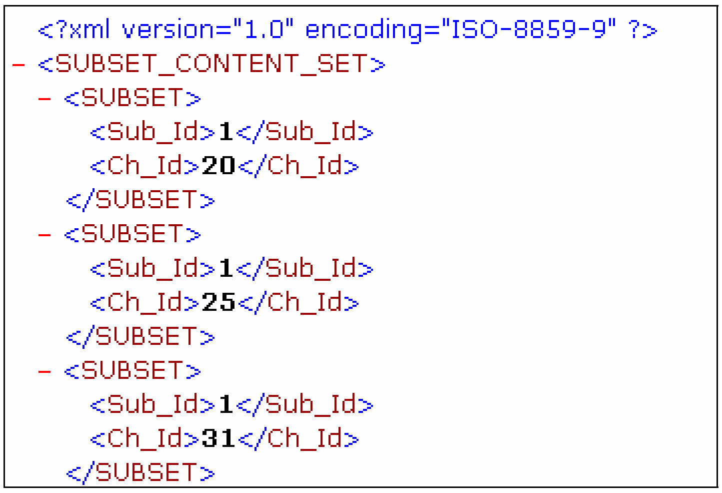
There are 12 subsets in Turkish rule definitions. Turkish subsets are stored "Subset.xml" and "Subset_Content.xml". Figure 5.5 shows all subsets and Figure 5.6 shows some subset elements in Turkish rule definitions.



```
  <?xml version="1.0" encoding="ISO-8859-9" ?>
- <SUBSET_SET>
    <SUBSET Sub_Id="1">X</SUBSET>
    <SUBSET Sub_Id="2">FRUNRV</SUBSET>
    <SUBSET Sub_Id="3">BKROV</SUBSET>
    <SUBSET Sub_Id="4">BKUNRV</SUBSET>
    <SUBSET Sub_Id="5">HIGHV</SUBSET>
    <SUBSET Sub_Id="6">FRONTV</SUBSET>
    <SUBSET Sub_Id="7">FRROV</SUBSET>
    <SUBSET Sub_Id="8">BACKV</SUBSET>
    <SUBSET Sub_Id="9">SVOWEL</SUBSET>
    <SUBSET Sub_Id="10">VOWEL</SUBSET>
    <SUBSET Sub_Id="11">NDCONS</SUBSET>
    <SUBSET Sub_Id="12">CONS</SUBSET>
  </SUBSET_SET>
```

**Figure 5.5 All Subsets for Turkish language**

Example: "X" is a subset and it includes three alphabetic characters in it. "Sub_Id = 1" identifies subset "X". "Ch = 20" identifies alphabetic character "n". "Ch = 25" identifies alphabetic character "s". "Ch = 31" identifies alphabetic character "y".



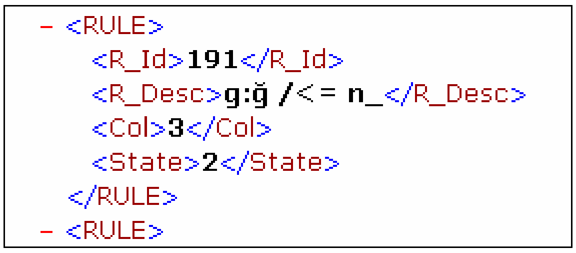
```
  <?xml version="1.0" encoding="ISO-8859-9" ?>
- <SUBSET_CONTENT_SET>
  - <SUBSET>
      <Sub_Id>1</Sub_Id>
      <Ch_Id>20</Ch_Id>
    </SUBSET>
  - <SUBSET>
      <Sub_Id>1</Sub_Id>
      <Ch_Id>25</Ch_Id>
    </SUBSET>
  - <SUBSET>
      <Sub_Id>1</Sub_Id>
      <Ch_Id>31</Ch_Id>
    </SUBSET>
```

**Figure 5.6 Part of Subset Content for Turkish language**

- Rules

There are 23 rules in Turkish rule definitions. Turkish rules are stored in "Rule.xml" and "Rule_Content.xml".

Figure 5.7 shows Rule number 19 in XML document. State transition table for Rule number 19 is shown in Table 4.22 in chapter four. This rule has three properties: description, number of columns and number od states. These are stored in rule document. Number of columns determines the number of feasible pair in this rule.



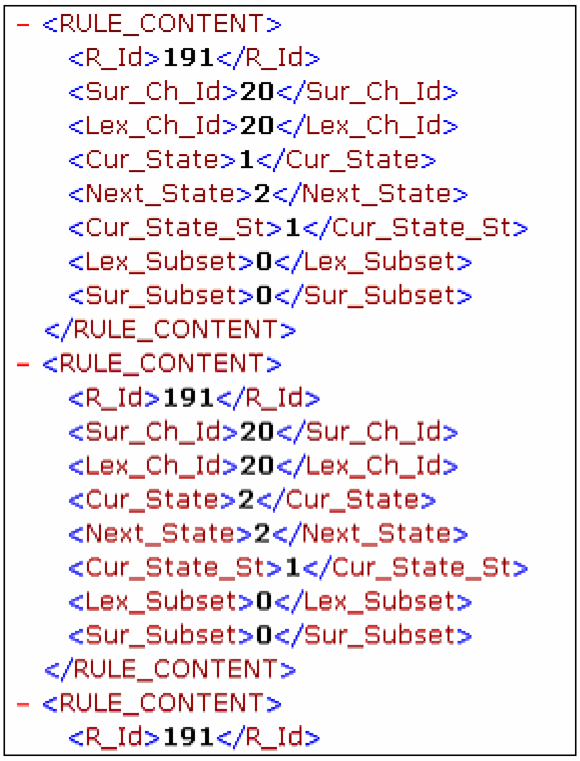
```
- <RULE>
    <R_Id>191</R_Id>
    <R_Desc>g:ğ /<= n_</R_Desc>
    <Col>3</Col>
    <State>2</State>
  </RULE>
- <RULE>
```

**Figure 5.7 RULE 19 "g:ğ /<= n_" as in XML document**

Figure 5.8 shows the detailed contents of rule number 191. This rule has three feasible pair: "n:n", "g:ğ", "@:@". There are two states: "1" and "2". Two states are final in rule number 191. "Cur_State_St" property determines the current state situation that is final or non-final states. It may be "0" or "1". If this property is "1" then this state is final otherwise is non-final state.

"Lex_Subset" and "Sur_Subset" properties determine whether the feasible pair is a subset or a character. These properties may be "0" or "1". If "Lex_Subset" property is "0" then it means lexical part is a character otherwise it is a subset. If "Sur_Subset" property is "0" then it means surface part is a character otherwise it is a subset.

"Cur_State" property determines the current state number and "Next_State" property determines the next state number.

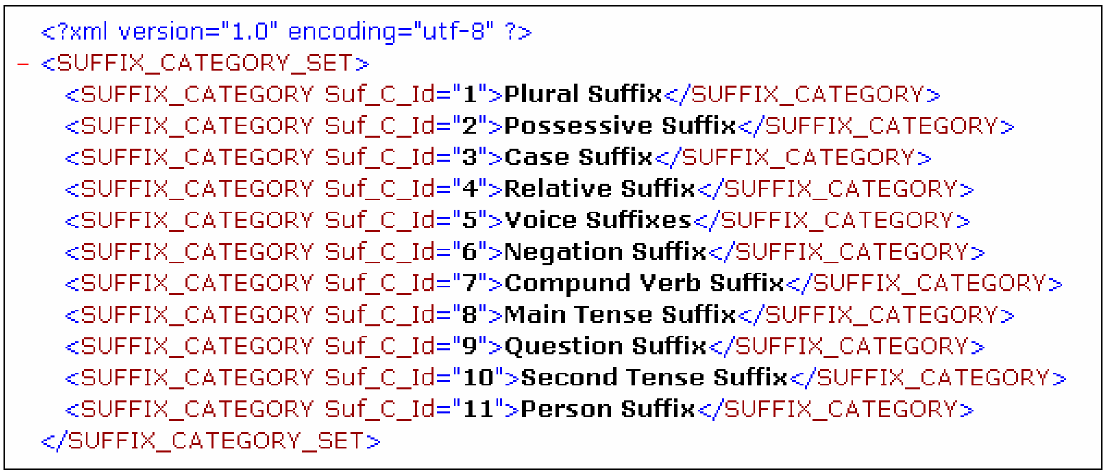


```
- <RULE_CONTENT>
    <R_Id>191</R_Id>
    <Sur_Ch_Id>20</Sur_Ch_Id>
    <Lex_Ch_Id>20</Lex_Ch_Id>
    <Cur_State>1</Cur_State>
    <Next_State>2</Next_State>
    <Cur_State_St>1</Cur_State_St>
    <Lex_Subset>0</Lex_Subset>
    <Sur_Subset>0</Sur_Subset>
  </RULE_CONTENT>
- <RULE_CONTENT>
    <R_Id>191</R_Id>
    <Sur_Ch_Id>20</Sur_Ch_Id>
    <Lex_Ch_Id>20</Lex_Ch_Id>
    <Cur_State>2</Cur_State>
    <Next_State>2</Next_State>
    <Cur_State_St>1</Cur_State_St>
    <Lex_Subset>0</Lex_Subset>
    <Sur_Subset>0</Sur_Subset>
  </RULE_CONTENT>
- <RULE_CONTENT>
    <R_Id>191</R_Id>
```

