OVERCOMING THE FONT AND SCRIPT BARRIERS AMONG INDIAN LANGUAGES

Master of Science (by Research)
in Computer Science

Himanshu Garg
200207004

International Institute of Information Technology
Hyderabad India
March 2006

OVERCOMING THE FONT AND SCRIPT BARRIERS AMONG INDIAN LANGUAGES

A Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (by Research)
in
Computer Science

by

Himanshu Garg

200207004

himanshu@students.iiit.ac.in

International Institute of Information Technology, Hyderabad

March

2006

CERTIFICATE

It is certified that the work contained in this thesis, titled "Overcoming the Font and Script Barriers among Indian Languages" by Himanshu Garg, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Advisor: Amba Kulkarni

To Ambaji

ABSTRACT

The current status of Indian languages on the web may best be described as "Diversity in Unity"! All Indian language scripts (except for Urdu & Sindhi) are derived from the same Brahmi Script and also share a common alphabet to a large extent. But languages like Hindustani, Sindhi use different scripts making the texts available in one script inaccessible to persons not knowing that script. Because of different scripts for different Indian languages, which otherwise share common culture, two languages can't share language independent information with each other. When it comes to electronic media, the situation is still worse. Even the texts in the same language are font dependent making them unsharable. Though standards exist they are not followed!

In this thesis we propose solutions

- to overcome the font barrier within a language for all brahmi based scripts.
- to overcome the script barriers across the language among all brahmi based scripts
- to overcome the script barrier in case of Hindustani-Hindi-Urdu

The problem of font barriers within a language is because of different coding schemes followed by different font designers. While a standard coding scheme (Unicode/ISCII) exists, it is not used. Converters that can convert from the unknown encoding to ISCII can largely solve the problem. However manually coding converters is difficult and time consuming. This thesis attempts to tackle the problem of generating such converters, using two different approaches. The first approach uses a mnemonic based description of glyphs in the font. This description is used to convert a font encoded text. Accuracies of above 75% for two Telugu fonts and above 90% for three Devanaagari fonts have been obtained. The second approach uses example based machine learning techniques to generate Finite State Transducers. These Finite State Transducers can be automatically

learnt from syllable mappings of font encoded text to Unicode text. Accuracy of above 90% has been obtained when 1000 syllable mappings from font encoding to Unicode were provided.

The second problem of overcoming the script barriers across the languages may simply be solved by following the ISCII standard.

The transliteration from Urdu to Hindi makes a text written in the Perso – Arabic script accessible to those who know the Devanaagari script. Such a transliteration is non trivial because there is not a one to one mapping between Urdu and Hindi alphabets. Omission of vowel signs in Urdu which must be present in Hindi complicates the issue further. We discuss these and other issues involved in overcoming the script barriers in case of Urdu - Hindi and propose a solution to overcome this barrier.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I can't thank my advisor Ms. Amba Kulkarni enough but I still thank for her guidance, patience and support during my thesis work. The improvement in the OSTIA algorithm is largely due to discussions with Dr. Vineet Chaitanya who helped identify the problems with the learning of the original one. The work on Telugu was taken up on Dr. Rajeev Sangal's suggestion. Thanks are also due to Dr. Uma Maheshwar Rao who helped me get started with the work on Telugu by discussing the issues and providing relevant material. I am also grateful to the number of students at IIIT who spared their time for me to help evaluate my programs for Telugu, a language which was earlier 'Greek' to me. The thesis includes another language new to me i.e. Urdu. The work on Urdu is largely due to impetus given by Prof. Rahmat Yusuf Zai and Dr. M. S. Hayat who spent time with us in fixing the transliteration. I am also grateful to so many others at the Akshar Bharti group at LTRC whose programs and support I used so often during the thesis work. Many improvements to the report were made after feedback from the reviewers. I thank them for their time. I will also be thankful to my parents and friends who pushed me for winding up my thesis work and submitting this report. Without their push and help, this report might never have been written and the work would never have reached the stage it has.

CHAPTER 1

INTRODUCTION

There has been a chaos as far as Indian languages in electronic form are concerned. Neither can one exchange the notes in Indian languages as conveniently as in the English language, nor can one perform search on texts in Indian languages available over the web. This is so because majority of the texts are being stored in non standard formats. In this thesis we attempt to ease the conversion of Indian language texts to standard formats. Additionally we also look at Hindi-Urdu transliteration so that a person who knows the Devanaagari script can access Urdu texts written in Perso-Arabic script.

The following sections provide the necessary introduction with section 1.1 giving the motivation for attempting this problem, section 1.2 giving the necessary background to gain an understanding of issues involved in Indian language display, section 1.3 provides a formal definition of the problem along with the different kinds of inputs available to solve it. The report organization is given in section 1.4.

1.1 MOTIVATION

Text on the computer can be displayed in different styles. Each style corresponds to a font. For example shown below is the word Hello written in four commonly used fonts:-

Font	Arial	Times New Roman	Comic Sans	Courier New
Example Word	Hello	Hello	Hello	Hello

Table 1 Same word, different fonts

Each character that is displayed or stored in the computer has a distinct numeric code used to distinguish it from other characters. Interestingly a character of the English language has the same code irrespective of the font being used to display it. For example ‘a’ has the same numerical code ‘97’ irrespective of the font that is used to display it.

