TURKISH AND TURKMEN MORPHOLOGICAL ANALYZER AND MACHINE TRANSLATION PROGRAM

by

Maxim SHYLOV

A thesis submitted to

the Graduate Institute of Sciences and Engineering

of

Fatih University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

June, 2008
Istanbul, Turkey

APPROVAL PAGE

Student : Maxim SHYLOV
Institute : Institute of Sciences and Engineering
Department : Computer Engineering
Thesis Subject: Turkish and Turkmen Morphological Analyzer and Machine Translation Program
Thesis Date : June 2008

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Bekir KARLIK
Head of Department

This is to certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assist Prof. Atakan KURT
Supervisor

Examining Committee Members

Assist Prof. Atakan KURT …….

Assist Prof. Veli HAKKOYMAZ …….

Prof. Mehmet KARA …….

It is approved that this thesis has been written in compliance with the formatting rules laid down by the Graduate Institute of Sciences and Engineering.

Assist Prof. Nurullah ARSLAN
Director

TURKISH AND TURKMEN MORPHOLOGICAL ANALYZER AND MACHINE TRANSLATION PROGRAM

MAXIM SHYLOV

M. S. Thesis - Computer Engineering
June 2008

Supervisor: Assist Prof. Atakan KURT

ABSTRACT

The machine translation is the one of the fundamental problems in Natural Language Processing. In this thesis a machine translation framework was implemented. The translation framework is mainly based on morphological analysis. Both Turkmen and Turkish morphological parsers and generators have been implemented. An in-depth morphological analysis of Turkmen has been done. The dictionary of word roots and stems for Turkmen and Turkish were created.

Keywords: Machine Translation System, Turkmen Morphological Analyzer, Turkmen Two – level Morphological Rules.

TÜRKÇE VE TÜRKMENCE BİÇİMBİRİMSEL ÇÖZÜMLEME VE MAKİNE ÇEVİRİ PROGRAMI

MAXIM SHYLOV

Yüksek Lisans Tezi – Bilgisayar Mühendisliği
Haziran 2008

Tez Yöneticisi: Yrd.Doç.Dr. Atakan KURT

ÖZ

Makine çevirisi, Doğal Dil İşlemenin temel konularından biridir. Bu tezde, Türkmence ve Türkçe iki taraflı biçimbirimsel makine çeviricisi uygulaması gerçekleştirilmiştir. Başta Türkmence için biçimbirimsel çözümleyeci ve biçimbirimsel üretici uygulama olarak gerçekleştirilmiştir. Daha sonra, Türkçe için biçimbirimsel çözümleyeci ve biçimbirimsel üretici uygulaması gerçekleştirilmiş, Türkçe ve Türkmence için kök sözlükleri oluşturulmuştur.

Anahtar Kelimeler: Makine Çeviri, Türkmence Biçimbilimsel Çözümleyici, Türkmence iki seviyeli Biçimbilimsel Kuralları.

DEDICATION

To my family

ACKNOWLEDGEMENT

I express sincere appreciation to Assist Prof. Atakan KURT and Prof. Mehmet KARA for their guidance and insight throughout the research. Moreover, I would like to thank Prof. Mehmet KARA for helping me in understanding the structure and morphology of Turkmen and Turkish languages.

I would like to thank Gülşah SOYAL for helping preparing dictionaries and all those who was directly or indirectly involved in the project.

I express my thanks and appreciation to my family for their understanding, motivation and patience.

TABLE OF CONTENTS

ABSTRACT iii
ÖZ iv
DEDICATION v
ACKNOWLEDGEMENT vi
TABLE OF CONTENTS vii
LIST OF TABLES ix
LIST OF FIGURES x
LIST OF ABBREVIATIONS xi
CHAPTER 1 INTRODUCTION 1
CHAPTER 2 TURKMEN PHONOLOGY 3
CHAPTER 3 TURKMEN MORPHOLOGY 6
3.1 Two – Level Morphology 6
3.2 Turkmen Ortography 7
3.3 Turkmen Morphotactics 19
CHAPTER 4 TRANSLATION FRAMEWORK 25
4.1 Sentence Tokenizer 25
4.2 Multi Word Tokenizer 26
4.3 Morphological Analysis and Ambiguity 27
4.4 Morphologıc Generator 28
CHAPTER 5 IMPLEMENTATION 30
5.1 Morphologic Analyzer 30
5.2 XML Schema for Morphotactics 30
5.3 XML Schema for Morphotactic Rules 32
5.4 Implementation of Morphotactic Rules 35
5.5 Translation System 38
5.6 Sample Translation 41
CHAPTER 6 CONCLUSIONS 42

REFERENCES 45

LIST OF TABLES

TABLE

2.1: Cyrillic Turkmen alphabet and its transliteration to Latin alphabet. 4
2.2: New Turkmen alphabet and its transliteration to Latin alphabet. 4
2.3: Turkmen vowels. 4
3.1: Two – level Rules 7
3.1: Tenses in Turkish and Turkmen 24
3.2: The tenses in Turkmen without a match in Turkish. 24

LIST OF FIGURES

FIGURES

3.1: A simple finite state automaton for lexical word representation. 7
3.2: The finite state automaton for nominal morphology. 20
3.3: The finite state automaton for verb morphology. 21
3.4: The finite state automaton for verb morphology (cont). 22
4.1: Components of the translation mechanism 25
4.2: A sample parsing for Turkish word gülseydim 28
5.1: The DTD for the finite state automaton of the morphological analyzer 30
5.2: A simple xml representation for finite state automaton. 31
5.3: A graphical representation for finite state automaton. 32
5.4: The DTD for the two – level rules of morphological analyzer. 33
5.5: XML representation for V:0 ⇨ _ + H:@yor rule. 33
5.6: XML representation for V:0 ⇔ $:0_C +:0 @:0V:@ rule. 34
5.7: XML representation for A:a ⇨ V:V_b':'*C*@:0* + :0*_ rule. 35
5.8: The pseudo code for the function used to parse a surface word. 36
5.9: The pseudo code for traverse function. 36
5.10: The pseudo code for NextState function. 37
5.11: The pseudo code for translation function. 38
5.12: The pseudo code for getWordFromPath function. 40
5.13: A sample for translation from Turkmen to Turkish. 41

LIST OF ABBREVIATIONS

ABBREVIATION	
Abl	Ablative
Acc	Accusative
Agr1PS	1st Person Agreement
ADJtoADJ	Adjective to Adjective
ADJtoV	Adjective to Verb
Dat	Dative
Gen	Genitive
Inf	Infinitive
Infer	Inferential
Loc	Locative
Neg	Negative
NegPastIndInf	Negative Past Indefinite Inferential
NtoADJ	Noun to Adjective
NtoN	Noun to Noun
NtoV	Noun to Verb
Plu	Plural
Poss1PS	Possessive 1st Person Single
Poss2PS	Possessive 2nd Person Single
Poss3PS	Possessive 3rd Person Single
PreCon	Present Continuous
PreSim	Present Simple
Rel	Relative
VtoADJ	Verb to Adjective
VtoN	Verb to Noun
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

The machine translation is one of the fundamental problems in the natural language processing. The design and implementation of machine translator is a complex process defined by divergences of languages in terms of their morphological, syntactical and lexical structures (Jurafsky & Martin, 2000). The morphological structures for different languages can vary in terms of morphemes used for word. For example, the English word *cats* has a root *cat* and inflectional morpheme *–s*, but in Turkish word *kediler* has a root *kedi* and an inflectional morpheme *-ler*. The syntactical structure for different languages can vary in term of the way the words are positioned in the sentence. The lexical structures can vary for different languages in terms of morpheme position. For example, in such languages as Russian and English the words can be formed by prepositions and suffixes, but in Turkic languages the words are formed by suffixes.

The languages discussed in this thesis are agglutinative languages which have productive inflectional and derivational suffixes. The inflectional suffixes change the form of words in order to express the grammatical features. For example, *gel(come) + di(came)* or *göl(lake) + ler(lakes)*. The derivational suffixes make new words from existing ones, with a different meaning. For example, *haber(news) + leş(communicate)*.

Koskenniemi introduced the two-level morphology to represent a word in two levels (Koskenniemi, 1983). One of the representations is the word orthographic form and the other is the word lexical form. The lexical form is represented by the word root and morphemes. The morphological rules were used for the word transformation from lexical representation to the orthographic form. Morphological analyzers are used in machine translation. For example, Česilko is the machine translation project for Slavic languages Czech and Slovak (Hajič et al. 2000). Later on, it was extended to deal

with Lithuanian (Hajič et al. 2003), Polish and Lower Serbian (Dvořák et al. 2006). Another project is interNOSTRUM, which was developed for translating between Catalan and Spanish (Canals-Marote et al. 2000). A machine translation system for Portuguese – Spanish was implemented in the same manner (Garrido-Alenda et al. 2003). However, the number of studies on machine translation between Turkic languages is limited. One of the studies is related to the translation from Turkish to Crimean Tatar, which is able to generate ambiguous translations with a limited dictionary (Altıntaş & Çiçekli, 2001).

The translation machine for Turkmen and Turkish languages is a current topic (Tantug et al, 2007). This study (Tantug et al, 2007) is based on the morphological analysis and morphological translation (Tantug et al, 2006). The morphological translation was modified by introducing new modules to perform syntactical analysis. However, it seems that a morphological analyzer is not fully describing the structures of the Turkmen language. Therefore those structures were analyzed in this thesis and morphotactic rules were redefined to cover those structures as well.

The morphological analyzer for Turkmen and machine translation from Turkmen to Turkish and Turkish to Turkmen is studied in this thesis. However, the framework for translation system between agglutinative languages were designed and implemented. The framework consists of morphological parser and generator. The rules and morphological parser for Turkish language were implemented and designed on the basis of Oflazer research (Oflazer, 1994). Therefore, the aspects of the design of the Turkish morphological parser are not described.

The rest of the thesis is as follows. The second chapter provides information about Turkmen language and its similarities and differences with Turkish language. In addition, the ways of dealing with differences are described. The morphological parser and generator for Turkmen language are described in the chapter three. Chapter four describes word by word translation system. The fifth chapter describes important technologies and algorithms used in the system. Chapter six discusses the conclusion and future work.

CHAPTER 2

TURKMEN PHONOLOGY

The Turkmen is the official language of Turkmenistan. It is also spoken by many people living outside Turkmenistan. The Turkmen language is one of the Turkic languages, belonging to the Oghuz group. An Arabic script was used for the Turkmen language as the first writing system since 18 century, although very little was written in it. The Unified Turkish Latin Alphabet (UTLA), based on the Latin alphabet, was introduced in 1928. It was very similar to the Latin alphabet used in Turkey. In 1940, the Cyrillic script for Turkmen replaced the UTLA. Finally, in 1995, the "Täze Elipbiÿi" or New Alphabet was formally introduced by the president and officially came into use in 1996. This alphabet and Cyrillic Turkmen alphabet are used today. Both alphabets are given in the Tables 2.1 and 2.2.

Like the rest of the Turkic languages, Turkmen is agglutinative, meaning that most grammatical functions are pointed out by attaching suffixes to the stems of words. One of the most notable features of the Turkmen language is the vowel harmony. All vowels can be classified as front vowels or back vowels. In the Turkmen language, if there is a back vowel in the first syllable of the word, back vowels are also used in the following syllables. The same can be said for the front vowels. However, some words, which were taken from other languages, don’t obey vowel harmony. The front, and back vowels are given in the Table 2.3.

Table 2.1: Cyrillic Turkmen alphabet and its transliteration to Latin alphabet.