**Figure 5.8 Part of rule content data of Rule 19**

- Suffix

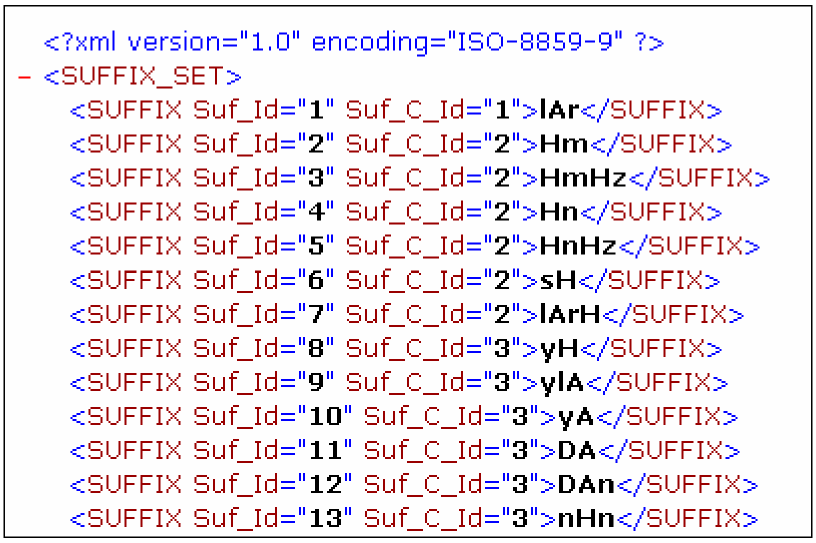
There are many suffixes in Turkish language. These suffixes are categorized in “Two-level description of Turkish Morphology” study of Oflazer. (Oflazer, 1993). These suffixes are used to analyze word. After each step of analyzing process, analyzer controls the string in suffix list or word list.



```
  <?xml version="1.0" encoding="utf-8" ?>
- <SUFFIX_CATEGORY_SET>
    <SUFFIX_CATEGORY Suf_C_Id="1">Plural Suffix</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="2">Possessive Suffix</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="3">Case Suffix</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="4">Relative Suffix</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="5">Voice Suffixes</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="6">Negation Suffix</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="7">Compund Verb Suffix</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="8">Main Tense Suffix</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="9">Question Suffix</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="10">Second Tense Suffix</SUFFIX_CATEGORY>
    <SUFFIX_CATEGORY Suf_C_Id="11">Person Suffix</SUFFIX_CATEGORY>
  </SUFFIX_CATEGORY_SET>
```

**Figure 5.9 All Suffix categories of Turkish language**

Turkish suffix categories are stored in "Suffix_Category.xml" and Turkish suffixes are stored in "Suffix.xml". Figure 5.9 shows all suffix categories. A part of suffixes are shown in Figure 5.10.

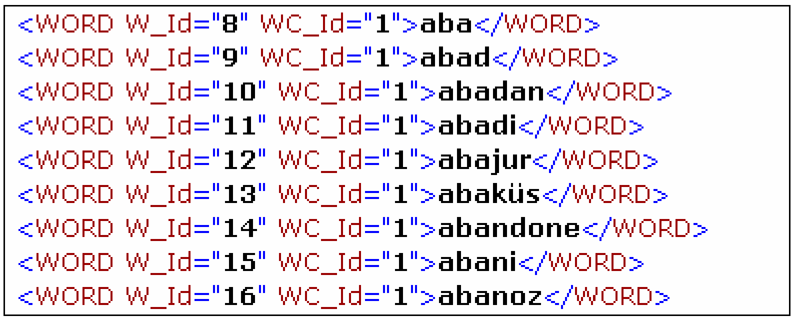


```
 <?xml version="1.0" encoding="ISO-8859-9" ?>
- <SUFFIX_SET>
   <SUFFIX Suf_Id="1" Suf_C_Id="1">lAr</SUFFIX>
   <SUFFIX Suf_Id="2" Suf_C_Id="2">Hm</SUFFIX>
   <SUFFIX Suf_Id="3" Suf_C_Id="2">HmHz</SUFFIX>
   <SUFFIX Suf_Id="4" Suf_C_Id="2">Hn</SUFFIX>
   <SUFFIX Suf_Id="5" Suf_C_Id="2">HnHz</SUFFIX>
   <SUFFIX Suf_Id="6" Suf_C_Id="2">sH</SUFFIX>
   <SUFFIX Suf_Id="7" Suf_C_Id="2">lArH</SUFFIX>
   <SUFFIX Suf_Id="8" Suf_C_Id="3">yH</SUFFIX>
   <SUFFIX Suf_Id="9" Suf_C_Id="3">ylA</SUFFIX>
   <SUFFIX Suf_Id="10" Suf_C_Id="3">yA</SUFFIX>
   <SUFFIX Suf_Id="11" Suf_C_Id="3">DA</SUFFIX>
   <SUFFIX Suf_Id="12" Suf_C_Id="3">DAn</SUFFIX>
   <SUFFIX Suf_Id="13" Suf_C_Id="3">nHn</SUFFIX>
```