Consider for example the word “Hello” written in the Roman script:-

Font	Arial	Times New Roman
Word	H e l l o	H e l l o
Numeric code for each character	72 101 108 108 111	72 101 108 108 111

Table 2 Different fonts, same underlying numeric codes

Two fonts whose names are Arial and Times New Roman are used to display the same word. The underlying codes for the individual characters, however, are the same and according to a standard called ASCII. Searching the web for the word Hello returns the web pages containing the word Hello irrespective of the font used to display it. This works because the underlying codes used to store the word Hello are always the same.

On the other hand most fonts for Indian languages assign different codes to the same character. Therefore when we consider Devanaagari fonts, the situation changes:-

Font	Jagran	Yogesh
Word	ि म ɵ य ा	ि म ा ɵ ɾ ा ा
Underlying Code for each byte	231 215 137 216 230	202 168 201 108 170 201 201

Table 3 Same word, different fonts, different codes

We will refer to the different shapes used by a font as glyphs. Examples are H e l l o ि म ा ɵ ɾ . What must be noted is that for Indian language fonts instead of characters, numeric codes exist for glyphs which combine together to form larger units. Moreover the numeric codes for glyphs (glyph codes) are not uniform across fonts. The code for ा is 230 in one font and 201 in another. म has a numeric code 215 in one font while two numeric codes 168 and 201 in another because the latter forms म from two glyphs.

Font	Jagran	Jagran
Word	ि म ɵ य ा	ि म ा ɵ ɾ ा ा
Underlying code for each byte	231 215 137 216 230	231 144 230 137 212 230 230

Table 4 Same word, same font, different codes

Moreover same fonts can be used to write the word which looks the same visually but uses another set of numeric codes.

This causes little problem as long as the only use of the text is viewing it as if it were on paper*. Problems arise when one tries to take advantage of its being in the electronic format. For example with the existing search engines such as Google someone trying to search the word मिथ्या gets only those web pages that use the same font as that used for searching and written using the same glyphs. Searches done with मिथ्या written in the Jagran font return only those results that contain the word written in Jagran font. Searches done with मिथ्या written in the Yogesh font return only those results that contain the word in Yogesh font. This is one reason why it is currently not possible to search Indian language texts on the web.

Standard schemes for assigning numeric codes to characters/glyphs exist but they have not been followed. Unicode is an example of a standard encoding. The following table shows the underlying codes for the characters forming the word मिथ्या.

Word	म ि थ ् य ा
Codes according to standard coding scheme	092e 093f 0925 094d 092f 093e

Table 5 A word written using a standard encoding scheme called Unicode

Text in electronic format has many other important uses in Natural Language Processing [1] including corpus building [2], conversion to speech etc. each of which becomes difficult due to the non standard encoding. Each of these now applications now requires the additional task of converting the text to a standard encoding. We refer to programs that carry out this conversion as converters.

* Problems also arise if the font used to write the text is not supported by the system on which it is being read. This happens when the source of the text is different from the place where it is being viewed. Conversion of text to a standard encoding again becomes useful here.

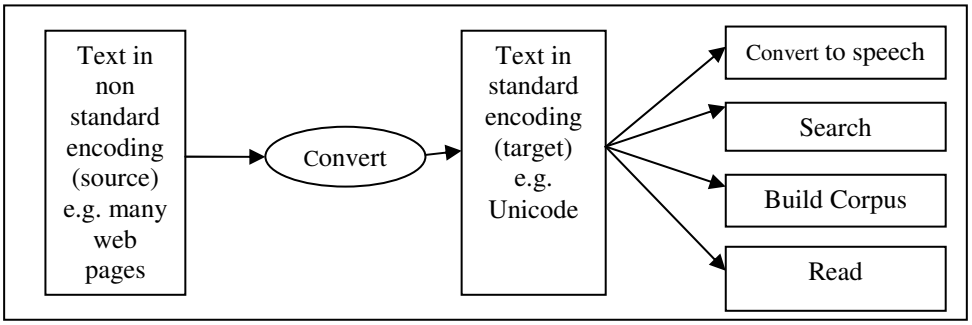

Figure 1 Conversion of a text in non standard encoding to a standard one and subsequent uses.

Sometimes applications requiring display of text can only do so by converting it to a non standard encoding. This happens when display of a standard encoded text directly is not supported by the system. Examples of such applications include some word processors [12], programming environments [3]. Applications that require conversion of text to image for looking up images with similar text [4] and for training Optical Character Recognition systems [5] also require conversion from a standard encoding to a glyph based encoding. While the solutions presented here (in particular see 3.2) can be adapted to creating such converters also, the requirements in such applications are of much higher conversion accuracy than is obtainable directly with the solutions proposed here.

The difficulty in writing converters from a non standard encoding to a standard one arises for various reasons:-

- There are about 15000 distinct source strings that must be mapped to target strings. Many mappings of these are many to one because of multiple ways of writing the same word (See
- Table **4**). Enumeration of these manually is difficult if not impossible.
- Sample data in source encoding is too little. Such a situation could arise when an article written in non standard encoding needs to be converted and the article is the only one that uses the particular non standard encoding.