Cyrillic	Latin	Cyrillic	Latin	Cyrillic	Latin
А а	A a	Л л	L l	Х х	H h
Б б	B b	М м	M m	Ц ц	Ts ts
В в	V v	Н н	N n	Ч ч	Ç ç
Г г	G g(Ğ ğ)	Ң ң	Ň ň	Ш ш	Ş ş
Д д	D d	О о	O o	Щ щ	Sç sç
Е е	Ye ye	Ө ө	Ö ö	Ъ ъ	Separation
Ё ё	Yo yo	П п	P p	Ы ы	I ı
Ж ж	J j	Р р	R r	Ь ь	Subtilization
Җ җ	C c	С с	S s	Э э	E e
З з	Z z	Т т	T t	Ә ә	Ä ä
И и	İ i	У у	U u	Ю ю	Yu yu
Й й	Y y	Ү ү	Ü ü	Я я	Ya ya
К к	K k	Ф ф	F f		

Table 2.2: New Turkmen alphabet and its transliteration to Latin alphabet.

Turkmen Latin	Latin	Turkmen Latin	Latin	Turkmen Latin	Latin
A a	A a	J j	C c	R r	R r
B b	B b	Ž ž	J j	S s	S s
Ç ç	Ç ç	K k	K k	Ş ş	Ş ş
D d	Ç ç	L l	L l	T t	T t
E e	E e	M m	M m	U u	U u
Ä ä	Ä ä	N n	N n	Ü ü	Ü ü
F f	F f	Ň ň	Ň ň	W w	V v
G g	G g(Ğ ğ)	O o	O o	Y y	I ı
H h	H h	Ö ö	Ö ö	Ý ý	Y y
I i	İ i	P p	P p	Z z	Z z

Table 2.3: Turkmen vowels.

	Unrounded		Round	
	Wide	**Narrow**	**Wide**	**Narrow**
Back	a	y	o	u
Front	e, ä	i	ö	ü

The other characteristic of Turkmen language is the vowel harmony related to the round vowels in the word. According to the rules of harmony, the wide round vowels *o* and *ö* can appear only once in the first syllable and cannot appear in the syllables following it. Once, the narrow round vowels *u* and *ü* appears in the syllable, it can appear only in the next closed syllable. The syllable is said to be closed if the syllable ends with a consonant letter. Otherwise, it is said to be open. If a syllable following the

syllable containing narrow round *u* or *ü* is open, then the vowel it contains can be one of the narrow unrounded vowels *y* or *i*. However, the closed syllable following the syllable containing a narrow round *u* or *ü* can contain a long vowel such as *y* and *i*. The long vowels are vowels with a long sound and are not disclosed within the orthography of Turkmen language. Some words in Turkmen language with suffixes *vuk/vük* don't obey the vowel harmony rules (Kara, 2005).

An interesting property of Turkmen language is the way the words are being pronounced. Against the vowel harmony, the wide round *o* and *ö* can appear after first syllable in the speech. However, this topic is out of the scope of the thesis work, therefore it will not be discussed.

CHAPTER 3

TURKMEN MORPHOLOGY

The two – level morphologic analyzer is the one of the most common approaches used in machine translation. The following subsections give general information about morphological analyzer and two – level rules employed in this translation project.

3.1 TWO – LEVEL MORPHOLOGY

The two – level morphology was firstly introduced by Koskenniemi in 1983 (Koskenniemi, 1983). The two-level morphology defines two different levels of word representation. The first is called lexical level and it represents a word as a list of concatenated morphemes. The lexical level expresses the grammatical features of the formed word. The second level is called surface and it represents word's orthographic realization. The mapping between these levels is performed by means of so called two - level rules. Each language can have a number of rules.

The rule is defined by *(lexical: surface)* correspondence pair followed by an operator and the immediate left and right context. There are four operators used in the rules. The rules using those operators and their meaning are given in the Table 3.1.

Any word can be represented by the finite state automaton in two - level morphology. Let's assume that we have a Turkish lexicon containing a set of words *çocuk (child)*, *çocuklar(children)*, *çocukların(your children)*, *çocuğun(your child)*. Then the words in the dictionary are the surface level. A simple finite state automaton, which can be used to obtain the lexical level for the dictionary is shown in Figure 3.1. The node named *Q1* is the initial state. The node named *Q3* is an end state. When the *çocuk* word is passed to the *Q1* node, the lexical representations that can be generated are

çocuk + *lAr*, *çocuk* + *lAr* + *Hn*, and *çocuk* + *Hn*. Each of the state can define a lexical representation of words with two – level morphology. The lexical – surface pair can be determined in the following way. The word root which can be a candidate for the surface word representation is located in the dictionary of roots. After that, the suffixes are appended one after another to the located word root. The word root with appended to it suffixes forms lexical word representation. The two – level morphological rules are then applied to the lexical word. The operation continues until the surface word form and a word generated by the application of two – level rules to the lexical form are identical.

Table 3.1: Two – level Rules

Rules	**Meaning of the Rule**
a:b ⇨ lc_rc	Lexical a is realized as surface b, only when it has lc to the left and rc to the right
a:b ⇦ lc_rc	Lexical a always realized as surface b when it has lc to the left and rc to the right.
a:b ⇔ lc_rc	Lexical a always and only realized as surface b when it has lc to the left and rc to the right
a:b /⇦ lc_rc	Lexical a is never realized as surface b when it has lc to the left and rc to the right

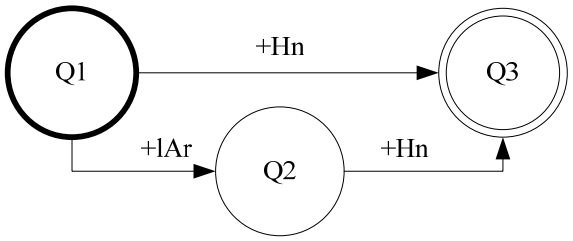

Figure 3.1: A simple finite state automaton for lexical word representation.

The finite state automaton can be designed to represent all possible legal combinations of the morphemes. Those legal combinations form lexical representations of the words. The surface form of the word can be obtained by application of two – level morphologic rules to the lexical form of the word.

3.2 TURKMEN ORTOGRAPHY

The Turkmen language is still using two alphabets. The Cyrillic and a new Latin Turkmen alphabet defined in 1996. The rules defined for the Turkmen language are

based on the Latin Turkmen alphabet. However, the switch to the Cyrillic alphabet can be achieved by mapping each Latin letter to the corresponding Cyrillic letter. The Latin Turkmen alphabet consists of 30 letters. There are 9 vowels: a, e, ä, i, o, ö, u, ü, y, and 21 consonants: b, ç, d, f, g, h, j, ž, k, l, m, n, ň, p, r, s, ş, t, w, ý, z. The sets defined below are used in two – level rules:

1. Lexical Consonants: C = { b, ç, d, f, g, h, j, ž, k, l, m, n, ň, p, r, s, ş, t, w, ý, z, Ç, G, S, P, T }
2. Lexical Vowels: V = { a, e, ä, i, o, ö, u, ü, y, H, A, Ä, Ö, Ü }
3. Back Vowels: V_b = { a, o, u, y }
4. Front Vowels: V_f = { e, ä, i, ö, ü }
5. Front Rounded Vowels: V_{fr} = { ö, ü }
6. Back Rounded Vowels: V_{br} = { o, u }
7. Lexical A = { a, e }
8. Lexical Ä = { a, ä }
9. Lexical H = { i, u, ü, y, ä }
10. Lexical Ö = { o, ö }
11. Lexical Ü = { u, ü }
12. Lexical consonant G = { k, g }
13. Lexical consonant P = { p, b }
14. Lexical consonant T = { t, d }
15. Lexical consonants which can disappear on the surface under certain conditions D = {s, n}
16. Lexical consonants which are always realized on the surface C_r = { Ç, S }

The two-level rules for the Turkmen language are defined as follows. Those rules are based on recently published Turkmen grammar (Kara, 2005). Examples are given after the rules.

1. k:g ⇨ _ +:0 (@:0)V

   The last k of the word becomes *g* whenever a morpheme starting with a vowel is affixed.

   Lexical: kirjimek + nH dirty + Acc
   Surface: kirjimeg00i kirjimegi
   Lexical: tovuk + Hm chicken + Poss1PS

Surface: tovug0ym tovugym

2. A:a ⇔ y:0 +:0_l

The rule deals with a special case for *Al* derivational morpheme. It states that a lexical *A* becomes an *a* on the surface if and only if the word ends with *y*. *y* letter is deleted upon the affixation.

Lexical: dogry + Al right + NtoV (to straighten up)

Surface: dogr00al dogral

3. A:e ⇔ i:0 +:0 _l

Similar to the rule 2, lexical *A* becomes *e* on the surface whenever a word ends with *i*. *i* is deleted upon the affixation.

Lexical: egri + Al bent + ADJtoV (to bend)

Surface: egr00el egrel

4. k:0 ⇨ _ + [jA:@k|jH:@k]

The last *k* of the word is deleted whenever *jAk* or *jHk* morpheme is affixed to a word.

Lexical: tovuk + jAk chicken + NtoN (small chicken)

Surface: tovu00jak tovujak

Lexical: dövük + jAk broken ADJtoADJ

Surface: dövü00jek dövüjek

Lexical: kiçik + jHk small + ADJtoADJ (smaller)

Surface: kiçi00jik kiçijik

Lexical: ýumşak + jHk soft + ADJtoADJ (softer)

Surface: ýumşa00jyk ýumşajyk

5. Ö:o ⇨ V_b +:0 C_C

The lexical *Ö* becomes *o* on the surface, when a word ends with a back vowel.

Lexical: damak + SÖv glutton + NtoADJ (gluttonous)

Surface: damak0sov damaksov

6. Ö:ö ⇨ V_f +:0 C_C

The lexical *Ö* is realized as *o* on the surface, when a word ends with a front vowel.

Lexical: çäge + SÖv sand + NtoADJ (sandy)

Surface: çäge0söv çägesöv

7. V:0 ⇔ $:0_C +:0 @:0V:@

Sometimes a vowel in lexical level will be deleted on the surface due to the ellipsis phenomenon. The phenomenon occurs when a vowel becomes unstressed. This vowel is identified by the *$* symbol in the lexical representation of the word root. This rule deals with such vowels. The rule states that a vowel following *$* symbol is deleted when the morpheme being affixed to the word starts with a vowel. The *$* symbol is deleted as well. This rule also appears in the Turkish two – level morphological rules (Oflazer, 1994).

Lexical: ag$yz + Hm	mouth + Pos1PS
Surface: ag00z0ym	agzym
Lexical: ag$yz + dA	mouth + Loc
Surface: ag0yz0da	agyzda
Lexical: or$un + Ä	place + Dat
Surface: or00n0a	orna
Lexical: ýyg$yn + Hň	group + Poss2PS
Surface: ýyg00n0yň	ýygnyň

8. 0:m ⇨ V_ +:0 SH:@rA:@

This rule deals with the case when a new consonant is added on the surface. The phenomenon is specific to *SHrA* morpheme. The word ending with vowel gets m consonant between the word and morpheme on the surface, whenever *SHrA* morpheme is affixed.

Lexical: çaý + SHrA	tea + NtoV (wish to drink tea)
Surface: çaý0syra	çaýsyra
Lexical: eýe + SHrA	owner + NtoV (behave as owner)
Surface: eýem0sire	eyemsire
Lexical: ulu + SHrA	grand + ADJtoV (do the grand)
Surface: ulum0syra	ulumsyra

9. G:k ⇨ [p|ç|t|k|s|ş|a|ä] +:0 _är

This rule deals with lexical *G* in the morpheme *Gär*. It states that a surface *k* occurs whenever a word ends with one of the letters in the option list.