**Figure 5.10 Part of suffixes of Turkish language**

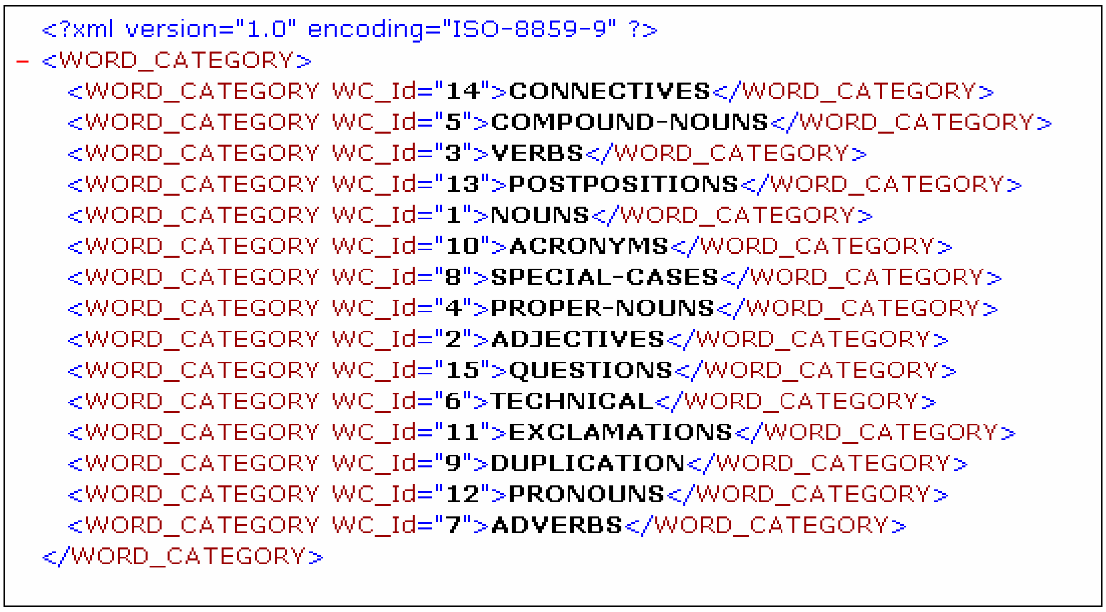
- Words

There are approximately 26.000 roots words in Turkish language. These words are used to determine the stem of input word while analyzing word. Turkish word categories are stored in "Word_Category.xml" and Turkish words are stored in "Word.xml". Figure 5.12 shows a part of word categories of Turkish language. A part of words are shown in Figure 5.11.



```
<WORD W_Id="8" WC_Id="1">aba</WORD>
<WORD W_Id="9" WC_Id="1">abad</WORD>
<WORD W_Id="10" WC_Id="1">abadan</WORD>
<WORD W_Id="11" WC_Id="1">abadi</WORD>
<WORD W_Id="12" WC_Id="1">abajur</WORD>
<WORD W_Id="13" WC_Id="1">abaküs</WORD>
<WORD W_Id="14" WC_Id="1">abandone</WORD>
<WORD W_Id="15" WC_Id="1">abani</WORD>
<WORD W_Id="16" WC_Id="1">abanoz</WORD>
```

**Figure 5.11 Part of words of Turkish language**



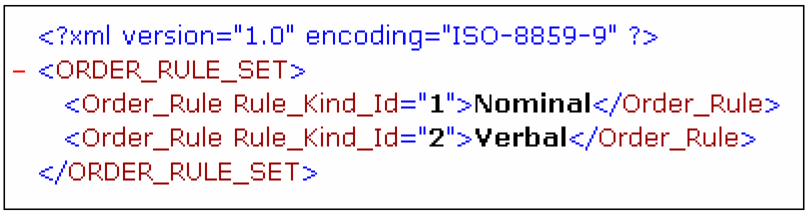
```
  <?xml version="1.0" encoding="ISO-8859-9" ?>
- <WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="14">CONNECTIVES</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="5">COMPOUND-NOUNS</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="3">VERBS</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="13">POSTPOSITIONS</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="1">NOUNS</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="10">ACRONYMS</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="8">SPECIAL-CASES</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="4">PROPER-NOUNS</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="2">ADJECTIVES</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="15">QUESTIONS</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="6">TECHNICAL</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="11">EXCLAMATIONS</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="9">DUPLICATION</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="12">PRONOUNS</WORD_CATEGORY>
    <WORD_CATEGORY WC_Id="7">ADVERBS</WORD_CATEGORY>
  </WORD_CATEGORY>
```

**Figure 5.12 All word categories of Turkish language**

- Morpheme Order Rules

There are two main classes for Turkish roots: *nominal* and *verbal*. These are presented in section 3.2.1. These two paradigms are represented in two rules. These rules are called morpheme order rules.

These rules are stored in “Order_Rule.xml” and “Order_Rule_Content.xml”. Figure 5.13 shows order rule’s header information in XML document.



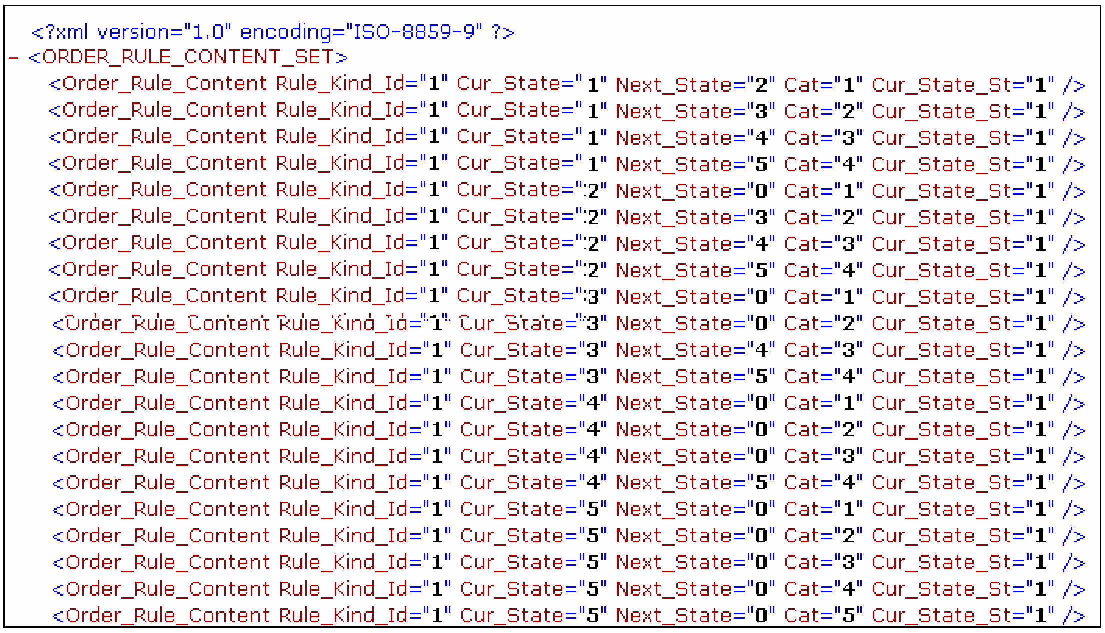
```
  <?xml version="1.0" encoding="ISO-8859-9" ?>
- <ORDER_RULE_SET>
    <Order_Rule Rule_Kind_Id="1">Nominal</Order_Rule>
    <Order_Rule Rule_Kind_Id="2">Verbal</Order_Rule>
  </ORDER_RULE_SET>
```

**Figure 5.13 Morpheme order rules**

State transition table for nominal model is shown in Table 4.27 and verbal model in Table 4.28 in chapter four. Nominal rule has two properties: *id* and *description*. These are stored in rule document.

Figure 5.14 shows the detail contents of nominal rule. There are five states that are final. There are five properties:

- o "Rule_Kind_Id" determines the identity of nominal rule,
- o "Cur_State" determines the current state number,
- o "Next_State" determines the next state number under certain situation,
- o "Cat" determines the identity of the suffix categories.,
- o "Cur_State_St" determines the current state situation that is final or non-final states. It may be "0" or "1". If this property is "1" then this state is final otherwise is non-final state.



```
  <?xml version="1.0" encoding="ISO-8859-9" ?>
- <ORDER_RULE_CONTENT_SET>
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="1" Next_State="2" Cat="1" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="1" Next_State="3" Cat="2" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="1" Next_State="4" Cat="3" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="1" Next_State="5" Cat="4" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="2" Next_State="0" Cat="1" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="2" Next_State="3" Cat="2" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="2" Next_State="4" Cat="3" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="2" Next_State="5" Cat="4" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="3" Next_State="0" Cat="1" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="3" Next_State="0" Cat="2" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="3" Next_State="4" Cat="3" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="3" Next_State="5" Cat="4" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="4" Next_State="0" Cat="1" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="4" Next_State="0" Cat="2" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="4" Next_State="0" Cat="3" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="4" Next_State="5" Cat="4" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="5" Next_State="0" Cat="1" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="5" Next_State="0" Cat="2" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="5" Next_State="0" Cat="3" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="5" Next_State="0" Cat="4" Cur_State_St="1" />
    <Order_Rule_Content Rule_Kind_Id="1" Cur_State="5" Next_State="0" Cat="5" Cur_State_St="1" />
```

**Figure 5.14 Detail contents of the Nominal rule**

## 5. 2 Implementation of the Project

This project consists of two parts: a library to analyze words and an example application that uses this library. The library is developed by Borland C++ Builder Version 6.0. This library has two main functions: *load data* and *analyze word.*

- *Load data* function must be run before analyze a word. This function reads all data about the Turkish rule definitions from XML documents and stored in memory as string array to faster operations. XML documents are in *Data* folder in application directory. These arrays are global variables for library. *TXMLDocument* object is used to read XML document in Borland C++ Builder. Load_Data functions call these functions:

    - Fill_Alphabet function finds alphabetic characters. It reads “Kind.xml” document and gets the alphabetic character’s kind identification then reads “Alphabet.xml” document for this kind identification. After reading data it stores these alphabetic characters in *Alphabet* array.