- The text that needs to be converted cannot be read because the only font that can display it is unsupported by the system. This happens when one visits a website on the internet that uses a font that cannot be installed on the reader's system.
- Even though the user may be able to read some specific text he/she may not be able to display shapes arising from arbitrary glyph code sequences. Such an ability could simplify the task of converter writing. This happens especially when the document being viewed contains the font embedded inside it. This allows the document to be viewed alright but other texts cannot be viewed using the same font.
- The distinct shapes that appear in the font are known but the rules by which these combine are unknown or hidden in a proprietary s/w. An example of this is the ILeap s/w which displays Indian Language texts. Converter programs then must be able to convert all glyph combinations (See
- Table **4**) .

This thesis attempts to reduce the difficulty in writing programs that do the conversion from a non standard encoding to a standard one. The solutions presented here become useful especially in the last two situations mentioned above.

A second problem that has been attempted is related to the script barriers across languages. India though is multilingual, shares a lot at the cultural level. Not only the scriptures/epics such as Ramayana, Mahabharat, Bhagvadgita but also festivals, names etc are common. This language independent information would be unsharable had there been no common encoding scheme. Standards such as ISCII and Unicode exist. Each has its positive and negative aspects. Unicode suffers from the problem of using different codes for the same character in different scripts, but is otherwise good because it covers most of the world's languages. Another encoding standard ISCII [7] doesn't suffer from the problem but is not universal. There is thus a necessity of a common encoding scheme even at the

Unicode level. Until that happens, the intermediate solution is to build converters among Unicodes for different languages.

There is a unique case politically motivated by the then British rulers – use of different scripts for the same language Hindustani! Hindustani in the Devanaagari script came to be referred as Hindi and that in the Perso-Arabic script as Urdu.

1.2 BACKGROUND

1.2.1 Script Types

A script defines a distinctive and complete set of symbols used for the written form of one or more languages. The scripts may be broadly classified into the following classes:-

Pictorial: Chinese, Japanese and Korean have a pictorial script. In these languages each concept corresponds to a picture or a combination of pictures. The basic pictures are in tens of thousands in these languages. Since there were too many symbols than could be accommodated in the conventional typewriter, an innovative solution was found. Photo copying machine was invented to reproduce the required shapes. On computers also naturally a keyboard driver was designed to input the text in these languages.

Alphabetic: In this representation a unique symbol is assigned to consonants and vowels. For example the word Indian composed of the symbols *I, n, d, a*. There is no necessity of special display drivers and keyboard drivers since there is a one to one mapping between characters and glyphs. The roman script is an example of this class.

Syllabic: In this scheme there is a different “atomic” symbol for each syllable.

Compositional Syllabic: In this scheme syllables are the basic unit in writing, but they are made up of alphabetic sequences. Brahmi script and all Indian scripts derived from it are examples of this class. A unique symbol is attached to each syllable. However the syllables themselves are made up of alphabetic characters. For example मिथ्या is composed of two syllables मि and थ्या. Syllables in the syllabic notation are written in the order they are pronounced.

Syllable	Constituent Consonants and Vowels
मि	म् इ
थ्या	थ् य् आ
क	क् अ
म	म् अ
ला	ल् आ

Table 6 Syllables and their constituent consonants and vowels

1.2.2 Nature of Indian Scripts

There are two kinds of letters in an alphabet: consonants and vowels. Consonants are those letters which cannot be pronounced without a surrounding vowel eg. क् म् ल् म् थ् य्. Vowels are those letters that can be pronounced independently e.g. अ आ इ.

The brahmi script and brahmi based Indian language scripts are compositional syllabic in nature. A syllable is defined as a series of one or more consonants followed by a vowel. Some examples of syllables are थ्या मि क ला.

1.2.3 Salient Features of Syllabic Scripts

The sequence of consonants is indicated either by writing them from top to bottom as in the case of ట్ర, ద్దు or from left to right త్ర, ग्वा .Telugu prefers top to

bottom whereas in Devanagari we find both usages, though left to right is more prominent.

The vowel अ (a) is part of the consonant, so e.g. क (ka) stands for क् (pure consonant k) followed by अ (a).

Since vowel अ is inbuilt, a concept of 'matras' is introduced in our scripts. The relation between a vowel and its matra is discussed in [6]. For example the relation between इ and ि is इ = अ + ि or ् + इ = ि as in म् + इ = मि

The position of matra has nothing to do with its pronunciation order. The order of pronunciation is strictly governed by definition by a syllable and hence vowels are to be pronounced at the end, irrespective of their position at the script level.

1.2.4 Standards for Indian Scripts

Standards that took the nature of Indian scripts into account were prescribed but were never followed. Here we give a brief description of the standards and the possible long term solution.

1.2.4.1 ISCII

For Indian languages already the standard exists in terms of ISCII [7]. This standard specifies codes for characters in the Indian language alphabet. Since all Brahmi based Indian scripts share the same alphabet the code it assigns to a character is same across all of them.

ISCII Code	Devanaagari	Telugu
179	క	క
180	ख	ఖ
181	గ	గ
182	ఘ	ఘ

Table 7 The ISCII standard assigns same code to the same character in 10 Indian languages

1.2.4.2 UNICODE

Unicode is an international standard that assigns codes to all the world's languages. As a result different languages can coexist in the same document without any additional markup. It is becoming increasingly popular due to its global nature. Critiques can be found in [8] [9] and an alternative in [10]

1.2.4.3 ISFOC

CDAC proposed a glyph level standardization ISFOC [11] and a custom keyboard driver called Inscript. This was proposed so that there was uniformity in the glyph encodings. However this was not followed and fonts using their own glyph encoding schemes continued to be used.