Lexical: çögap + Gär	responsibility + NtoADJ (responsible)
Surface: çogap0kär	çogapkär
Lexical: günä + Gär	sin + NtoN (sinner)
Surface: günä0kär	günäkär

10. G:g ⇨ [!(p|ç|t|k|s|ş|a|ä)] +:0 _är

    Similar to Rule 9, the lexical *G* is dealt with in this rule. *G* becomes a surface *g* whenever a word ends with one of the letters not in the option list.

Lexical: küýze + Gär	earthenware pot + NtoN (potter)
Surface: küýze0gär	küýzegär
Lexical: umydy + Gär	hope + NtoADJ (hopeful)
Surface: umydy0gär	umydygär

11. G:g ⇨ [p|ç|t|k|s|ş] +:0 _(H:@)r

    Similar to previous rule, the lexical *G* in the *GHr* morpheme is dealt with. The surface *g* appears whenever a word ends with one of the letters in the option list.

Lexical: hynç + GHr	sob + NtoV (to sob)
Surface: hynç0gyr	hynçgyr
Lexical: pyş + GHr	sneeze + NtoV (to sneeze)
Surface: pyş0gyr	pyşgyr

12. G:k ⇨ [!(p|ç|t|k|s|ş)] +:0 _(H:@)r

    Like in Rule 11, the surface *k* appears whenever a word ends with one of the letters not in the option list.

Lexical: haý + GHr	exclamation + NtoV (to exclaim)
Surface: haý0kyr	haýkyr
Lexical: heň + GHr	cry + NtoV (to cry)
Surface: heň0kir	heňkir

13. A:0 ⇔ V +:0 [_lg(A:@)|_v(A:@)ç|_vul]

    The lexical *A* disappears on the surface only and only when one of *AlgA*, *AvAç* or *Avul* morphemes are affixed and the word ends with a vowel.

Lexical: oka + AlgA	to read + VtoN (reading hall)
Surface: oka00lga	okalga
Lexical: germe + AvAç	to stretch + VtoN (stretcher)
Surface: germe00vaç	germevaç
Lexical: çapa + Avul	distribute + VtoN (courrier)
Surface: çapa00vul	çapavul

14. Ü:0 ⇨ V +:0 _

    This rule deals with lexical *Ü* which disappears on the surface when a word ends with a vowel which is not deleted on the surface.

Lexical: alda + Üv	to cheat + VtoN (cheat)
Surface: alda00v	aldav
Lexical: derňe + Üv	to inspect + VtoN (inspection)
Surface: derňe00v	derňev

15. Ü:ü ⇨ V_fC +:0 C:@_

The rule states that a lexical *Ü* is realized as *ü* on the surface when the last vowel of a word ending with a consonant is front vowel and the suffix's first lexical vowel is *Ü*.

Lexical: düz + Üv	to be right VtoN (right)
Surface: düz0üv	düzüv
Lexical: biç + Üv	to give a shape + VtoN (shape)
Surface: biç0üv	biçüv
Lexical: gübürde + vÜk	to make noise +VtoADJ (noisy person)
Surface: gübürde0vük	gübürdevük
Lexical: jürle + vÜk	to whistle + VtoN (whistle)
Surface: jürlü0vük	jürlüvük

16. U:u ⇨ V_bC +:0 C:@ _

The rule states that a lexical *Ü* is realized as u on the surface when the last vowel of a word ending with a consonant is a back vowel and the suffix's first lexical vowel is *Ü*.

Lexical: haşyrda + vÜk	to rustle + VtoN (rustling noise maker)
Surface: haşyrda0vuk	haşyrdavuk
Lexical: ýaldyra + vÜk	to shine + VtoADJ (shiny)
Surface: ýaldyra0vuk	ýaldyravuk
Lexical: çap + Üv	run + VtoN (race)
Surface: çap0uv	çapuv

17. e:ä ⇔ m_k +:0 Ä:@

In Turkmen when an infinitive form of verb takes the suffix of dative case, the last vowel of the word changes on the surface in case it is *e*. This rule defines this particular case.

Lexical: bermek + Ä	to give + Inf + Dat
Surface: bermäg0e	bermäge
Lexical: gelmek + Ä	to come + Inf + Dat
Surface: gelmäg0e	gelmäge

18. [n|s]:0 ⇨ C +:0 _

The lexical *n* or *s* is deleted on the surface, whenever a word ends with a consonant and the suffix being affixed starts with an *n* or *s* consonant.

Lexical: gijeler + sH	night + Plu + Poss3PS
Surface: gijeler00i	gijeleri
Lexical: gijeler + nHň	night + Plu + Gen
Surface: gijeler00iň	gijeleriň

19. H:ä ⇔ e:0 +:0 @:0_

If and only if the word ends with *e* and the suffix starts with a lexical vowel *H*, the last vowel of a word is deleted and the *H* lexical vowel realized as *ä* on the surface.

Lexical: düýe + Hm	camel + Poss1PS
Surface: düý00äm	düýäm
Surface: gözle + Hp dur	to observe + PreSim
Lexical: gözl00äp	gözläp

20. H:0 ⇨ V(:!e) +:0 _

This rule states that the first letter of the suffix is deleted whenever the word it is affixed ends with a vowel other from *e*.

Lexical: oba + Hm	village + Poss1PS
Surface: oba00m	obam
Lexical: avçy + Hň	hunter + Poss2PS
Surface: avçy00ň	avçyň

21. e:ä ⇔ _ +:0 [n(H:@)|n(H:@)ň|k(H:@)]

This rule deals with a word ending with the vowel *e* and the suffix affixed is one of the suffixes in the option list. When a match is located the last vowel of the word is realized as *ä* on the surface.

Lexical: düýe + nHň	camel + Gen
Surface: düýä0niň	düýäniň
Lexical: gije + nH	night + Acc
Surface: gijä0ni	gijäni
Lexical: çölde + kH	desert + Loc + Rel
Surface: çöldä0ki	çöldäki
Lexical: deňizde + kH	sea + Loc + Rel
Surface: deňizdä0ki	deňizdäki

22. Ä:ä ⇔ V_f:0 +:0 @:0_

This rule deals with the words ending with one of the front vowels and taking suffixes starting with lexical *Ä*. In this case, the last vowel of the word is deleted on the surface and the suffix's lexical *Ä* is realized as *ä* on the surface.

Lexical: bergi + nÄ	debt + Dat
Surface: berg00ä	bergä
Lexical: düýe + Ä	camel + Dat
Surface: düý00ä	düýä
Lexical: ele + Än dälmiş	to eliminate + NegPastIndInf
Surface: el00än dälmiş	elän dälmiş

23. Ä:a ⇔ V_b:0 +:0 @:0_

This rule is the same as the previous one except it deals with words ending with back vowels.

Lexical: alada + nÄ	care + Dat
Surface: alad00a	alada
Lexical: tuty + Ä	curtain + Dat
Surface: tut00a	tuta

24. Ä:e ⇔ V_fC +:0 @:0_

This rule deals with the suffixes affixed to the words ending with consonant and with a front vowel in the last syllable. According to the rule, a lexical *Ä* becomes a surface *e* when it is located at the beginning of a suffix.

Lexical: gül + Än dälmiş	to lugh + NegPastIndInf
Surface: gül0en dälmiş	gülen dälmiş
Lexical: gel + Än eken	to come + NegPastIndInf
Surface: gel0en eken	gelen eken

25. Ä:a ⇔ V_bC+:0 @:0_

This rule is similar to Rule 24. It deals with words with a back vowel in the last syllable.

Lexical: ýap + Än dälmiş	to do + NegPastIndInf
Surface: ýap0an dälmiş	ýapan dälmiş
Lexical: kol + Ä	arm + Dat
Surface: kol0a	kola

26. ç:j ⇨ V_ +:0 @:0 V

In Turkmen morphology, when the *ç* character occurs between two vowels, it realized as j on the surface. This rule deals with words ending with *ç* and a suffix affixed to the word starts with vowel.

Lexical: bozguç + Hň eraser + Poss2PS

Surface: bozguj0yň bozgujyň

Lexical: yangyç + nH fuel oil + Acc

Surface: yangyj00y yangyjy

27. 0:ý ⇔ V_ +:0 H:@ş

This rule introduces the addition of a new letter when it doesn't occur in the lexical level. It states that a new character *ý* should be added on the surface whenever a word ends with a vowel and the suffix is *Hş*.

Lexical: gora + Hş protect + VtoN (protection)

Surface: gora0ýyş goraýyş

Lexical: sözle + Hş to speak + VtoN (speech)

Surface: sözle0ýiş sözleýiş

28. {P,T}:{b,d} ⇨ _ +:0 V

The letter *p* of some Turkmen words ending with *p*, changes to *b* when a suffix starting with a vowel is affixed. The letter *t* of some Turkmen words ending with *t*, changes to *d* when a suffix starting with a vowel is affixed. The last letters of the words which can have such conversion were specified by letters *P* and *T* in the lexicon. This rule deals with those letters. According to the rule the lexical P and T are realized as surface b and d whenever an affixed suffix starts with a vowel.

Lexical: gaP + Hm container + Poss1PS

Surface: gab0ym gab0ym

Lexical: aT + Hm name + Poss1PS

Surface: ad0ym adym

29. {P,T}:{p,t} ⇨ _ +:0 C

This rule realizes lexical P and T as surface p and t whenever the suffix affixed to the word starts with a consonant.

Lexical: gap + ndAn container + Abl

Surface: gap00dan gapdan

Lexical: taT + dA taste + Loc

Surface: tat0da tatda

In Turkmen language the vowel harmony is different for open and closed syllables affixed to a word consisting of one syllable. The rules from 30 to 33 deal with such cases.

30. H:y ⇨ C*V_{br}C +:0 C:@_

This rule states that *H* vowel of the suffix with an open syllable is realized as *y* only when the word consists of one syllable with a back round vowel.

Lexical: ýol + nH	road + Acc
Surface: ýol00y	ýoly
Lexical: guş + sH	bird + Poss3PP
Surface: guş00y	guşy

31. H: i ⇨ C*V_{fr}C +:0 C:@_

This rule states that *H* vowel of the suffix with an open syllable is realized as *i* only when the word consists of one syllable with a front round vowel.

Lexical: göz + sH	eye + Poss3PS
Surface: göz00i	gözi
Lexical: kül + nH	ash + Acc
Surface: kül00i	küli

32. H: u ⇨ C*V_{br}C +:0 C*_C

This rule deals with the words consisting of one syllable with a back round vowel and the suffixes with one closed syllable. When this condition is satisfied, the lexical *H* of the suffix is realized as *u* on the surface.

Lexical: kol + Hm	arm + Poss1PS
Surface: kol0um	kolum
Lexical: kub + nHň	cube + Gen
Surface: kub00uň	kubuň

33. H: ü ⇨ C*V_{fr}C +:0 C*_ C

This rule states that a lexical *H* is realized as *ü* on the surface, when a suffix with one closed syllable is affixed to a word with one syllable with a front round vowel.

Lexical: öý + nHň	house + Gen
Surface: öý0üň	öýüň
Lexical: küý + Hm	consideration + Poss1PS
Surface: küý0üm	küýüm

34. y:u ⇔ C*V_{br}C*_ +:0 C

This rule deals with the case when a suffix starting with a consonant is affixed to a word consisting of two syllables such that the first syllable contains a back round vowel and the last syllable is open. In this case, if the word ends with lexical *y*, it is realized as *u* on the surface.