    - Find_Boundary_Character function finds boundary symbol and stores it in a global variable. It reads “Kind.xml” document and gets the boundary symbol’s kind identification then reads “Alphabet.xml” document for this kind identification.

    - Find_Any_Character function finds any symbol and stores it in a global variable. It reads “Kind.xml” document and gets the any symbol’s kind identification then reads “Alphabet.xml” document for this kind identification.

    - Find_Null_Character function finds NULL symbol and stores it in a global variable. It reads “Kind.xml” document and gets the NULL symbol’s kind identification then reads “Alphabet.xml” document for this kind identification.

    - Fill_Subset_Content function finds subset contents. It reads “Subset_Content.xml” document. After reading data it stores these subset content data in *Subset_Content* array.
    - Fill_Rule function finds rule headers data. It reads “Rule.xml” document. After reading data, rule description data are stored in

*Rule_Description* array. Other rule header data are stored in *Rule* array

- Fill_Rule_Content function finds rule content data. It reads "Rule_Content.xml" document. After reading data, data are stored in *Rule_Content* array.

- Fill_Feasible_Pair function finds feasible pairs data. It reads "Feasible_Pair.xml" document. After reading data, data are stored in *Feasible_Pair* array. *Feasible_Pair* is a two-dimensional array. First dimension represents to the pair of lexical and the second dimension represents to the pair of surface.

- Fill_Word_Category function finds word categories data from "Word_Category.xml" document. After reading data, data are stored in *Word_Category* array.

- Fill_Word_Entry function finds words data from "Word.xml" document. After reading data, data are stored in *Word* array.

- Fill_Suffix_Category function finds suffix categories data from "Suffix_Category.xml" document. After reading data, data are stored in *Suffix_Category* array.

- Fill_Suffix_Entry function finds suffixes data from "Suffix.xml" document. After reading data, data are stored in *Suffix* array.

- Fill_Order_Rule function finds order rule headers data. It reads "Order_Rule.xml" document. After reading data, order rule description data are stored in *Order_Rule_Description* array. Other rule header data are stored in *Order_Rule* array

- Fill_Order_Rule_Content function finds rule content data. It reads “Order_Rule_Content.xml” document. After reading data, data are stored in *Order_Rule_Content* array.

- *Analyze word* function aims to analyze words as stem and suffixes. It determines the types of stem and suffixes at the same time. This function gets the word that is a surface form as input. If analyzer founds a result then it continues looking for additional results. It returns a structure that is called *MY_RESULT.* This structure has four elements: *My_Way, My_Alter, My_Way_File_Name and My_Alter_File_Name.* All these elements are string data type.

  - *My_Way* includes the ways of all alternatives of analysis
  - *My_Alter* includes all alternative results of analysis.
  - *My_Way_File_Name* is the path of XML document for *My_Way*
  - *My_Alter_File_Name* is the path of XML document for *My_Alter*

This function also creates two XML documents: *Result* and *Result_Way*. It saves *My_Way* and *My_Alter* to these XML documents. The data of *My_Way* are stored in *Result_Way* XML document and the data of *My_Result* are stored in *Result* XML document. These documents are created in Debug folder in the application directory. These XML documents are presented in following subsection.

**Result.xml**

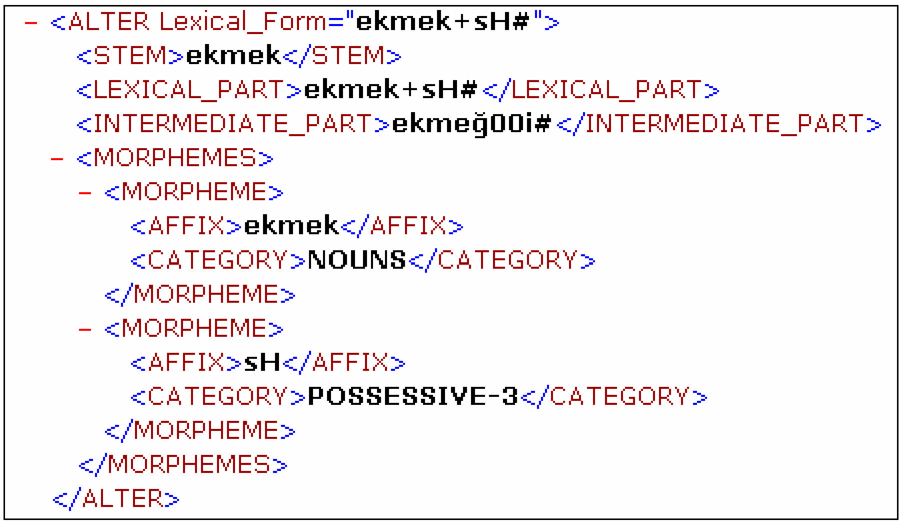
The alternative outputs are in “Result.xml” document containing the following data are stored in this document for each alternation

- *Stem*: It is the stem of this alternation,
- *Lexical_Part*: It is lexical form of this alternation,
- *Intermadiate_Part*: It is an intermediate form of this alternation,

- *Morphemes*: It is a set of all morphemes and their data of this alternation. Each morpheme of the alternation has following data:

    - *Affix*: This is the morpheme,
    - *Category*: This is the category of this morpheme

Figure 5.15 shows the example that is one of the alternations of input word "ekmeği".

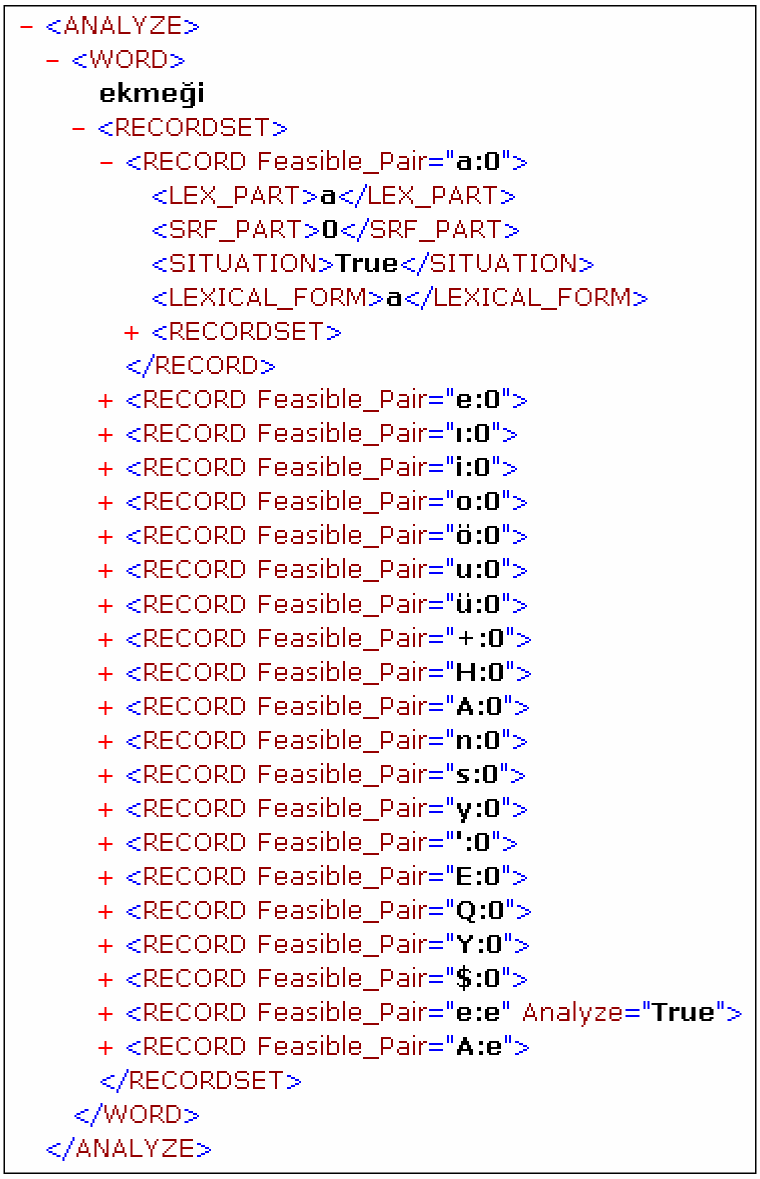


```
- <ALTER Lexical_Form="ekmek+sH#">
    <STEM>ekmek</STEM>
    <LEXICAL_PART>ekmek+sH#</LEXICAL_PART>
    <INTERMEDIATE_PART>ekmeğ00i#</INTERMEDIATE_PART>
  - <MORPHEMES>
    - <MORPHEME>
        <AFFIX>ekmek</AFFIX>
        <CATEGORY>NOUNS</CATEGORY>
      </MORPHEME>
    - <MORPHEME>
        <AFFIX>sH</AFFIX>
        <CATEGORY>POSSESSIVE-3</CATEGORY>
      </MORPHEME>
    </MORPHEMES>
  </ALTER>
```