1.2.4.4 Others

There are others which cannot really be called standards but are mentioned here because they are used to store Indian language texts. These include itrans, omtrans, wx, etc. These are notations that use symbols from the roman script to

store Indian language characters. Their importance arises from the fact that they use the roman notation which can be viewed almost everywhere. There is no dependence on availability of fonts, conversion to a glyph encoding etc.

1.2.5 Font Formats for Indian Scripts

Various font formats exist for displaying scripts. We will discuss the TTF and OTF formats which are the prominent ones. A font encodes the glyphs that will be displayed by it and codes that will be displayed using one or more of these glyphs.

1.2.5.1 TTF

TTF or the True Type Font format has been used for the Roman script. Its only shortcoming is that it can map a numeric code to only one glyph. Due to its widespread support the same was adopted for Indian languages. While this worked well for roman script where there is a one to one correspondence between a character and its shape for complex scripts such as Indian this required the use of glyphs that combine to form different syllables.

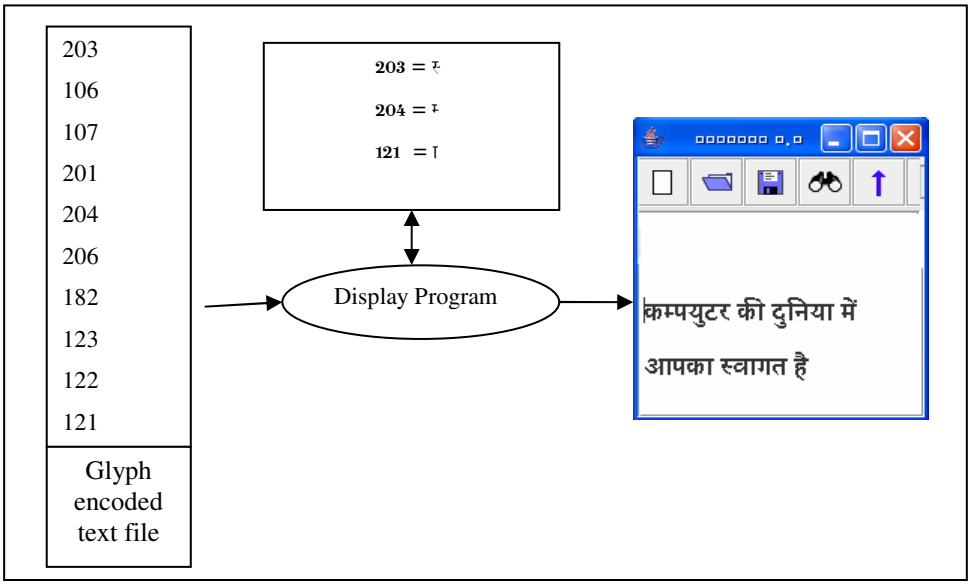

Figure 2 Abstraction of how display of a text file takes place using TTF's

1.2.5.2 The Need for Font Converters

The font developers did not follow the ISCII/Unicode standards because they require a program (keyboard driver) to map keystrokes to character codes and a sophisticated program (rendering engine) to map one or more character codes to one or more glyph codes (In TTF we can do only one code to one glyph mapping). The operating system did not support the sophisticated rendering engine (until Windows 2000 and GNU/Linux a little later). Applications that did follow it had to have their own keyboard drivers and rendering engines [12]. Some of those who did not follow the standard used ad hoc solutions in order to avoid the display driver and keyboard driver problem. The font glyph mappings and/or the design of fonts was governed by factors such as

a) Familiarity with the keyboard (e.g. Kruti Dev font)

b) Ease of use of the keyboard (e.g. Shusha)

c) Aesthetic considerations (e.g. Yogesh)

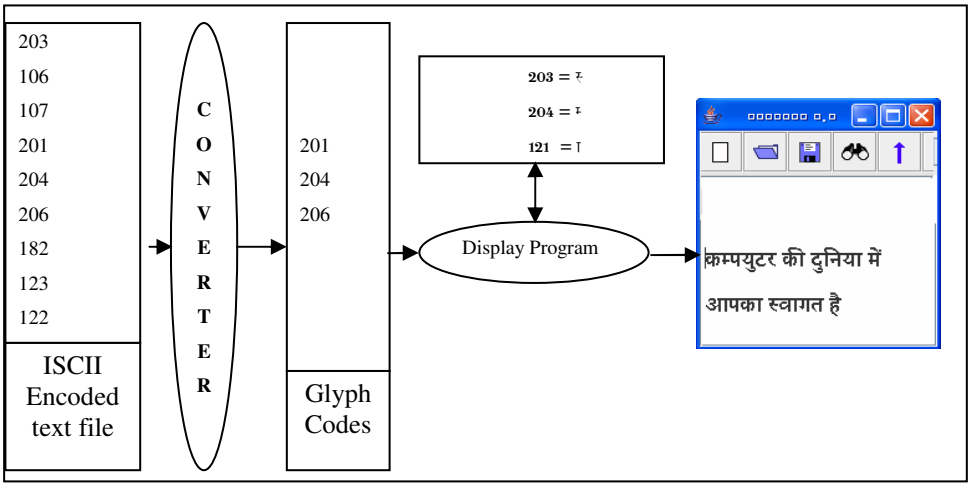

Figure 3 Workaround to TTF's limitation used by Indian Linux and ILeap

Language processing applications that work with the older True Type fonts store and process the text in a glyph-based representation. Alternatively the storage is done in a standard encoding and for viewing it is converted to a font encoding (See Figure 3). We therefore see the need for conversion to a font encoding to display a standard encoded file.