Lexical: koly + ndA	arm + Poss3PS + Loc
Surface: kolu0nda	kolunda
Lexical: ruhy + nH	soul + Poss3PS + Acc
Surface: ruhu0ny	ruhuny

35. i:ü ⇔ C*V_{fr}C*_ +:0 C

This rule deals with the case when a suffix starting with a consonant is affixed to a word consisting of two syllables such that the first syllable contains a front round vowel and the last syllable is open. In this case, if the word ends with lexical *i*, it is realized as *ü* on the surface.

Lexical: gözi + nH	eye + Poss3PS + Acc
Surface: gözü0ni	gözüni
Lexical: küýi + ndA	consideration + Poss3PS + Loc
Surface: küýü0nde	küýünde

36. H:y ⇨ C*V_bC*V_bC +:0 C*_C

This rule deals with a word consisting of more than one syllable and the last two syllables contain a back vowel. Moreover, the last syllable of the word should be closed. When a suffix affixed to such a word, its lexical *H* is realized as *y* on the surface.

Lexical: gonşum + nH	neighbour + Poss1PS + Acc
Surface: gonşum00y	gonşumy
Lexical: darak + Hm	comb + Poss1PS
Surface: darag0ym	daragym

37. H:i ⇨ C*V_fC*V_fC +:0 C*_C

This rule deals with a word consisting of more than one syllable and the last two syllables contain a front vowel. Moreover, the last syllable of the word should be closed. When a suffix affixed to such a word, its lexical *H* is realized as *i* on the surface.

Lexical: enek + Hň	cow + Poss2PS
Surface: enegiň	enegiň
Lexical: mekdeb + Hm	school + Poss1PS

Surface: mekdebim mekdebim

38. A:a ⇨ C*V_bC* +:0 C*_C

This rule states that the lexical *A* of a suffix is realized as surface *a* when the vowel preceding it is a back vowel.

Lexical: kolun + dA arm + Poss2PS + Loc

Surface: kolunda kolunda

Lexical: oba + dAn village + Abl

Surface: obadan obadan

39. A:e ⇨ C*V_fC* +:0 C*_C

This rule states that the lexical *A* of a suffix is realized as surface *e* when the vowel preceding it is a front vowel.

Lexical: gözün + dA eye + Poss2PS + Loc

Surface: gözün0de gözünde

Lexical: öý + dA house + Loc

Surface: öý0de öýde

40. Ä:a ⇨ C*V_bC +:0 C_C

The lexical *Ä* is realized on the surface as *a* when the last vowel of a word is a back vowel.

Lexical: gara + ýÄr to look + PreSim

Surface: gara0ýar garaýar

Lexical: dur + mÄn eken to stand + NegPastIndInf

Surface: dur0man eken durman eken

41. Ä:ä ⇨ C*V_fC* +:0 C_C

The lexical *Ä* is realized on the surface as *ä* when the last vowel of a word is a front vowel.

Lexical: bil + ýÄ to know + PreSim + Agr3PS

Surface: bil0ýä bilýä

Lexical: söý + mÄn eken to like + NegPastIndInf

Surface: söý0män eken söýmän eken

3.3 TURKMEN MORPHOTACTICS

The finite state automaton defines the lexical representation of the word as discussed previously. In addition, it specifies the order in which the morphemes can be affixed to a word. So, the structure of the word can be restricted. The finite state automaton for the nominal words is presented in the Figure 3.2. The finite state automaton for the verbs is defined in similar way and presented in Figure 3.3 and 3.4. In Turkmen language adverb, adjectives and pronouns take all inflectional suffixes the nouns take therefore the finite state automaton defined for the nouns can be reused for adverbs, adjectives, and pronouns.

The finite state automaton's entry point so called initial state is defined by the thick-bordered rectangle shown in the Figure 3.2. Internal states have regular normal borders. They can be a solution for the generation of the word's lexical representation. The states that have double line border are called end states and they also can be treated as a solution. The nodes with dotted-border are not treated as a solution; even thought they can appear in the sequence of the morphemes. The reason for defining such node is the specific aspects of Turkmen language. For example, *öýleriň* and *öýleň* have the same meaning (your villages), but the word *öýle* is meaningless in Turkmen.

The finite state automaton was defined in such a way that it could not only translate the word from surface to lexical level, but determine the structure and the ordering of the morphemes. Therefore the surface word with incorrect order of the morphemes will be rejected. For example, the possessive suffixes are not acceptable after the accusative case in Turkish; as a result the Turkish word with incorrect order *kitabıyım* will be rejected by the morphological analyzer. On the other hand, the correctly ordered word *kitabımı* will be accepted. The examples provided below will clarify the main ides of word construction.

The nominal for word *düýeleriňdäki* can be constructed as follows.

düýe +lAr +Hň +ndA +kH

düýe +ler +iň +0dä +ki

things on your camels

The states passed to generate the word are

1. Noun (root)

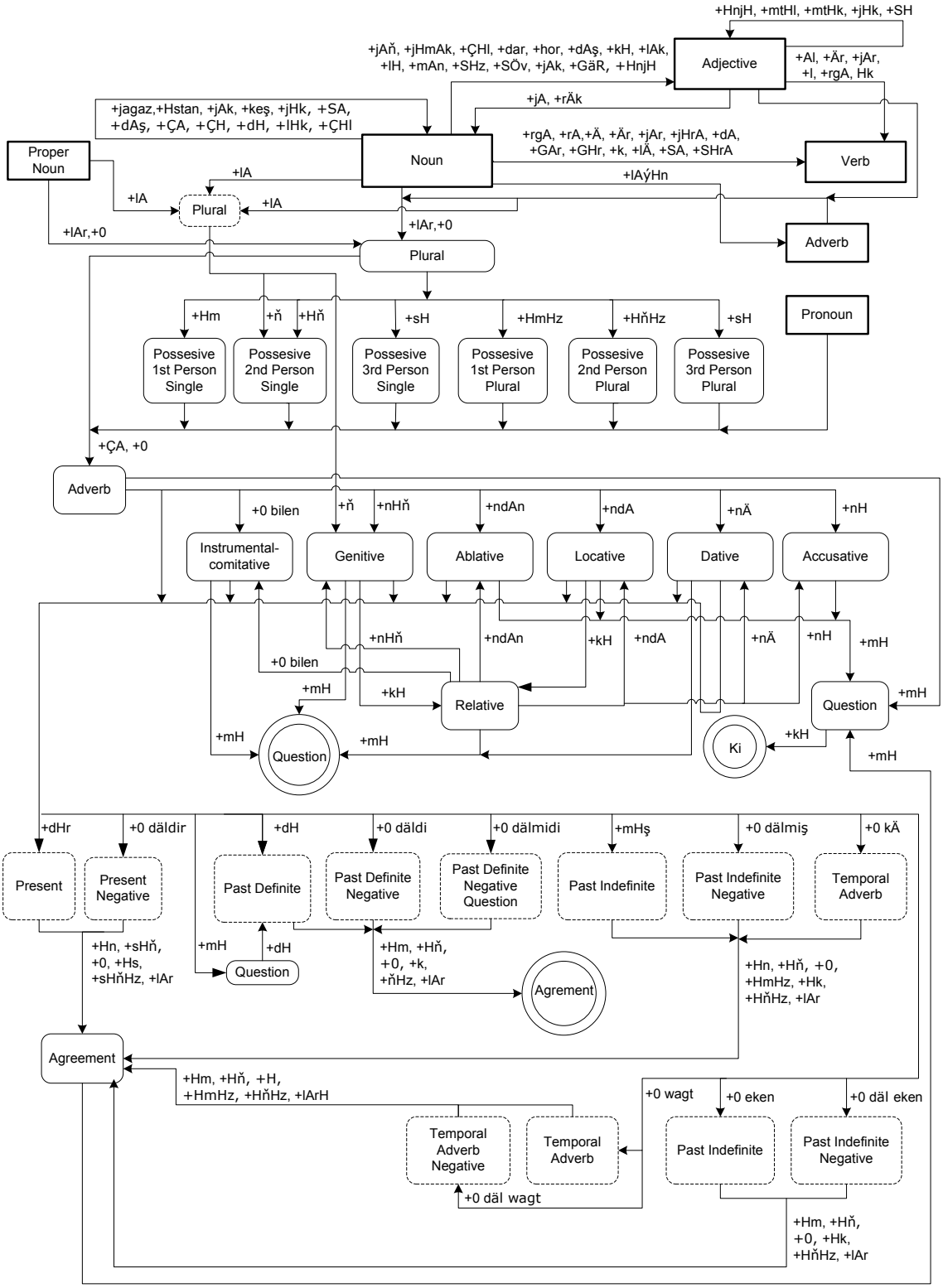

Figure 3.2: The finite state automaton for nominal morphology.

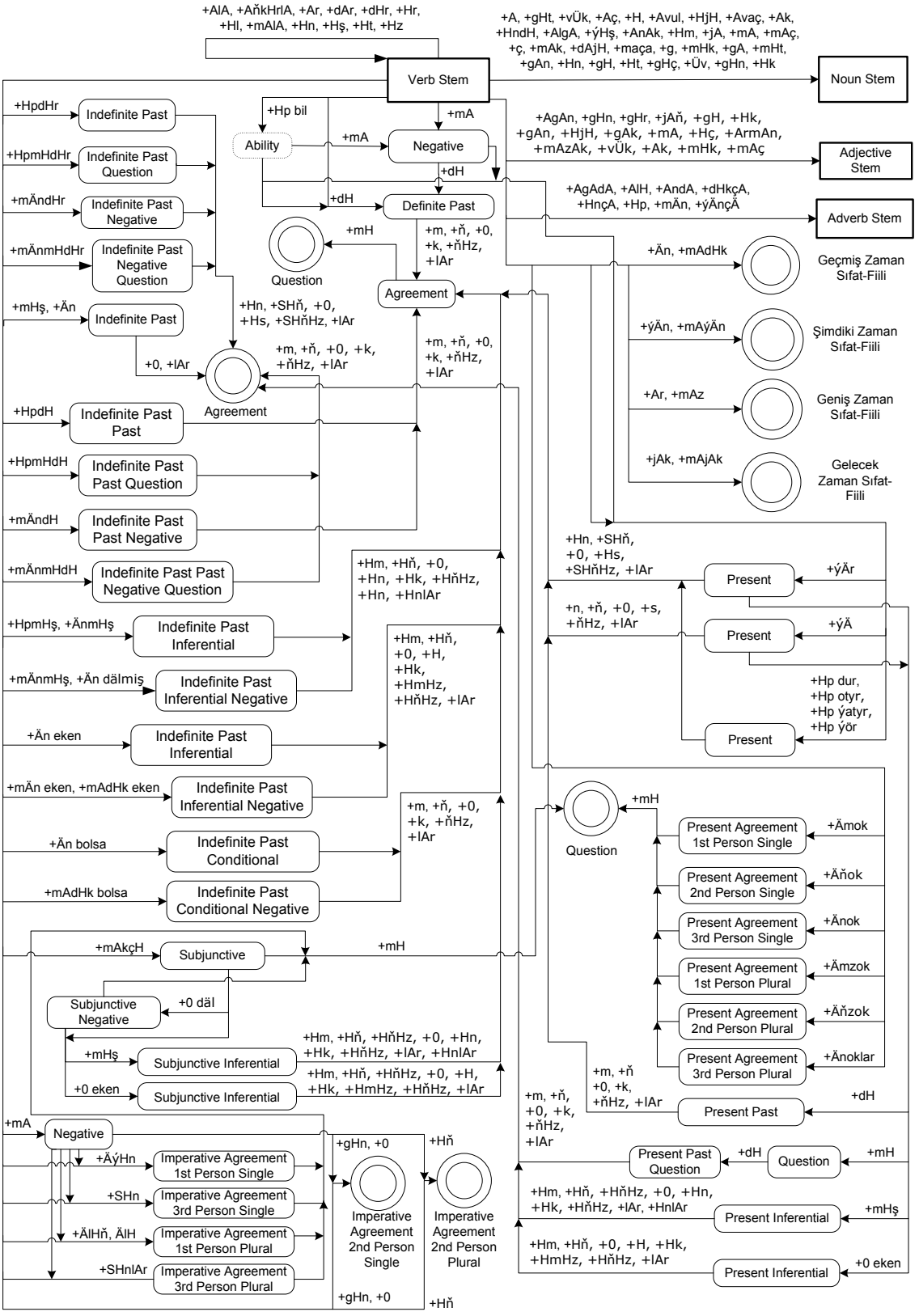

Figure 3.3: The finite state automaton for verb morphology.