**Figure 5.15 One of alternations of input word "ekmeği"**

**Result_Way.xml**

The steps of the alternative outputs are in "Result_Way.xml" document. The purpose of this document is to allow to user to see how a surface form is processed so if a rule is wrong or dictionary is incomplete then the user can see the problem and change rules or dictionary if necessary. Figure 5.16 shows an example of Result_Way.xml document as not detail.



```
- <ANALYZE>
  - <WORD>
      ekmeği
    - <RECORDSET>
      - <RECORD Feasible_Pair="a:0">
          <LEX_PART>a</LEX_PART>
          <SRF_PART>0</SRF_PART>
          <SITUATION>True</SITUATION>
          <LEXICAL_FORM>a</LEXICAL_FORM>
        + <RECORDSET>
        </RECORD>
      + <RECORD Feasible_Pair="e:0">
      + <RECORD Feasible_Pair="ı:0">
      + <RECORD Feasible_Pair="i:0">
      + <RECORD Feasible_Pair="o:0">
      + <RECORD Feasible_Pair="ö:0">
      + <RECORD Feasible_Pair="u:0">
      + <RECORD Feasible_Pair="ü:0">
      + <RECORD Feasible_Pair="+:0">
      + <RECORD Feasible_Pair="H:0">
      + <RECORD Feasible_Pair="A:0">
      + <RECORD Feasible_Pair="n:0">
      + <RECORD Feasible_Pair="s:0">
      + <RECORD Feasible_Pair="y:0">
      + <RECORD Feasible_Pair="':0">
      + <RECORD Feasible_Pair="E:0">
      + <RECORD Feasible_Pair="Q:0">
      + <RECORD Feasible_Pair="Y:0">
      + <RECORD Feasible_Pair="$:0">
      + <RECORD Feasible_Pair="e:e" Analyze="True">
      + <RECORD Feasible_Pair="A:e">
      </RECORDSET>
    </WORD>
  </ANALYZE>
```

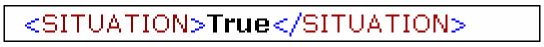
**Figure 5.16 Example of *Result_Way.xml* document as not detail**

The following data are stored in this document:

- *Word*: This is an input word,
- *Recordset*: This is a set of data of each feasible pair. There are following data for each *recordset*:
    - *Record*: It has two parameters: *Feasible Pair* and *Analyze*. *Feasible Pair* is the feasible pair (lexical:surface pair) that is

currently being considered. This parameter exists in every *Record*. *Analyze* is a boolean parameter. If result is found then this parameter will be *true* otherwise *false*.

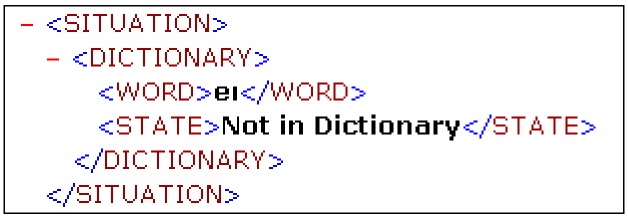
- *Lex_Part*: This is the lexical character of the feasible pair,
- *Srf_Part*: This is the surface character of the feasible pair,
- *Situation*: It represents the result of the one step of the analyzing operation. There are four alternatives for *situation*.
  - If this step is accepted by all rules and lexical form is in dictionary then *situation* will be true. This case is shown in the Figure 5.17.



```
<SITUATION>True</SITUATION>
```

**Figure 5.17 Example *Situation* result - I**

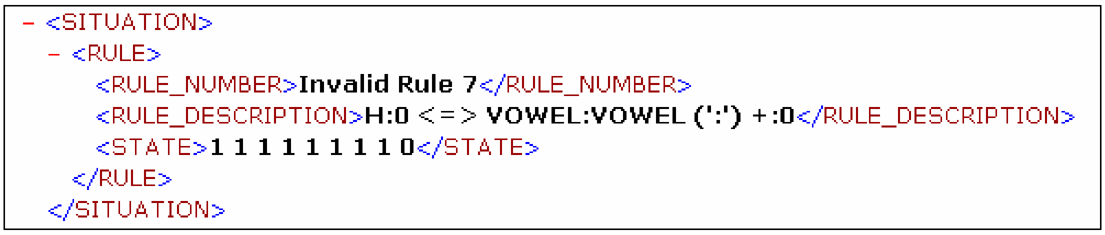
  - If current lexical form is not in the dictionary then *situation* represents it. *Word* represents the lexical form and *state* represents reason. This case is shown in Figure 5.18.



```
- <SITUATION>
  - <DICTIONARY>
      <WORD>eı</WORD>
      <STATE>Not in Dictionary</STATE>
    </DICTIONARY>
  </SITUATION>
```

**Figure 5.18 Example *Situation* result - II**

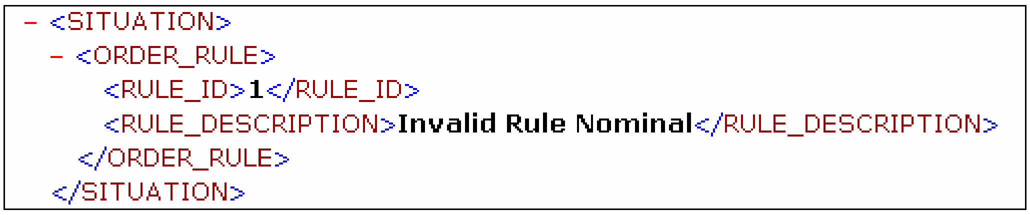
  - If this feasible pair is not accepted at least one rule then *situation* represents it. It contains information about this rule that does not accept. *RULE_NUMBER* represents the number of rule, *RULE_DESCRIPTION* represents the description of the rule, *and STATE* represents the set of current states. The leftmost number is the state of rule 1, the second of is rule 2, and so on. This case is shown in Figure 5.19.



```
- <SITUATION>
  - <RULE>
      <RULE_NUMBER>Invalid Rule 7</RULE_NUMBER>
      <RULE_DESCRIPTION>H:0 <=> VOWEL:VOWEL (':') +:0</RULE_DESCRIPTION>
      <STATE>1 1 1 1 1 1 1 1 0</STATE>
    </RULE>
  </SITUATION>
```

**Figure 5.19 Example *Situation* result - III**

- If the stem of the input is a noun or adjective then nominal order rule is applied. If the stem of the input is a verb then verbal order rule is applied. If these two rules fail then *situation* represents it. *Rule_Id* represents the identity of the order rule. *Rule_Description* represents the description of the order rule. An example case is shown in Figure 5.20.



```
- <SITUATION>
  - <ORDER_RULE>
      <RULE_ID>1</RULE_ID>
      <RULE_DESCRIPTION>Invalid Rule Nominal</RULE_DESCRIPTION>
    </ORDER_RULE>
  </SITUATION>
```

**Figure 5.20 Example *Situation* result - IV**

- o *Lexical_Form*: This is the current value of the result lexical form.

- o *Recordset*: This is the same with the *recordset* above. This is a set of data for next feasible pair.