Problems arise when the font encoded file created for display is transferred to other locations. If the font is not supported at the destination the file cannot be read. Even if the font is supported it is difficult to process the file because it is glyph encoded. Indian Linux [13] does not suffer from this problem because the conversion process is transparent to the user and it is not possible for the user to transfer a font encoded file. The problem with it is however that it works with some (now older) versions of the GNU/Linux operating system.

1.2.5.3 OTF

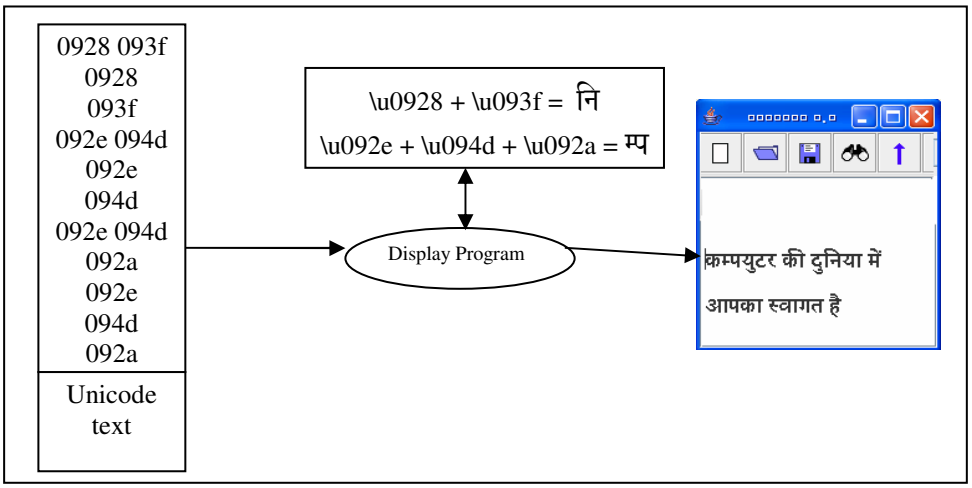

Figure 4 Display of Unicode text using an OpenType font and an associated display program

The OTF or the OpenType Font format overcomes the problems associated with TTF by also encoding the character code to glyph conversion rules within the font. The TTF fonts which had space only for 256 bytes are now being replaced by Open Type fonts which have no such limitation. Moreover TTF fonts had no provision to state the glyph grammar rules. Open Type fonts make it possible to specify the glyph grammar within the font. This glyph grammar gives the mapping of the character codes to glyphs. The rendering engine then uses this information to map character code sequences to appropriate glyphs.

The Open Type font format solves this problem by forcing the font developer to make the character(s) to glyph(s) mapping explicit within the font itself. Codes which were assigned to glyphs in the TrueType fonts are no longer assigned by the font developer. His/Her task is to map the standard Unicode character codes to glyphs thereby eliminating the problem due to font dependent glyphs and glyph codes. The glyphs that are used to display different syllables are all internal to the Open Type font.

Applications such as text editors, word processors that work on platforms that use the OpenType font format don't have to do the conversion from character based representation to a glyph based representation. The conversion is done automatically by the rendering engine using the OpenType font.

So the long term solution is to follow the Unicode standard for storing and processing texts and using Open Type fonts for displaying them[†]. Shifting to Unicode from the font based encodings will take some time. Till then we should have some short term solutions. Moreover we also need solutions to convert the existing texts in Indian languages in proprietary fonts to Unicode.

1.3 FORMAL DEFINITION

Given a language L_G (the language of **G**lyphs) consisting of about 200 symbols. Each symbol has an associated numeric code and a distinct shape called glyph and an associated position in which it is displayed. Examples of such a language include a font with its glyphs. This language is used to display a natural language.

Given another language L_C (the language of **C**haracters) consisting of about 150 symbols. Each symbol has an associated numeric code and a name. A symbol in isolation has a shape which is different from the shape when it appears with other symbols. A number of symbols combine to form a distinct visual unit called a syllable. Examples of such a notation include the ISCII and Unicode standards. This language is used to store/process natural language texts.

Each language contains potentially infinite number of valid strings. However since both are used to display/store natural languages the number of strings that cover more than 99% of a natural language is limited to between 10000 and 15000.

One or more than one strings in L_G can be used to display a given string in L_C. The task is to identify the 10000 or so strings in L_G and their corresponding

[†] There still are issues with UTF-8 with respect to Indian languages, however these being beyond the scope of this thesis, are not discussed here.

equivalents in L_C. Alternatively rules that can be used to convert L_G strings to L_C strings can also be identified.