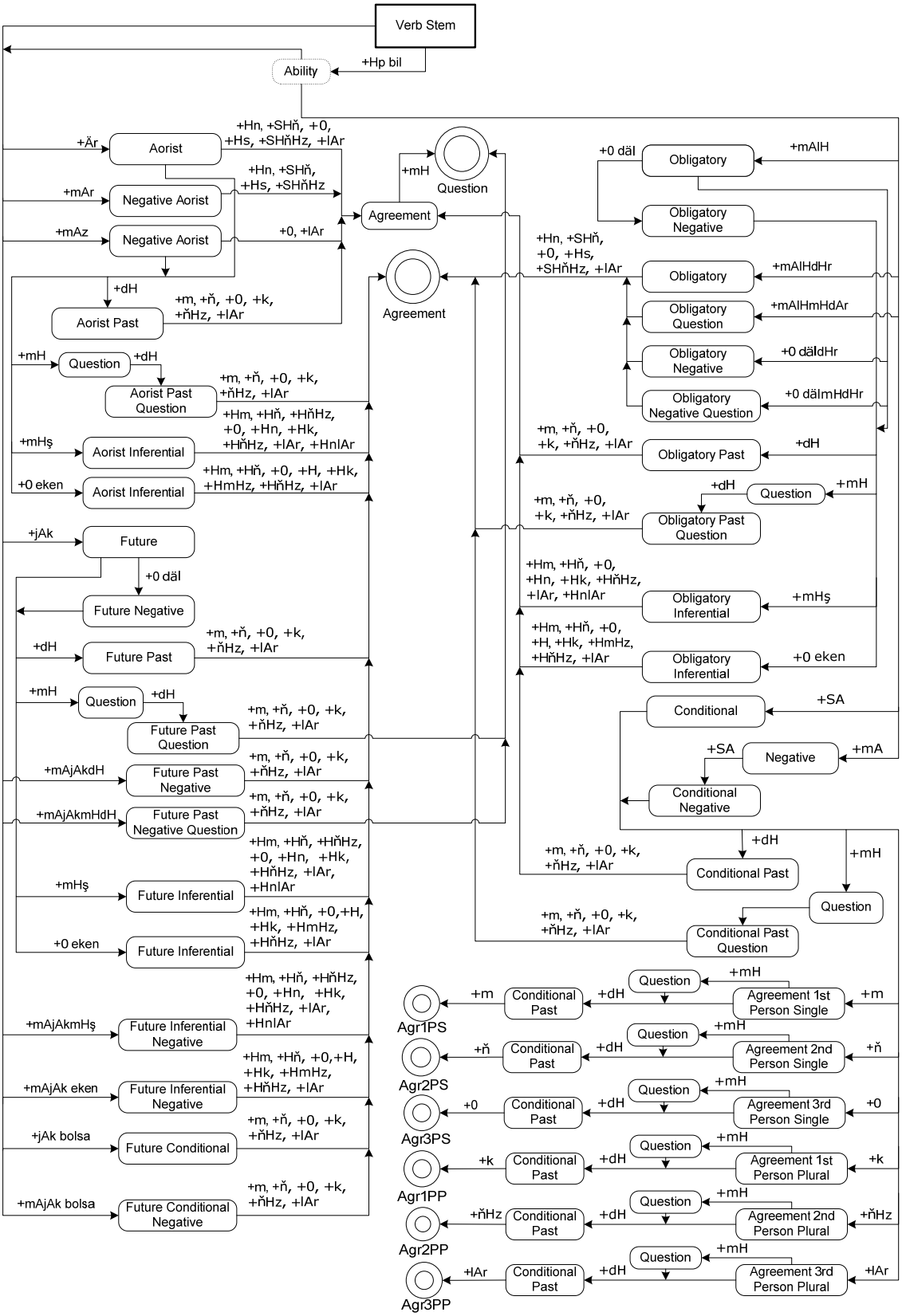

Figure 3.4: The finite state automaton for verb morphology (cont).

2. Plural with +lAr
3. Possessive 2nd Person Single with +Hň
4. Locative with +ndA
5. Relative with +kH

The verb for word *görmeýär ekenim* can be constructed as follows

gör	+mA	+ýÄr	+0 eken	+Hm
gör	+me	+ ýär	+ eken	+im
see	+Neg	+PreCon	+Infer	+Agr1PS

The states passed to generate the word are

1. Verb (root)
2. Negative with +mA
3. Present with + ýÄr
4. Present Inferential with +0 eken
5. Agreement with +Hm

The Turkish and Turkmen languages are both agglutinative languages as stated before, but they have developed independently despite they are from the same family. The Turkmen language has some differences concerning the affixation of suffixes. The person agreement suffixes are affixed to the obligatory and intentional forms of verb in Turkish, but the pronoun and obligatory and intentional forms of verb combination is used to obtain the same result in Turkmen. In addition, the order of some suffixes affixed to word in Turkish can be different for Turkmen. One such example is the question suffix. Moreover, the copulatives used in Turkish have no corresponding copulatives in the Turkmen. The *ki* copulative of Turkish is one of such example. And finally, there are the tenses in Turkish which have no corresponding tense in Turkmen. The list of tenses for Turkish and Turkmen are listed in the Table 2.4. The +/- sign specifies the presence/absence of tense.

When a tense which has no equivalent in the other language, then the meaningfully most similar tense was chosen. These tenses are given in Table 2.5. However, some of the tenses are presented by lexical suffixes, which are decided after asking the native speakers of the Turkmen language.

Table 3.1: Tenses in Turkish and Turkmen

Tense	**Turkish**	**Turkmen**
Present Simple	+	+
Past Simple (Past Indefinite)	+	+
Aorist Simple	+	+
Future Indefinite	+	+
Past Definite	+	+
Conditional Form	+	+
Obligatory Form	+	+
Imperative Form	+	+
Subjunctive Form	+	+
Past Indefinite Past	+	+
Past Definite Past	+	-
Present Past	+	+
Future Past	+	+
Aorist Past	+	+
Conditional Past	+	+
Past Obligatory Form	+	+
Subjunctive Past Form	+	-
Past Indefinite Inferential	+	+
Present Inferential	+	+
Future Inferential	+	+
Aorist Inferential	+	+
Conditional Inferential	+	-
Inferential Obligatory Form	+	+
Subjunctive Inferential Form	+	+
Past Indefinite Conditional	+	+
Past Definite Conditional	+	-
Present Conditional	+	-
Future Conditional	+	+
Aorist Conditional	+	+
Obligatory Conditional Form	+	-

Table 3.2: The tenses in Turkmen without a match in Turkish.

Verb State for Turkish	**Corresponding Verb State for Turkmen**
Past Definite Past	Past Indefinite Past
Subjunctive Past Form	Conditional Past
Past Definite Conditional	Conditional Past
Present Conditional	+ ýÄ + Än bolsa
Obligatory Conditional Form	+ mAlH + 0 bolsa

The implementation of two-level rules and the finite state automaton of the morphological analyzer discussed in the next chapter.

CHAPTER 4

TRANSLATION FRAMEWORK

The translation framework consists of sentence tokenizer, multi word tokenizer, source language morphological analyzer and target language generator components. The interaction between those components is presented on the Figure 4.1. The framework was designed for agglutinative languages.

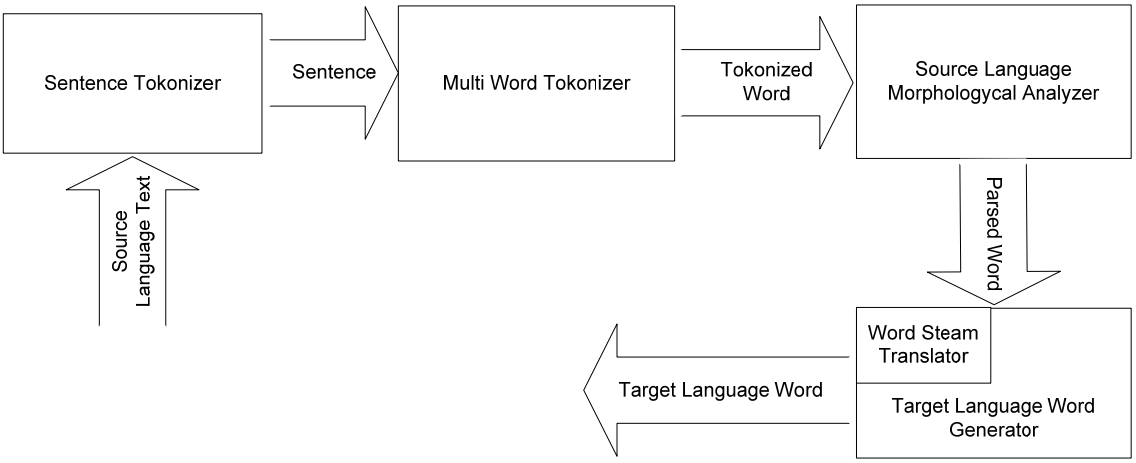

Figure 4.1: Components of the translation mechanism

4.1 SENTENCE TOKENIZER

The sentence tokenizer is responsible to split a given text into paragraphs and sentences. The paragraph is identified by the new line character. After paragraphs are identified by the system, each paragraph is splitted into sentences. The sentence is identified using punctuation marks, such as ., *;*. *!*, *:* and *?*.

4.2 MULTI WORD TOKENIZER

The Turkic languages are closely related. However some of the words in one language are represented by a phrase in the other. Especially, it concerns the tenses. For example, the word in Past Indefinite Inferential for the Turkmen has the surface representation *gelen eken*, but in Turkish it is *gelmişmiş*. The multiword tokenizer is responsible for splitting a given text into the phrases or words which correspond to the word in the target language. To achieve this goal, the multiword - tokenizer splits each sentence into words. During the division process the words and punctuation marks are stored in a list. So, each sentence is represented by list of words and punctuation marks. After that, the multiword tokenizer groups the words. The grouping process is achieved by taking each word in the list and joining all the words in the list before a punctuation mark. Then the generated group is passed to the morphological analyzer. If the morphological analyzer fails to parse a given group of words, the number of words in the word group is decreased by one. This process continues till the number of words in a group is one. For example:

Sentence: Howa sergindi, ýektaý dony hem çykaraýasyň gelip durdy.

List of words and punctuation marks: ‘Howa’, ‘sergindi’, ‘,’, ‘ýektaý’, ‘dony’, ‘hem’, ‘çykaraýasyň’, ‘gelip’, ‘durdy’, ‘.’.

Group of words for word ‘Howa’: <Howa sergindi>,<Howa>.

Group of words for word ‘sergindi’:<sergindi>.