- *Rule_Analyze* is called by *Analyze_Word* function for execution of analysis operation. This function does the main operations. It a recursive function, it computes lexical forms from the surface form by recursively. Its process is left to right and one character of the surface input form is processed at a time. *Rule_Analye* function has following inputs:

- *Total_Result_Lexical* is initially empty. This string gets longer while analyzing. It holds the lexical form of the result.

- *Total_Result_Intermediate* is initially empty. This string gets longer while analyzing. It holds the intermediate form of the result.

- *xNode* represents the step of analyzing process. Steps of analyzing are stored in tree structure. *xNode* belongs to this structure.

- *Result_Node* represents the correct results. Results are stored in tree structure. *Result_Node* belongs to this structure.

- *My_State* is an array. It represents the current state of all rules. "1" is the initial state for all rules, elements of this array is set to "1".

- *Sozcuk* is a string variable. It represents the input word. This word is in surface form. The symbol "#" concatenate to this word. The symbol "#" means that the end of the word. This string gets shorter while analyzing. After each of the steps the first character of this variable is dropped i.e. initially it is "ekmeği#", it is "kmeği#" in the second step. If its length is 1 then process is finished.

- *Result* is initially empty. It holds the last morpheme that is found.

- *Kok* is a boolean variable. It is initially *true.* If the stem of the input word is found then this variable is set to *false.*

## 5.3 Functions in Library

The following some functions in library to help to analyze the surface form. The role of the each function is represented in the following.

- Find_In_Dictionary

  It has two inputs: *word* and *statu.* If *statu* is equal to *false* then this function finds the word in set of words. If it is equal to *true* then this function finds the word in set of suffixes. This function returns *true* or *false.* If it finds then returns *true* else *false.*

- Ara_Rule_Content

  It has three inputs: *Lexical_Harf_Id*, *Surface_Harf_Id* and *My_State. Lexical_Harf_Id* is lexical part of current feasible pair and *Surface_Harf_Id* is surface part of current feasible pair. *My_State* is a set of current state numbers for all rules.

  This function returns a structure. This structure has two elements: *count* and *Set_Feasible_Pair. Count* is an integer variable. It represents the equivalent rules count. *Set_Feasible_Pair* is a five dimensional array. Its elements represent:

  - First index represents the lexical character id,
  - Second index represents the surface character id,
  - Third index represents the next state,
  - Fourth index represents final/non-final state,
  - Fifth index represents the rule id,

- Determine_Categories

  This function is used to determine the categories for stem and all suffixes. It has two inputs: *Lexical* and *Result_Node. Lexical* is the one of the results and *Result_Node* is node of the result's tree. It does not return any value. It adds the category data in XML nodes.

- Find_Character_In_Subset

This function is used learn whether the input identification of character exists in input identification of subset or not. It has two inputs: two inputs: *Harf_Id* and *Subset_Id.* *Harf_Id* is the identification of character. This character is a part of current feasible pair. *Subset_Id* is the identification of the subset. This subset is in the current rule. The function returns *true* or *false.* If the input identification of character is in the identication of the subset then function returns *true* otherwise returns *false.*

- Lexical_Form_Harf_Karsiligi_Bul

This function is used to determine the character of the input identification. It has one input that is *Harf_Karsiligi.* This input is integer data type that is identification of a character. This function is used to determine the character of this identification. If it finds the character then returns this character otherwise returns null string.

- Find_Parent_Node

This function is used to determine the correct paths of results. It is called when a result is found. It adds the *Analyze* parameter to the parent of correct paths. If path is not correct there is no *Analyze* parameter in *Record.* This is shown in Figure 5.18.

## 5.4 How Analyzer Works

The analyzer computes lexical forms from a surface form recursively. It processes the surface input form by one character at a time. It goes left to right. It tries suitable feasible pairs for each surface character. Analyzer finds suitable feasible pairs according to surface character using this control:

**Step 1**

```
for (int i=0; i<Feasible_Pair_Count; i++)
{
     if ((Feasible_Pair[i][1] = NULL) or
      (Feasible_Pair[i][1] = Surface_Character))
  {
          // step 2
          // step 3
  }
}
```

Analyzer controls a boundary symbol in each step. If a boundary symbol is reached then it controls the morpheme order in the following of operation.

**Step 2**

```
if (result.SubString(result.Length(),1) = Boundary_Character)
{
            Current_Lexical_Form =  Feasible_Pair[i][0];
      my_flag = true;
}
else
{
            Current_Lexical_Form = Current_Lexical_Form +

                                   Feasible_Pair[i][0];
      my_flag = false;
}
```

Analyzer looks dictionary in every step for current lexical form. If current lexical form is not in dictionary then operation fails. In this case analyzer goes back and tries another feasible pair and does same looking operation. If current lexical form is in dictionary then analyzer applies all rules in parallel using the function *Ara_Rule_Content.*

**Step 3**

```
if (Find_In_Dictionary(Current_Lexical_Form,my_flag) = true)
{
     my_feasible_pair = Ara_Rule_Content(Feasible_Pair[i][0],

                                        Feasible_Pair[i][1],

                                           My_State);
     // step 4
}
else
{
     my_flag = false;
  // write reason of fail as situation structure in
               //    Result_Way.xml like Figure 5.18.
}
```

*Ara_Rule_Content* returns a structure and assigns to *my_feasible_pair.* Analyzer tries all possible results in *my_feasible_pair.* If a rule fails with current feasible pair and current state then create a *situation* structure like Figure 5.19.

**Step 4**

```
for (k=0; k<my_feasible_pair.count ; k++)
{
          if ((my_feasible_pair.Set_Feasible_Pair[k][3]  = 0)
or                        (Current_State_Situation = 0))
    {
               my_flag = false;
               // write reason of fail as situation structure
in                        // Result_Way.xml like Figure 5.19.
          }
     My_State[k] = my_feasible_pair.Set_Feasible_Pair[k][2];
}
// step5
```

The last operation is the control of morpheme order. Two rules are designed according to nominal and verbal model of Turkish language to control morpheme order of lexical form.

If boundary symbol is found in step 2 and stem of the surface form is found then analyzer controls the morpheme order rule. If stem of the surface form is a noun or an adjective then nominal rule is applied. If it is a verb then verbal rule is applied.

**Step 5**

```
If ((Current_Lexical_Form.AnsiLastChar() = Boundary_Symbol) and
            (stem = true))
{
      // find morpheme category
      for (x=0; x<Order_Rule_Content_Count; x++)
      {
    if((Order_Rule_Content[x][0] = Morp_Order_Rule.Id) and
           (Order_Rule_Content[x][1] = Morp_Order_Rule.Cur_State)
and
       (Order_Rule_Content[x][2] = Morp_Order_Rule.Morpheme_Cat))
   {
       my_flag_order = true;
       break;
    }
  }
    if ((my_flag_order = false ) || (Order_Rule_Content[x][3] = 0))
    {
            my_flag = false;
              // write reason of fail as situation structure in
              // Result_Way.xml like Figure 5.20.
        }
}
```

After the entire controls, if *my_flag* variable is *true* then this feasible pair is accepted, otherwise not. After all surface characters are processed, analyzer stops and shows the alternation results and the steps of these results as tree structure.

## 5.5 Test Application

There is a test application that uses library in this project. This application is developed by Borland C++ Builder 6.0. There are two main functions in it: *Read Data* and *Analyze Word.* These functions call library's functions.

- *Read Data* function must be run before analyzing words. Figure 5.21 shows the usage of the *read data* function.