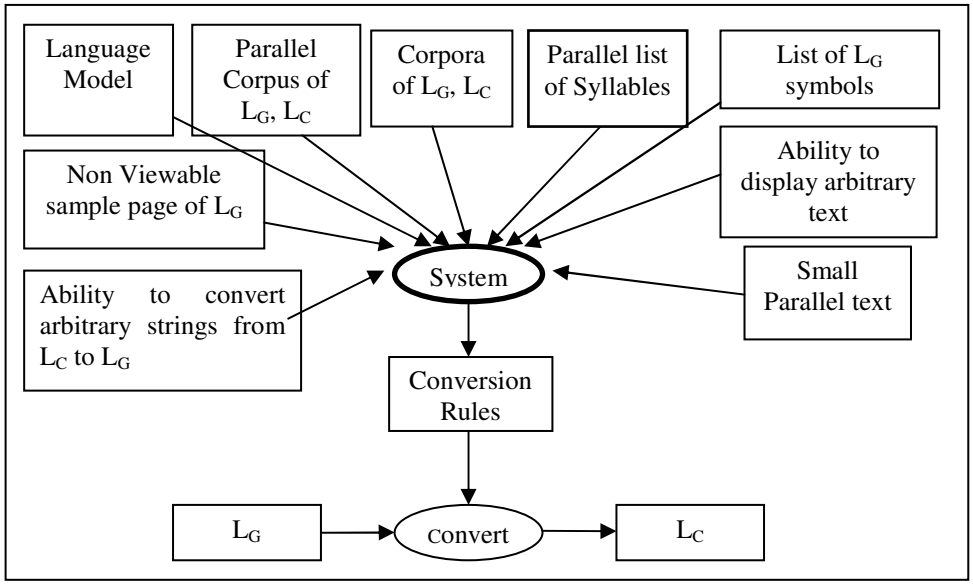

Figure 5 Potential resources available to a sytem that has to determine conversion rules

The task is to determine the rules of conversion of strings in L_G to strings in L_G. Different inputs are available to a system that wants to figure out the conversion rules from L_G to L_C. These include the following in the increasing order of difficulty for a system that has to automatically identify the rules:-

1. **Ability to convert arbitrary strings from L_C to L_G**. Such ability allows us to get L_G equivalents of all strings available in L_C. These mappings can then be used to convert from L_C to L_G. Problems arise only when L_G strings are written in ways different from that generated by the system.
2. **Installed Font or ability to display arbitrary text**. Such ability allows the user to generate the visual representation of all strings in L_G and see automatically or manually whether it has L_G equivalents. This approach was identified during a group discussion by Dr. C. V. Jawahar of IIIT, Hyderabad.
3. **List of L_G symbols**. A table of all the symbols is available that lists all the glyphs with their numeric codes. This can then be used to write rules manually to convert from L_G to L_C.

4. **Parallel list of Syllables**. A list of syllables in L_C and L_G is available. Since syllables are the basic unit of shaping this input contains more information than a parallel list of words. This input can then be used to train an example based learning system.
5. **Parallel corpus of L_G, L_C.** Similar to 4 above.
6. **Small parallel text**. Although the reader can view the text he/she cannot select text to create a list of L_G symbols..
7. **Corpora of L_G, L_C** These can be used to create L_G and L_C models. To what extent are the models useful to convert L_G texts to L_C is unknown to the author.
8. **Non viewable sample page of L_G**. Example of this include texts where the font used to display the text is unavailable. All that the user has is numeric codes that represent some unknown text.

Sometimes more than one of the above may be available. The above ordering assumes only one resource is available at a time.

When looked at from a Machine Learning perspective one must identify the version space [14] for this problem. A hypothesis for this problem is a set of mappings from L_C strings to L_G strings. Assuming each string is no more than 5 glyphs there are 200^5 such strings possible from 200 or so glyphs. Strings in L_C may or may not map to one of 15000 or so commonly occurring syllables. Therefore if the training examples are denoted by D the size of version space can be bounded from above by the following

$$|V| = (200^5 - |D|) * 15000$$

The thesis attempts to solve the problem when only 3 or 4 are available. One approach works when 3 is available; the other when only 4 is available.

1.4 ORGANIZATION OF THE REPORT

In the Second chapter we look at related work in this area. The Third chapter discusses two solutions – glyph grammar based and Machine Learning based – to develop the font converters automatically. The Fourth chapter suggests a solution to reduce the script barrier among Urdu-Hindi. The Fifth chapter discusses the results.

CHAPTER 2

RELATED WORK

The problem of glyph based encodings is not unique to Indian scripts. Indian scripts are examples of complex scripts mainly because there is not a one to one correspondence between characters and their shapes. Other complex scripts include [15]:-

Arabic
Hangul (Korea)
Hebrew (Israel)
Khmer (Cambodia)
Lao (Laos)
Syriac (Syria, Turkey)
Thaana (Maldives)
Thai (Thailand)

It is not unlikely that such issues exist for most of the above languages as well. However the author is not aware of more than one attempt to automate the process of creation of converters. This could be due to different reasons including the limited number of encodings that need to be converted to standard for these languages. The approach that perhaps all have taken is to program the few needed converters by hand. For example see [16]. In the Indian context the problem is large because of the number of scripts used and also the proprietary nature of

converters. Font vendors / websites provide fonts but do not provide the rules by which font encoded texts can be generated or decoded.

Tools exist for doing the conversion from one encoding to another in the Indian context too. We mention in Section 2.1 tools aimed for encoding conversion. Section 2.2 then mentions perhaps the only attempt at automatic generation of converters.