Group of words for word ‘ýektaý’: <ýektaý dony hem çykaraýasyň gelip durdy>, <ýektaý dony hem çykaraýasyň gelip>, <ýektaý dony hem>, <ýektaý dony>, <ýektaý>.

Group of words for word ‘dony’: <dony hem çykaraýasyň gelip durdy>, <dony hem çykaraýasyň gelip>, <dony hem çykaraýasyň>, <dony hem>, <dony>.

Group of words for word ‘hem’: <hem çykaraýasyň gelip durdy>, <hem çykaraýasyň gelip>, <hem çykaraýasyň>, <hem>.

Group of words for word ‘çykaraýasyň’: <çykaraýasyň gelip durdy>, <çykaraýasyň gelip>, <çykaraýasyň>.

Group of words for word ‘gelip’: <gelip durdy>, <gelip>.

Group of words for word ‘durdy’: <durdy>.

The main disadvantage of such approach is the possibility, that two words that should be treated separately could come together and match the morpheme from the list. To handle such situation, the words that satisfied the match should be also passed to the

morphological analyzer separately. In any case, the words that are not satisfying morpheme ordering and structure will be rejected by the morphological analyzer.

4.3 MORPHOLOGICAL ANALYSIS AND AMBIGUITY

The source language morphological analyzer component is responsible for taking a word or phrase in its surface representation and outputting the word or phrase in its lexical representation. In addition, list of mapId's of the nodes visited during the obtaining of lexical representation for the word is provided to the word generator. The mapId is a number used for mapping between source node and destination node. For example, if value of mapId for source node with name noun is one then the value for destination node with name noun should be one.

The root dictionary is a must for the morphological analyzer. Two lookup dictionaries were used, one for Turkish, and the other for the Turkmen. Every row of the dictionary stores *word root*, *word lexical root*, *the type of the word root*, and *the language the word belongs to*. Using these dictionaries the morphological analyzer determines the root for the analyzing word. Each root from the dictionary is compared with a word being analyzed. If that word starts with any root from the dictionary then the root is said to be a possible solution for the analysis. After that the finite state automaton is traversed. During that traversing the suffixes used during movement from one node to another are affixed to the candidate root. After each movement the generated lexical form is converted to surface form and the word wished to be analyzed is checked to start with word surface form. If the word doesn't start with surface form of the word the node is ignored. The traversing continues until all nodes are traversed or all generated surface forms are mismatch the beginning of the word being analyzed.

A sample result of Turkish *gülseydim* word parsing is given in Figure 4.2. Map path is the list of mapIds. The maximal stem is the stem obtained by affixing derivational suffixes.

1st Person Single: gül + SA + yDH + m
Map Path: 3~81~106~94
Suffixes used for lexical level
SA
yDH
m
Root: gül
Root's Part of Speech: verb
Maximal Stem: gül
Maximal Stem's Part of Speech: verb

Figure 4.2: A sample parsing for Turkish word gülseydim

The morphologic analyzer has capability of returning the parsed word, the list of mapIds, a list of suffixes in lexical form, the root and its part of speech, and maximal stem and its part of speech. The maximal stem is the same as the root in this situation. However, the maximal stem and its part of speech can change when the derivational suffixes are used.

In Turkish language some words have more than one meaning. For example, *yüz* (face, swim [imperative form], one hundred) can be treated as a nominal noun or a verb. As a result, there is not a single lexical form for the word. In addition, some orders of suffixes are realized equally on the surface, causing more than one lexical representation of the word. Therefore, the morphological analyzer can produce more than one lexical representation for a word. For example, *hastalık* (illness) is represented as *hasta* + *lHk* (+Noun) and *hasta* + *lH* + *Hk* (+Noun+Verb) on lexical level. However, some of the representations which have no corresponding mapping are eliminated by the target language word generator.

4.4 MORPHOLOGIC GENERATOR

Morphologic generator generates the surface form of a word from lexical form. The generator have internal module used to translate the stem of source language to the stem of target language. However, not the root of the word, but maximal stem is translated, since it is difficult to create relation between derivational suffixes.

The stem dictionary represented as a table which has the following fields: id, word stem, word stem's speech part, and word language. The id field is a primary key for the

table. For example, the word *gül(laugh)* is stored as *1, gül, verb, Turkish*. Another, row from the same table is *2, gül(laugh), verb, Turkmen*. The translations between those words are stored in second table which has the following fields: *id*, *source id*, *destination id*, and *order of the stem*. The source and destination ids are referenced to the stem dictionary table's id. Using this table one source stem can be related to more than one destination stem. The order of stem field specifies primary and secondary orders of translation stems.

The morphologic generator uses map ids referring to states of the morphological analyzer's finite state automaton. When a word is passed from one state to another state the map ids are combined in a list. The system then takes this list and uses it to generate a word by passing the list of map ids and a translated stem to the morphological analyzer of the target language. However, Turkmen and Turkish languages have differences in terms of lexical structures. Therefore, the generated list of map ids in Turkmen is different from that in Turkish. To deal with such differences a map translator was introduced. It takes a map ids list of source language and returns the map ids list or map ids lists of the destination language. Therefore the translation of a given word can have more than one result. For example, the word *gülseydim* (if I laughed) in Turkish, have two translations in Turkmen: *gülsedim* and *gülsemdim*.

The current translation mechanism is designed to take a sentence and translate words or group of words, the syntactical structure is not taken into account. Due to this some translated sentences may be meaningless. In addition, a word can have many translations, since there is no way to resolve meaning of the word. An example for this is Turkmen word *abadan*, which can be translated as *durgun*(calm), *geçimli*(easygoing), or *babadan*(from father), *dededen*(from grandfather). First two words are translations for stem *abadan*, but last two are translations for stem *aba* taking ablative inflection suffix.

CHAPTER 5

IMPLEMENTATION

5.1 MORPHOLOGIC ANALYZER

The software for morphologic analyzer and translator was developed in Java. The web page interface was implemented with Java Server Page technology. The morphotactics and morphotactic rules for Turkmen and Turkish were defined in xml files. This approach was used because of flexibility of XML. Later on, same xml schema can be used for other languages without making any changes to the software.

5.2 XML SCHEMA FOR MORPHOTACTICS

The DTD for morphologic analyzer's finite state machine is given in Figure 5.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT finiteStateMachine (states)>
<!ELEMENT states (state+)>
<!ELEMENT state (name, action*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT action (suffix+)>
<!ELEMENT suffix (#PCDATA)>
<!ATTLIST state type (initialState | middleState | endState) #REQUIRED stateId ID
        #REQUIRED rootType (noun | verb | adverb | adjective | pronoun| proper noun)
        #IMPLIED mapId CDATA #REQUIRED isNotSolution (Y) #IMPLIED>
<!ATTLIST finiteStateMachine language (Turkish | Turkmen | Kyrgyz | Uzbek | Kazakh |
Azerbaijani) #REQUIRED>
<!ATTLIST action resultStateId IDREF #REQUIRED>
```

Figure 5.1: The DTD for the finite state automaton of the morphological analyzer

The DTD starts with the element called finiteStateMachine, which can include only one element called states. In addition, the finiteStateMachine element has an attribute called language, which specifies what language the morphological analyzer

was defined for. The states element can have one or more elements called state. Each of the *state* elements has a *name* element and zero or more *action* elements. The *action* element specifies the states to which the owner of the *action* leads to. The *state* element has a set of attributes. The first one is the *type*, which defines the kind of the state. The values it can take are *initialState*, *middleState*, or *endState*. The *type* attribute is required. The next attribute is *stateId*, which defines a unique state id. Another attribute for the *state* element is called *rootType*, which is optional and should appear within the element whenever the *type* attribute's value is *initialState*. The values for the *rootType* attribute are *noun*, *verb*, *adverb*, and *adjective*. The *state* element has *mapId* required attribute, which is used for the mapping between two different finite state automata of morphological analyzers. The last attribute that the *state* element can have is the *isNotSolution* attribute, which have the only one possible value *Y* meaning *yes*. That attribute has to appear at the states generating a lexical word with no meaning on the surface. The *action* element can have one or more *suffix* elements. The attribute it takes is *resultStateId*. This attribute's value specifies the *state action* leads to. The value for the *suffix* element is the lexical representation of the suffix morpheme. An example for the xml structure provided by the DTD is shown in Figure 5.2. The graphical representation for the example is given in Figure 5.3.

```
<finiteStateMachine language="Turkish">
    <states>
          <state type="initialState" stateId="s-1" rootType="noun" mapId="1">
            <name> Nominal Root</name>
            <action resultStateId="s-2">
                <suffix>lAr</suffix>
            </action>
        </state>
        <state type="middleState" stateId="s-2" mapId="2">
            <name>Plural</name>
            <action resultStateId="s-1">
                <suffix>lH</suffix>
                <suffix>SHz</suffix>
            </action>
        </state>
    </states>
</finiteStateMachine>
```

Figure 5.2: A simple xml representation for finite state automaton.

The finite state automaton presented in the Figure 5.3 consists of two states. The name for the first state is *Nominal Root*, and the name for the second state is *Plural*. Each of the states has one action; the first state's action leads to the second state, and the

second state's action leads to the first state. In addition, the first state is an initial state and the root types it accepts are nouns, therefore it is an entry point for the finite state machine. So the lexical representations of the word that can be generated are *word root + lAr*, *word root + lAr + lH*, and *word root + lAr + SHz*.

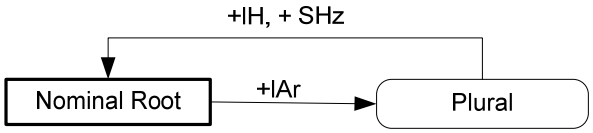

Figure 5.3: A graphical representation for finite state automaton.

5.3 XML SCHEMA FOR MORPHOTACTIC RULES

The DTD for two – level rules given in Figure 5.4 is a bit more complicated than that of morphotactics.

The DTD starts with element named *rules* which can have element called *letters* followed by one or more elements called *rule*. The *rules* element has an attribute called *language*, which is required and specifies the language the rules were defined for. The *letters* element consists of one or more elements called *consonant*, which is followed by one or more elements called *vowel*. The *vowel* element can be followed by the elements called *conversion*. The *consonant* element defines the consonant letters available for the language. The *vowel* element defines the vowel letters for the language. The *conversion* element is defining conversion for the letters. It has two attributes called *from* and *to*, and specifying that value of *from* attribute is substituted with value of *to* attribute. There are cases when the last letter of the word is changing when a suffix is affixed. The conversion element is introduced to deal with such situations. An example for such case is Turkmen word *düýä* (camal + Dat). The root for this word is *düýe*. The root of first word is hidden since the roots last letter is dropped when a dative suffix taken. So using, the values of conversion element the last letter of the word can be replaced. In this case, *ä* with *e*, as a result the words will match.

The rest of DTD is explained using the examples bellow, since it is better to understand from examples than from writing.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT rules (letters, rule+)>
<!ATTLIST rules language (Turkish | Turkmen | Kyrgyz | Uzbek | Kazakh | Azerbaijani)
      #REQUIRED>
<!ELEMENT letters (consonant+, vowel+, conversion*)>
<!ELEMENT consonant EMPTY>
<!ATTLIST consonant char CDATA #REQUIRED>
<!ELEMENT vowel EMPTY>
<!ATTLIST vowel char CDATA #REQUIRED>
<!ELEMENT conversion EMPTY>
<!ATTLIST conversion from CDATA #REQUIRED>
<!ATTLIST conversion to CDATA #REQUIRED>
 <!ELEMENT rule (((condition, statement+) | (condition, statement+, otherwise+) | statement+) |
      for-each)>
 <!ATTLIST rule order CDATA #REQUIRED>
<!ELEMENT condition ((expressionGroup) | (expression) | (expressionGroup?, expression+,
      operator))>
<!ELEMENT for-each (((condition, statement+) | (condition, statement+, otherwise+) |
      statement+)+)>
<!ATTLIST for-each select (suffixVowelLetter) #REQUIRED>
<!ELEMENT expressionGroup (expressionGroup?, expression+, operator?)>
<!ELEMENT expression EMPTY>
<!ATTLIST expression
operator (equals | not_equals | endsWith | contains | startsWith | not_endsWith | preLastCharEquals
      | lastVowelEquals | previousVowelEquals | firstVowelEquals | vowelCountEquals |
      preLastVowelEquals | isFirst | vowelCountNotEquals | secondCharIs) #REQUIRED
      operandOne (word | suffix | suffixLetter | wordLetter) #REQUIRED
operandTwo CDATA #REQUIRED>
<!ELEMENT operator (#PCDATA)>
<!ELEMENT statement EMPTY>
<!ATTLIST statement operator (replace | deleteLastChar | deleteFirstChar | addToEnd )
      #REQUIRED oldPattern CDATA #IMPLIED newPattern CDATA #IMPLIED flag
      (firstOccurrence | lastOccurrence | All) #IMPLIED operandOne (word | suffix | suffixLetter
      | wordLetter) #REQUIRED      operandTwo CDATA #IMPLIED>
<!ELEMENT otherwise (condition?, statement+)>
```

Figure 5.4: The DTD for the two – level rules of morphological analyzer.