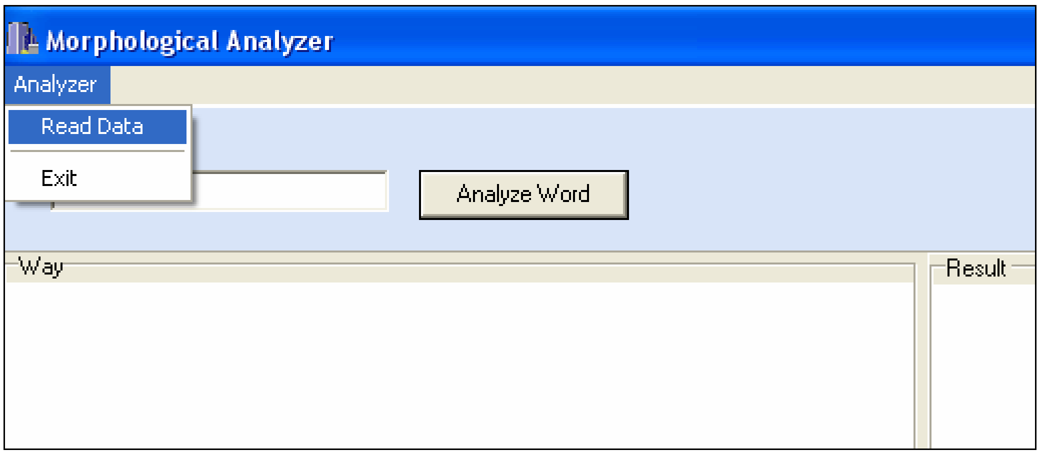


**Figure 5.21 Usage of the "*read data*" function**

- Analyze Word function must be run after reading data process. It can analyze only one word at the same time.

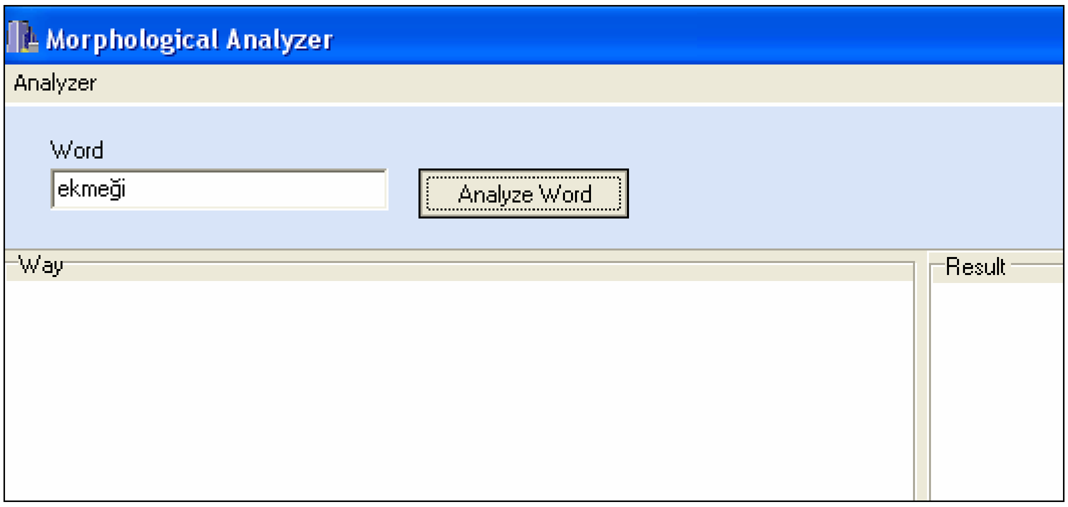
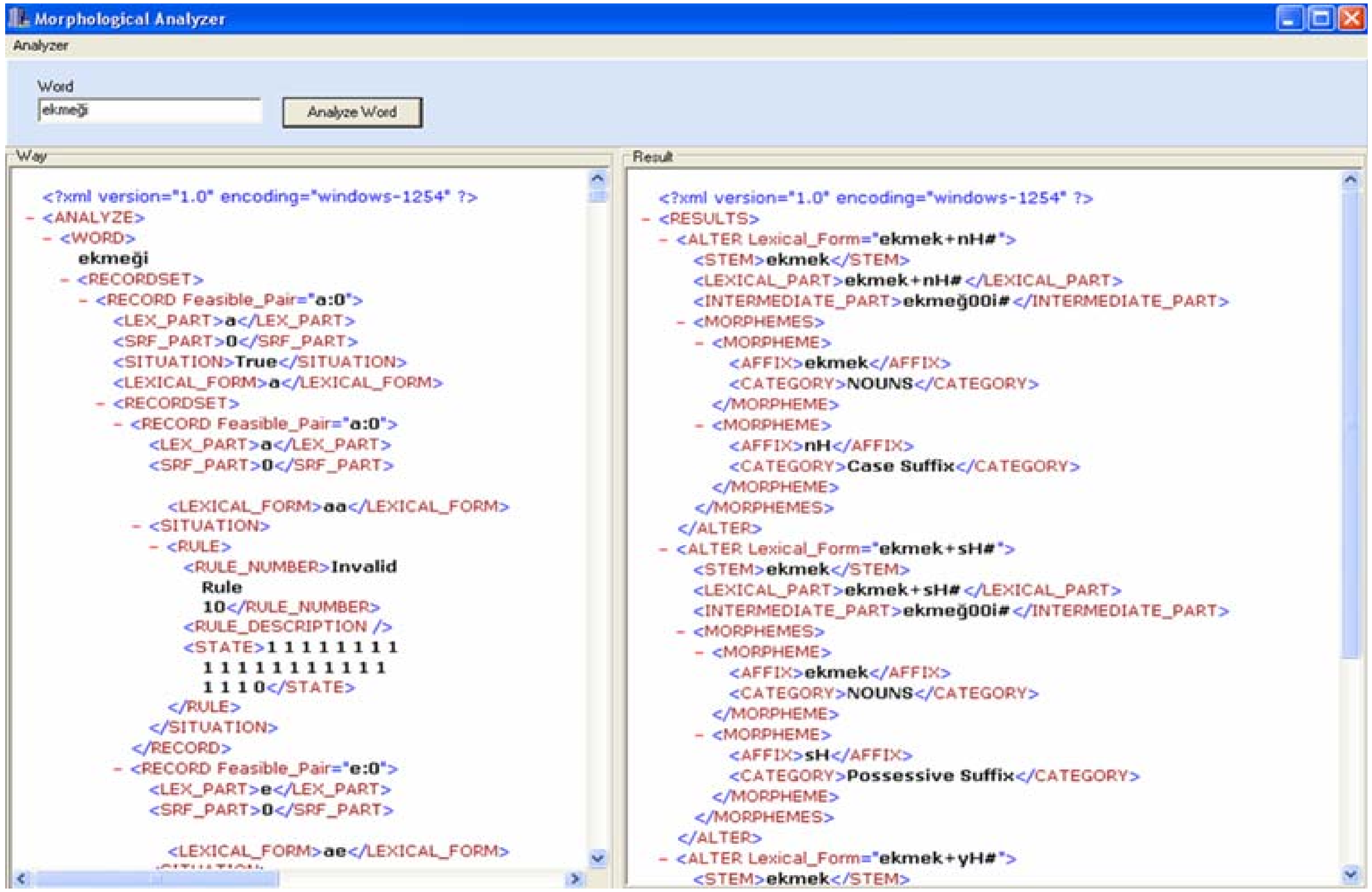