2.1 TOOLS

2.1.1 STED [17]

Operating System/Platform: Any
Application Integration: None
Conversions Provided: User specified

The tool allows the user to specify rules for conversion from one encoding to another. The rules can contain context information to specify the conditions under which a conversion must be done. The four types of rules include:-

IS First Letter (in the word)

IS Last Letter (in the word)

IS Preceded By (symbol 1 is preceded by the selected symbol)

IS Followed By (symbol 1 is followed by selected symbol)

2.1.2 SILConverters [18]

Operating System/Platform: Windows

Application Integration: Firefox

Conversions Provided:

Source	Target	Source	Target	Source	Target
Itrans	Hindi Unicode Bengali Unicode Gujarati Unicode Telugu Unicode Tamil Unicode Kannada Unicode Oriya Unicode Malayalam Unicode	Devpooja Devpriya DV-TTYogesh DVB-TTYogesh Sanskrit-98 Shusha Mithi DVBW-TTYogesh AkrutiDev1 Ankit Devlys Kruti46 Naidunia Telugu-Hemalatha Telugu-Hemalathab	ISCII	ISCII	Unicode
				Unicode	ISCII
				Annapurna Shusha CDAC-ISFOC IPA	Unicode
				Devanaagari Bengali Gujarati Gurmukhi Kannada Malayalam Oriya Tamil Telugu	Latin

2.1.3 ICU [19]

Operating system/platform: Any

Application integration: NA

Programming language API: C [20]/C++ [21]/Java [22]

Source	**Target**
ISCII	Unicode
Unicode	ISCII

2.1.4 TECkit [23]

Operating System/Platform: Windows
Application Integration: Firefox
Conversions Provided: Same as SILConverters

This tool aims at providing encoding conversion facility and uses mapping tables in a specific binary format. Information about the context in which a conversion works can be specified in these mapping tables. For example the following rules specify how glyphs codes for diacritic signs should be converted to Unicode

```
0x40 > U+0301 ; acute over wide low characters
0xDB > U+0301 ; acute over narrow low characters
0x8F > U+0301 ; acute over wide tall characters
0x90 > U+0301 ; acute over narrow tall characters
```

and the following rule specifies how Unicode should be converted to the appropriate diacritic sign depending upon the context:-

```
0x40 < U+0301 / [lowWide] _
0xDB < U+0301 / [lowNarr] _
0x8F < U+0301 / [highWide] _
0x90 < U+0301 / [highNarr] _
```

The first rule specifies that the Unicode codepoint U+0301 should be converted to the glyph code 0x40 only when it is preceded by classes of characters represented by lowWide. If it is preceded by codepoints of the class highWide target glyph must be 0x8f and so on. More sophisticated notation can be used to specify rules of reordering.

2.1.5 ISCIIG/ICONVERTER [13]

Operating System/Platform: GNU/Linux

Application Integration: Netscape Navigator and others

Programming Language API: C/C++

Source	**Target**
ISCII/Unicode	ISFOC Assamese ISFOC Bangla ISFOC Devanaagari ISFOC Gujarati ISFOC Kannada ISFOC Malayalam ISFOC Oriya ISFOC Punjabi ISFOC Telugu ISFOC Tamil
ISCII	Unicode
Unicode	ISCII

It must be noted that the conversion is provided in one direction only i.e. from ISCII/Unicode to glyph based encoding schemes.

The conversion rules are specified in encoding files containing rules of the following kind

```
CONSONANT LMATRA  -> G_LMATRA($2) $1;
```

The above rule specifies the I vowel sign movement in Devanaagari i.e. any consonant (CONSONANT) followed by the I vowel sign (LMATRA) must be converted to the glyph corresponding to the vowel sign (G_LMATRA($2)) followed by the glyph corresponding to CONSONANT ($1). These rules are read by a yacc [24] based parser after which the necessary conversions are carried out.

2.1.6 Padma [25]

Operating System/Platform: Any

Application Integration: Mozilla based applications including Firefox, Thunderbird, Netscape Navigator etc.

Programming Language API: None

Source	**Target**
Bhaskar (Devanaagari) Chanakya (Devanaagari) EPatrika (Devanaagari) Jagran (Devanaagari) Mithi (Devanaagari) Subak (Devanaagari) Amar Ujala (Devanaagari) Gopika (Gujarati) Nandi (Kannada) Kairali (Malayalam) Karthika (Malayalam) Manorama (Malayalam) Revathi (Malayalam) Thoolika (Malayalam) Kumudam (Tamil) ShreeTam0802 (Tamil) Vikatan (Tamil) Eenadu (Telugu) Hemalatha (Telugu) ShreTel0900 (Telugu) ShreeTel0902 (Telugu) TCSMith (Telugu) TeluguLipi (Telugu) Tikkana (Telugu) Vaartha (Telugu)	Unicode

Each converter is a JavaScript program which is packaged together as extensions to the Mozilla based applications.

2.1.7 Font Converters at the Language Technologies Research Centre, IIIT, Hyderabad [26]

Operating System/Platform: GNU/Linux, Windows

Application Integration: None

Programming Language API: None

Source/Target	**Target/Source**
Devpooja Devpriya DV-TTYogesh DVB-TTYogesh Sanskrit-98 Shusha Mithi DVBW-TTYogesh AkrutiDev1 Ankit Devlys Kruti46 Naidunia Telugu-Hemalatha Telugu-Hemalathab	ISCII

These converters consist of mapping tables containing 10000 to 15000 entries each of which is a source to target mapping. Some entries from the DV-TTYogesh to ISCII converter is shown below:-

```
…
क्ल     ISCII codes for क्ल
क़्सी    ISCII codes for क़्सी
क्     ISCII codes for क्
कँ     ISCII codes for कँ
…
```

These tables contain entries for the most commonly occurring syllables. The converter program looks up these tables to find the target mappings.