An XML for orthographic rule *V:0* ⇨ _ + *H:@yor* (Oflazer, 1994) is shown in Figure 5.5.

```
<rule order="3">
    <condition>
        <expression operator="equals" operandOne="suffix" operandTwo="Hyor"/>
        <expression operator="endsWith" operandOne="word" operandTwo="vowel"/>
        <operator>AND</operator>
    </condition>
    <statement operator="deleteLastChar" operandOne="word"/>
</rule>
```

Figure 5.5: XML representation for V:0 ⇨ _ + H:@yor rule.

From *rule*'s attribute *order* it can be clear that the rule should be applied in third order. There is a condition which checks that the suffix is *Hyor* and word ends with a

vowel. If the conditions a satisfied then the last letter of the word is deleted before affixation.

An XML for orthographic rule *V:0 ⇔ $:0_C +:0 @:0V:@* (Oflazer, 1994) is shown in Figure 5.6.

```
<rule order="4">
    <condition>
        <expression operator="contains" operandOne="word" operandTwo="$"/>
        <expression operator="startsWith" operandOne="suffix" operandTwo="vowel"/>
        <operator>AND</operator>
    </condition>
    <statement operator="replace" operandOne="word" oldPattern="$." newPattern=""
        flag="lastOccurrence"/>
    <otherwise>
        <condition>
            <expression operator="contains" operandOne="word" operandTwo="$"/>
            <expression operator="startsWith" operandOne="suffix" operandTwo="consonant"/>
            <operator>AND</operator>
        </condition>
        <statement operator="replace" operandOne="word" oldPattern="$" newPattern=""
            flag="lastOccurrence"/>
    </otherwise>
</rule>
```

Figure 5.6: XML representation for V:0 ⇔ $:0_C +:0 @:0V:@ rule.

The XML representation states that *$.* characters are deleted when the word have a $ symbol and suffix starts with a vowel. Dot means any character. However, if the conditions are not satisfied the word is checked for containing *$* symbol and suffix is checked for starting with a consonant. If the condition is satisfied the *$* symbol is deleted.

The next example is related to the application of a rule repeatedly. The reason for such need is appearance of suffixes which consist of two or more syllables. An example for such suffix is *mAktA*. Therefore, each vowel should be processed separately. The XML representation for rule *A:a ⇨ V:V_b':'*C*@:0* + :0*_* (Oflazer, 1994) is shown in Figure 5.7

The vowels in suffix are checked repeatedly by introducing *for-each* element and providing a *suffixVowelLetter* value to its *select* attribute. So, each vowel is processed one by one. Firstly, the previous, according to processed one, vowel is checked using first condition. If the checked vowel is the first in the suffix then the second condition is used to check last vowel of the word.

```
<rule order="17">
    <for-each select="suffixVowelLetter">
        <condition>
            <expressionGroup>
                <expression operator="previousVowelEquals" operandOne="suffixLetter"
                    operandTwo="a"/>
                <expression operator="previousVowelEquals" operandOne="suffixLetter"
                    operandTwo="ı"/>
                <expression operator="previousVowelEquals" operandOne="suffixLetter"
                    operandTwo="o"/>
                <expression operator="previousVowelEquals" operandOne="suffixLetter"
                    operandTwo="u"/>
                <operator>OR</operator>
            </expressionGroup>
            <expression operator="equals" operandOne="suffixLetter" operandTwo="A"/>
            <operator>AND</operator>
        </condition>
        <statement operator="replace" operandOne="suffixLetter" oldPattern="A" newPattern="a"/>
        <condition>
            <expressionGroup>
                <expression operator="lastVowelEquals" operandOne="word" operandTwo="a"/>
                <expression operator="lastVowelEquals" operandOne="word" operandTwo="ı"/>
                <expression operator="lastVowelEquals" operandOne="word" operandTwo="o"/>
                <expression operator="lastVowelEquals" operandOne="word" operandTwo="u"/>
                <operator>OR</operator>
            </expressionGroup>
            <expression operator="equals" operandOne="suffixLetter" operandTwo="A"/>
            <operator>AND</operator>
        </condition>
        <statement operator="replace" operandOne="suffixLetter" oldPattern="A" newPattern="a"/>
    </for-each>
</rule>
```

Figure 5.7: XML representation for A:a ⇨ V:V_b':'*C*@:0* + :0*_ rule.

5.4 IMPLEMENTATION OF MORPHOTACTIC RULES

The finite state automaton for the morphological analyzer is represented as a directed graph. Each edge of the graph has a list of suffixes. The vertexes of the graph are stored in the list. Therefore they can be visited one by one. In addition, they can be visited using the edges. When the graph is traversed the lexical form of the word is generated. The generation is achieved by passing the word root into the initial state. Starting from that state each of the suffixes located on the edge is added to the word. For example, if the edge has two suffixes, two lexical representations will be generated for each word from the outgoing vertex. Those words are passed to the vertex, to which the edge leads. The traversing of the tree is done using the breath first search algorithm. The pseudo code for the generation of the lexical word representation is given in Figure 5.8.

```
Input: wordToParse
Output: list of word lexical representations
function parseWord
    Dictionary←get all roots from dictionary
    for i ← 0 to sizeof(Dictionary) do
        j ← 0
        if wordToParse starts with Dictionary[i].lexicalRoot then
            SolutionCanditates[j].lexicalRoot ← Dictionary[i].lexicalRoot
            SolutionCanditates[j].rootType ← Dictionary[i].rootType
            j ← j + 1
    for i ← 0 to sizeof(SolutionCanditates) do
        for j ← 0 to sizeof(NodeList) do
            if NodeList[j].stateType = 'initialState' AND NodeList [j].rootType =
                SolutionCanditates [i].rootType then
                traverse(NodeList [j], wordToParse)
    return solution
```

Figure 5.8: The pseudo code for the function used to parse a surface word.

Dictionary is the list containing the structure, which has two elements: the lexical root and a lexical root's part of speech. *SolutionCandidates* is the list of the candidates to be a solution. *NodeList* is the list of the nodes of the finite state automaton for the morphological analyzer. The *solution* is a global variable and filled by the traverse function. The function actually takes a word and tries to match it with the one in the root dictionary. The root from the root dictionary is said to match the comparison word if the comparison word begins with the root from the dictionary. All located words are candidates for the solution. After that, the initial node, whose speech part the same with the speech part of candidate root, and the candidate root are passed to the *traverse* function. The pseudo code for traverse function is given in Figure 5.9.

```
Input: state, wordToParse
Output: N/A
function traverse
    if state.stateType != 'endState' then
        k ← 0
        for i ← 0 to sizeof(state.actions) do
            processedStates[k] ← state.getNextState(wordToParse)
            k ← k +1
        k ← 0
        for i ← 0 to sizeof(processedStates) do
            if sizeof(processedState[i].candidates) > 0  then
                NextStates[k] ← processedState[i]
                if getSolutionSetSize() > 0 then
                    solution.add(processedState[i])
                k ← k + 1
        for i ← 0 to sizeof(NextStates) do
            for j ← sizeof(NextStates [i].candidates) do
                NextStates [i].candidate[j].wordList ← NextStates [i].wordList
            traverse(NextStates[i], wordToParse)
```

Figure 5.9: The pseudo code for traverse function.

The *traverse* function is recursive function which returns when the passed state is an end state or the list of the states that can be processed is empty. The function takes a state node and evaluates each child. The number of children is defined by number of actions for state node. The evaluation of each child is achieved by calling *getNextState* function. The *getNextState* function is calling the *NextState* function of the action object. The pseudo code for the *NextState* function of the state is given in Figure 5.10. The *getNextState* function returns the state, which is stored in the *processedStates* list. Each element of this list has a list of *candidates* or list of solutions. The state whose candidates list is not empty is added to the *NextStates* list and those whose solutions list is not empty are added to the *solution* list. Some nodes can be child of themselves. An example is the node named noun in Figure 3.2. In addition, some morphemes consist of one vowel which is deleted on the surface in certain situations. As a result, in these circumstances the traverse function will fall in endless loop. To avoid those loops the number of repeatable same morphemes is limited. Each element of *NextStates* points to the state to be processed next, but this state has no information about candidate words. Therefore the list of candidate words is copied to the state list of words. And then, each element of the *NextStates* list is passed to traverse function with the word to be parsed.

The *NextState* function in Figure 5.10 is used to get the state that can be the candidates or the solution for word lexical representation. First of all the lexical word representations are generated by concatenating words from child state's list of words and each suffix from suffixes list. The suffixes list contains suffixes that are located on the edge from parent vertex to child vertex. After that, the rules are applied and the sur-

```
Input: wordToParse
Output: next state to process
function NextState
    k ← 0
    for i ← 0 to sizeof(suffixes) do
        for j ← 0 to sizeof(childState.wordList) do
            generatedWords[k] ← wordList[j] + ' + ' + suffixes[i]
            k ← k + 1
    for i ← 0 to sizeof(generatedWords) do
        surfaceWord ← applyRules(generatedWords[i])
        if wordToParse starts with surfaceWord then
            processedState.candidatesList.add(generatedWords[i])
        if wordToParse  = surfaceWord then
            processedState.addSolution(generatedWords[i])
            processedState.candidatesList.add(generatedWords[i])
    return processedState
```

Figure 5.10: The pseudo code for NextState function.

face word representation is obtained. The surface word form is compared with the word to parse. If the word to parse starts with the surface word form then the lexical representation of the word is added to the candidate list of the *processedState* structure. This structure consists of such information as candidates list, solution list and pointer to the state. If the word surface representation is the same with word to parse then the word's lexical representation is added to the list of solutions. When the list of generated lexical word's representations is processed the *processedState* is returned.