**Figure 5.22 Screen before analyzing operation**

When a word is written text box on the screen and clicked the *Analyze Word* button then test application shows the result of analyzing operation. The screen before analyzing operation is shown in Figure 5.22 and the screen after analyzing operation is shown in Figure 5.23. The steps of the alternative outputs are shown on the left of the screen and the alternative outputs are shown on the right of the screen in Figure 5.23.



```
<?xml version="1.0" encoding="windows-1254" ?>
- <ANALYZE>
  - <WORD>
      ekmeği
    - <RECORDSET>
      - <RECORD Feasible_Pair="a:0">
          <LEX_PART>a</LEX_PART>
          <SRF_PART>0</SRF_PART>
          <SITUATION>True</SITUATION>
          <LEXICAL_FORM>a</LEXICAL_FORM>
        - <RECORDSET>
          - <RECORD Feasible_Pair="a:0">
              <LEX_PART>a</LEX_PART>
              <SRF_PART>0</SRF_PART>

                <LEXICAL_FORM>aa</LEXICAL_FORM>
              - <SITUATION>
                - <RULE>
                    <RULE_NUMBER>Invalid
                      Rule
                      10</RULE_NUMBER>
                    <RULE_DESCRIPTION />
                    <STATE>1 1 1 1 1 1 1 1
                      1 1 1 1 1 1 1 1 1 1 1
                      1 1 1 0</STATE>
                  </RULE>
                </SITUATION>
              </RECORD>
          - <RECORD Feasible_Pair="e:0">
              <LEX_PART>e</LEX_PART>
              <SRF_PART>0</SRF_PART>

                <LEXICAL_FORM>ae</LEXICAL_FORM>
```

```
<?xml version="1.0" encoding="windows-1254" ?>
- <RESULTS>
  - <ALTER Lexical_Form="ekmek+nH#">
      <STEM>ekmek</STEM>
      <LEXICAL_PART>ekmek+nH# </LEXICAL_PART>
      <INTERMEDIATE_PART>ekmeğ00i# </INTERMEDIATE_PART>
    - <MORPHEMES>
      - <MORPHEME>
          <AFFIX>ekmek</AFFIX>
          <CATEGORY>NOUNS</CATEGORY>
        </MORPHEME>
      - <MORPHEME>
          <AFFIX>nH</AFFIX>
          <CATEGORY>Case Suffix</CATEGORY>
        </MORPHEME>
      </MORPHEMES>
    </ALTER>
  - <ALTER Lexical_Form="ekmek+sH#">
      <STEM>ekmek</STEM>
      <LEXICAL_PART>ekmek+sH# </LEXICAL_PART>
      <INTERMEDIATE_PART>ekmeğ00i# </INTERMEDIATE_PART>
    - <MORPHEMES>
      - <MORPHEME>
          <AFFIX>ekmek</AFFIX>
          <CATEGORY>NOUNS</CATEGORY>
        </MORPHEME>
      - <MORPHEME>
          <AFFIX>sH</AFFIX>
          <CATEGORY>Possessive Suffix</CATEGORY>
        </MORPHEME>
      </MORPHEMES>
    </ALTER>
  - <ALTER Lexical_Form="ekmek+yH#">
      <STEM>ekmek</STEM>
```

Figure 5.23 Example results of the analyzing operation

# CHAPTER SIX
# CONCLUSION

The aims of the thesis were to implement a morphological analyzer for Turkish language. This analyzer receives a surface form as input and gives lexical form of this word as output.

Turkish rule definitions and words are stored in XML documents. Some words are not stored as original i.e.: ağ$ız, bur$un. When certain suffixes are affixed then some vowels can be deleted in these roots. These vowels are prefixed with a "$" in the lexical form. These kinds of words are generally designating parts of the human body. This exception must be solve in the future. "ağız" must be used instead of "ağ$ız" in the dictionary. New rules may be written for this purpose.

The application is developed with Borland C++ Builder Version 6.0. The performance of this analyzer can be better. String classes of Borland C++ builder are unsatisfying for high performance. The application can be recoded by using any other programming languages that has stronger string classes to get higher performance.

The analyzer is able to process one word at a time. In the future, the analyzer can be improved to analyze the sentences by using word order rules, paragraphs by using sentence order rules. Such analyzer can be able to analyze the style of an article. By collecting the results of analyzing articles we may analyze the style of an author. This process is called as *stylometry*.

# REFERENCES

Antworth, E.L.(1995). Introduction to PC-KIMMO. North Texas Natural Language Processing Workshop. University of Texas, Arlington, USA.

Antworth, E.L.(1995). Developing the Rules Component. North Texas Natural Language Processing Workshop. University of Texas, Arlington, USA.

Antworth, E. L. (1995). Two-Level Phonology Revisited. North Texas Natural Language Processing Workshop. University of Texas, Arlington, USA.

Antworth, E. L. (1995). Reference Manual. North Texas Natural Language Processing Workshop. University of Texas, Arlington, USA.

Barton, G. E. (1986). Computational Complexity In Two-Level Morphology. In ACL Proceedings, 24th Annual Meeting (Association for Computational Linguistics). Cambridge

Beesley, K.R. & Karttunen, L.(2001). A Short History of Two-Level Morphology. ESSLLI-2001 Special Event titled "Twenty Years of Finite-State Morphology.". Helsinki, Finland.

Duran, A. & Kürkçü, L. (2004). A Drawing Tool For Morphologic Analyzer of Turkish.
Dokuz Eylul University, Izmir, Turkey

Eryiğit, G. & Adalı, E. (2004). An Affix Stripping Morphological Analyzer For Turkish. Artifical Intelligence and Applications. Innsbruck, Austria.

Hankamer, J.. (1986). Finite State Morphology and Left to Right Phonology. Proceedings of the Fifth West Coast Conference on Formal Linguistics, Stanford, CA.

Jurafsky, D & Martin, J.H. Speech and language processing. Prentice Hall, New Jersey 2000

Mengüşoğlu, E. & Deroo, O. (2001). Turkish LVCSR: Database preparation and Language Modeling for an Agglutinative Language. ICASSP. Salt-Lake City

Oflazer, K.(1993). Two-level Description of Turkish Morphology. Association for Computational Linguistics. Morristown, NJ, USA

Oflazer, K. & Solak, A.(1993). Design and Implementation of a Spelling Checker For Turkish. Literary and Linguistic Computing. Oxford Univ., USA

Oflazer, K., Göçmen, E. & Bozşahin, C. (1994) An Outline of Turkish morphology. Technical Report, Middle East Technical University, Ankara, Turkey

Oflazer, K. & Güngördü, Z. (1994). Parsing Turkish Using the Lexical Functional Grammar Formalism. International Conference On Computational Linguisitics.USA

Oflazer, K. & Bozşahin, C. (1994). Turkish Natural Language Processing Initiative: An overview. Proceedings of the Third Turkish Symposium on Artifical Intelligence. Turkey.

Oflazer, K. & Güvenir, H.A. (1994). Using a Corpus For Teaching Turkish Morphology. University of Twente. The Netherlands

Weber, D.J. , Black, H.A & McConnel, S.R.. (1988). AMPLE: a tool for exploring morphology. Summer Institute of Linguistics. Dallas.

WEB_1. (2003). Computational Linguistics and Phonetics Web Site. http://www. http://www.coli.uni-sb.de/ ~kris/nlp-with-prolog/html/node20html(12.12.2003)

WEB_2. (2003). Computational Linguistics and Phonetics Web Site. http://www. http://www.coli.uni-sb.de/ ~kris/nlp-with-prolog/html/node21html(12.12.2003)

WEB_3. (2004). School of Computing - University of Leeds Home Page. http://www.comp.leeds.ac.uk/nti-kbs/ai5/Misc/morphemes.html (30.03.2004)

WEB_4. (2004). Bob's Home Page. http://www.cromwell-intl.com/turkish/nouns.html (16.06.2004)

WEB_5. (2004). Bob's Home Page. http://www.cromwell-intl.com/turkish/orthography.html (16.06.2004)

WEB_6. (2004). Online Free Dictionary Web Site. http://encyclopedia.thefreedictionary.com/Finite%20state%20automata (20.07.2004)

WEB_7. (2004). Department of Computer Science of College of Sciences Web Site. http://www.cs.odu.edu/~toida/nerzic/390teched/regular/fa/nfa-definitons.html (28.07.2004)

# APPENDIX A
# CD CONTENTS

**Root**

**Sources**

**Icons**

FONT02.ICO

DCLAD050.BPI

Drawing2.DDL

MA.bpr

MA.cpp

MA.obj

MA.res

MA.tds

Unit1.obj

UnitFonksiyon.cpp

UnitFonksiyon.obj

UnitFonksiyon.h

UnitMain.cpp

UnitMain.ddp

UnitMain.dfm

UnitMain.h

UnitMain.obj

UnitMorphologicalAnalyzer.cpp

UnitMorphologicalAnalyzer.ddp

UnitMorphologicalAnalyzer.dfm

UnitMorphologicalAnalyzer.h

UnitMorphologicalAnalyzer.obj

UnitMorphologicalAnalyzer.cpp

**Executables**

**Data**

Alphabet.xml

Feasible_Pair.xml

Kind.xml

Order_Rule.xml

Rule.xml

Rule_Content.xml

Subset.xml

Subset_Content.xml

Suffix.xml

Suffix_Category.xml

Word.xml

Word_Category.xml

**Debug**

Result.xml

Result_Way.xml