5.5 TRANSLATION SYSTEM

The translation is implemented outside of the morphological analyzer's implementation. It was implemented as a simple function which takes source language morphological analyzer, destination language analyzer and word as parameters and outputs the list of translations. The pseudo code for the function is given in the Figure 5.11.

```
Input: sourceLanguageAnalyzer, targetLanguageAnalyzer, word
Output: list of translations
function translate
    listOfLexicalWords ← sourceLanguageAnalyzer.parseWord(word)
    for i ← sizeof(listOfLexicalWords) do
        maximalStem ← listOfLexicalWords[i].maximalStem
        wordRoot ← listOfLexicalWords[i].wordRoot
        stemType ← listOfLexicalWords[i].maximalStemType
        if maximalStem = wordRoot then
            mapPath ← listOfLexicalWords[i].mapPath
        else
            for j ← 0 to sizeof(listOfLexicalWords[i].suffixes) do
                genWord ← applyRules(wordRoot,listOfLexicalWords[i].suffixes[j])
                if maximalStem = genWord then
                    break
                mapIds = split(wordRoot,listOfLexicalWords[i].mapPath,'~')  j ← j + 1
            for k ← j to sizeof(mapIds) do
                if k = j then
                    mapPath = mapIds[k]
                else
                    mapPath ← mapPath + '~' + mapIds[k]
          listOfTranslatedStems ←
            getTranslationFromDictionary(sourceLanguageAnalyzer.language,
            targetLanguageAnalyzer.language, maximalStem, stemType)
        for i ← 0 to sizeof(listOfTranslatedStems) do
            translatedWords ← targetLanguageAnalyzer.getWordFromMapPath(
                listOfTranslatedStems[i], mapPath, stemType)
            for j ← 0 to sizeof(translatedWords) do
                result.add(translatedWords[j])
return result
```

Figure 5.11: The pseudo code for translation function.

In translate function the word wished to be translated is firstly given to the morphological analyzer of source language. The result from morphological analyzer is the list of structures. Each structure consists of lexical word form, word root, word speech part, word maximal stem, word maximal stem speech type, list of map ids, and a list of suffixes. An example for the structure content is given in Figure 4.2.

After the word is parsed, the new list of map ids is created. This list is created by taking last map id of derivational suffix and all following it map ids. The reason for such approach is the difficulty to create relation between derivational suffixed of two any language. After the list of map ids has been constructed the translation is looked up in the dictionary. The resulting list of translation stems, list of map ids, and maximal stem‘s speech are then passed to the *getWordFromPath* function of destination analyzer. The *getWordFromPath* function is used to generate a word from the given path. The result of the function is the list of generated words. The pseudo code for the function is given in Figure 5.12.

In *getWordFromPath* function the first element of map ids list is used to determine the entrance point to the morphological analyzer. This is achieved by visiting each node of the morphological analyzer and comparing its map id with the first element from the map ids list. In addition, type of node is checked for being initial state and node’s speech part is checked to be the one passed to the function. If no of these conditions is satisfied an empty list is returned. Once the conditions are satisfied such information as state satisfied condition, word stem, and map id is stored in the structure *vState*. The structure then added to the list of visited states.

For the rest of the element of the map ids list each element from visited states list is processed. For each child, if any, of the visited state, the map id is compared with the processed one. If the map ids match then the list of words for the visited state is appended the suffixes of the child node and then rules are applied. After that the child node is added to the visited states list. After that the states that were added from previous loop but are not the solution for the problem are removed from the list of visited states.

And finally, the list of words is copied to the result list from visited states' word list. During this operation the words lists are copied from states that are marked to be a solution.

```
Input: wordStem, mapPath, rootType
Output: the list of words generated from path
function getWordFromPath
    mapIds ← split(mapPath, '~')
    for i ← 0 to sizeof(mapIds) do
        if i = 0 then
            for j ← 0 to sizeof(states)  do //states is the list of state in the morphologicanalyzer
                if  states[j].mapId = mapIds[i] and states[j].stateType = 'initialState' and
                    states[j].rootType = rootType then
                    vState.state = states[j]
                    vState.words.add(wordStem)
                    vState.visitedPath = states[j].mapId
                    visitedStates.add(vState)
        else
            if sizeof(visitedStates) = 0 then
                return empty
            else
                statesSize ← sizeof(visitedStates)
                for j ← 0 to sizeof(visitedStates) do
                    state ← visitedStates[j].state
                    if state.stateType != 'endState' then
                        wordsSize ← sizeof(state.words)
                        for k ← 0 to sizeof(state.actions) do
                            action ← state.actions[k]
                            if action.resultingState.getType != 'initialState' then
                                if visitedStates[j].visitedPath != mapPath then
                                    constructPath ← visitedStates[j].visitedPath + '~' +
                                        action. resultingState.mapId
                                    if mapPath starts with constructedPath then
                                        for m ← 0 to sizeof(action.suffixes) do
                                            vState.state = action.resultingState
                                            for n ← 0 to wordsSize do
                                                state.words.add(applyRules(state.words[n],
                                                    action.suffixes[m]))
                                            vState.wordStem = wordStem
                                            vState.visitedPath = constructPath
                                            visitedStates.add(vState)
                k ← 0
                for m ← 0 to statesSize do
                    if visitedStates[k].visitedPath != mapPath then
                        visitedStates.removeAt(k)
                    else
                        k ← k + 1
    for i ← 0 to sizeof(visitedStates) do
        //some nodes not accepted as a solution when they are located at the end of the chain
        if visitedStates[i].visitedPath = mapPath and visitedStates[i].state.canBeSolution then
            for j ← 0 to sizeof(visitedStates[i].words) do
                resultList.add(visitedStates[i].words[j])
    return words
```

Figure 5.12: The pseudo code for getWordFromPath function.

5.6 SAMPLE TRANSLATION

A sample translation from Turkmen to Turkish is shown in Figure 5.13.

Figure 5.13: A sample for translation from Turkmen to Turkish.

The words that have more than one translation are shown in combo boxes such as *biz*. The words that were not translated are followed by the possible reason for failure in square brackets. There can be two possible reasons: one is *no root* meaning that morphological analysis failed. The second is *no stem* meaning that translation dictionary doesn't have this stem.

CHAPTER 6

CONCLUSIONS

The machine translation is the one of the fundamental problems in Natural Language Processing. It is a challenging task. However, the design of translators for languages similar in terms of morphological, syntactical and lexical structures seems to be relatively easier. A translation framework between Turkic languages was designed and implemented in this study. The framework was used to implement Turkmen – Turkish and Turkish – Turkmen translators.

A common approach used in morphological machine translation has three components: the two–level morphological analyzer, stem translator and word generator. The two–level morphological analyzer is used to get a word's lexical form. The stem translator uses the stems of the word's lexical form as an input, and outputs a list of stems for the target language. After that, the stems in target language are passed to the word generator. The output of the word generator is a list of words in the target language.

The morphotactics for Turkmen language was studied and orthographic rules for Turkmen were written. After the design these rules were encoded in XML. And finally, a two–level morphological analyzer was implemented in Java. The encoding of morphotactics and orthographic rules into XML gives users unfamiliar with programming ability to define morphotactics and orthographic rules for agglutinative languages.

Initially the system was designed for two-way Turkmen-Turkish translation. However, isolating morphotactics and orthographic rules from the system and storing them in XML gave the system ability to work with other languages as long as their morphotactics and orthography can be represented in the XML format set forth here.

The root dictionary is a must for two - level morphological analyzer. Two root dictionaries were used, one for Turkish, and the other for the Turkmen. Current dictionaries contain 14.888 word roots for Turkish and 10.349 word roots for Turkmen. However, new roots can be added in the future thru the web interface. The words in the dictionary are divided into eleven categories (part of speech). These are noun, adjective, verb, compound noun, adverb, exclamation, proper noun, preposition, compound verb, conjunction, and pronoun.

In initial testing, it was noticed that the words which should take inflectional suffixes are also can be presented as root and a chain of derivational suffixes, therefore producing unexpected results. For example, *gelir (income)* and *gelir (he comes)* have same morphological representation *gel* + *Hr*. These cases will be dealt with in the future.

The translation component is one of the important components of the system. It consists of two stem dictionaries (lexicons). This component simply performs a lookup on a target language stem and returns the corresponding list of destination language stems. A web interface was designed to manage stem dictionaries.

The word generator component was implemented using the list of unique ids referring to states of the morphological analyzer's finite state automaton. Each state has a unique id. When a word is passed from one state to another state the ids are combined in a list. The system then takes this list and uses it to generate a word by passing the list of ids and a translated - stem to the morphological analyzer of the target language. Naturally Turkmen and Turkish languages have differences in terms of lexical structures. Since, the generated list of ids in Turkmen was different from that in Turkish. To deal with such differences a map translator was introduced. The differences of such mappings were encoded into XML to provide flexibility to users. The map translator takes a list of ids of source language and returns the one or more lists of ids for the destination language. As a result, a word can be translated to one or more words.

Current machine translator is designed to translate words on one to one basis. Therefore, the created output can be misunderstanding since the syntactical structure of the sentences may differ for languages. To remedy this problem to some extend we use word groups in addition to the single word in the dictionary. So the word in any

language can correspond to a word or a group of words (phrases) in the other language. The part of the system is experimental. As indicated above, a word or phrase can have many translations. Therefore, a sentence translated in such manner can have many more translations than a single word or phrase. Choosing the best translation among many is another challenging task and outside this study. Currently web interface produces all possible translations.

REFERENCES

Altıntaş K. & Çiçekli İ., "A Morphological Analyzer for Crimean Tatar", *Proceedings of the 10th Turkish Symposium on Artificial Intelligence and Neural Networks, TAINN*, pp. 180-189, North Cyprus, 2001

Canals-Marote R., Esteve-Guillén A., Garrido-Alenda A., Guardiola--Savall M.I., Iturraspe-Bellver A., Montserrat-Buendia S., Pérez-Antón-Rojas P., Ortiz-Pina S., Pastor-Antón H. & Forcada M.L., "interNOSTRUM: a Spanish-Catalan Machine Translation System", *Machine Translation Review*, Vol.11, pp. 21-25, 2000

Dvořák B., Homola P. & Kuboň V., "Exploiting similarity in the MT into a minority language", *LREC-2006: Fifth International Conference on Language Resources and Evaluation*, Genoa, Italy, 2006

Garrido-Alenda A., Gilabert-Zarco P., Pérez-Ortiz J.A., Pertusa-Ibáñez A., Ramírez-Sánchez G., Sánchez-Martínez F., Scalco M.A. & Forcada M.L., "Shallow Parsing for Portuguese-Spanish Machine Translation", *TASHA 2003: Workshop on Tagging and Shallow Processing of Portuguese*, Lisbon, Portugal, 2003

Jurafsky D. & Martin J. H., *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*, Prentice Hall, New Jersey, 2000

Hajič J., Hric J. & Kuboň V., "Machine translation of very close languages", *Proceedings of the sixth conference on Applied natural language processing,* pp. 7-12, 2000

Hajič J., Homola P. & Kuboň V., "A simple multilingual machine translation system" *MT Summit IX*, New Orleans, USA, 2003

Kara M., *Türkmen Türkçesi Grameri*, Ankara, 2005

Koskenniemi K., "Two – level morphology: A general computational model of word-form recognition and production", *Tech. rep. Publication*, No. 11, Department of General Linguistics, University of Helsinki, 1983.

Oflazer K., "Two – level Description of Turkish Morphology", *Literary and Linguistic Computing*, Vol. 9, No. 2, 1994

Tantuğ A. C., Adalı, E. & Oflazer K., "Computer Analysis of the Turkmen Language Morphology", in T. Salakoski (Eds.), *FinTAL 2006, Lecture Notes in Computer Science*, pp. 186-193, Springer, 2006

Tantuğ A. C., Adalı, E. & Oflazer K., A MT System from Turkmen to Turkish Employing Finite State and Statistical Methods, *Proceedings of MT Summit XI*, 2007