2.2 APPROACHES FOR AUTOMATIC GENERATION OF CONVERTERS

An approach that automates the task of creating converters to a reasonable extent has been described in [27]. It uses a page or two of parallel texts in font encoding and the ISCII encoding as input. It then learns the correspondence between glyph codes and ISCII characters. This program has possible glyph grammars that facilitate the learning process. It can also use a glyph table if that is available.

The author in an earlier study obtained experimental results with this approach [28]. This system was tested on rediff.com's hindi news articles (written in Devanaagari script). 133 words were typed in ISCII for a news article written in the Shree708 font. More equivalent ISCII text was produced using the converter generated using these 133 words. For the new characters that appeared manual corrections were made only to generate the input ISCII text. The results using the grammar based approach are as follows:-

Name of font: Shree708
Size of test data: 11000 words
Text used: http://www.rediff.com's news articles

No of word-pairs in training data	Extent of conversion by the converter generated.
100	74%
200	77%
300	79%
400	79%
500	82%
600	83%
700	84%
800	84%
900	84%
1000	84%

Table 8 Conversion done by an earlier grammar based system

The system uses one grammar for all fonts of a given script but the accuracy of the conversion from font encoding to ISCII depends upon the coverage of the grammar. Moreover different grammars are required for different scripts. We propose two new approaches that address these problems to some extent. The first approach improves the accuracy for Devanaagari and Telugu without any parallel text, and the second one provides a script independent solution for similar accuracy.

2.3 TRANSLITERATION ATTEMPTS BETWEEN URDU AND DEVANAAGARI

Attempts for the transliteration from Urdu to Devanaagari have been made in [29] and [30]. The former uses a roman transliteration scheme to store the text

and displays it in Devanaagari and Urdu. The latter is a commercial system from CDAC, India.

Transliteration in the reverse direction i.e. from Devanaagari to Urdu has also been attempted [31].

CHAPTER 3

PROPOSED SOLUTIONS

The systems presented here use two different approaches to address the problem of conversion from font encoding to ISCII/Unicode.

i) Glyph grammar based approach: Here description of each glyph in the font is given using mnemonics. If a glyph has equivalent ISCII/Unicode characters the equivalents are given. Otherwise, a mnemonic is used. Given the description of glyphs using the notation, the system generates converters with an accuracy of over 90% within a span of two hours for the Devanaagari script. For Telugu converters that convert with an accuracy of over 75% for Telugu can be generated over duration longer than 2 hours. This time can be reduced if mnemonics are reused from a font that contains similar glyphs. Some amount of manual editing is then required to make corrections. Although some training is also required for the user to learn the notation, this approach has an advantage that it requires little typing. The information about glyphs can be obtained from a print out of the glyph table and it is not necessary that the fonts be available on one's system.

ii) Machine Learning Approach: The second approach uses Finite State Transducers that encode the correspondence between glyphs and equivalent ISCII/Unicode characters. These can be automatically learnt from a parallel text. The input to these is a parallel list of syllables. The advantage of this compared to the previous approach is that little or no training is required to use the system. The user has to merely give a equivalent ISCII/Unicode syllables for the glyph encoded

ones. Moreover, unlike statistical machine learning approaches the model built by the system is human readable.

3.1 GLYPH GRAMMAR BASED APPROACH

In this method mnemonics are used to list all the smallest possible logical units that a glyph can be a part of. A logical unit is a glyph or a group of glyphs combined, that can be encoded by the Unicode/ISCII encoding system. The glyph ा in the Devanaagari script can be used to denote the vowel sign, part of consonants/conjuncts (ग घ च ज ञ क्ष …) and vowels (अ आ ओ ओ ऑ) and also as a part of other vowel signs (ो ौ). While trying to convert the glyph code for ा to Unicode/ISCII one has to look at the context in which it is used. Without the context it cannot be decided what the glyph ा should be converted to. Similarly in Telugu the glyph ాis used not only as a vowel modifier but also as a part of the consonant ఠ. This problem is similar to the problem of word sense disambiguation. The word 'flies' has a number of meanings but in a given context (e.g. the sparrow flies.) it has only one. With this approach we give the different "senses" of a glyph in different contexts. The user specifies the various logical units a glyph can be a part of. The program uses the context to decide which logical unit the glyph along with its neighbors gets converted to.

3.1.1 Background

As noted earlier (See section 2.3), Indian scripts are compositional syllabic in nature. The older True Type Fonts compose a syllable from its constituent glyphs e.g.

Devanaagari	क्का = क् क ा	ट्ट = ट् ट	ग्वो = ग् व ा ो	्ति = ि ् त
Telugu	క్లో = క ్ ో	ట్టు = ట ్ ు	గ్వో = గ ్ ో	త్తి = త ్ ి

Table 9 A syllable's shape is formed from smaller glyphs joining together

For each glyph (about 200 or so) present in the font we must identify each syllable it can be a part of. However the number of syllables that covers around 99% of the text is above 10,000 and such identification becomes cumbersome. Since syllables have well defined ways of formation, the problem of identifying the larger unit a glyph belongs to, gets reduced as we will see below.

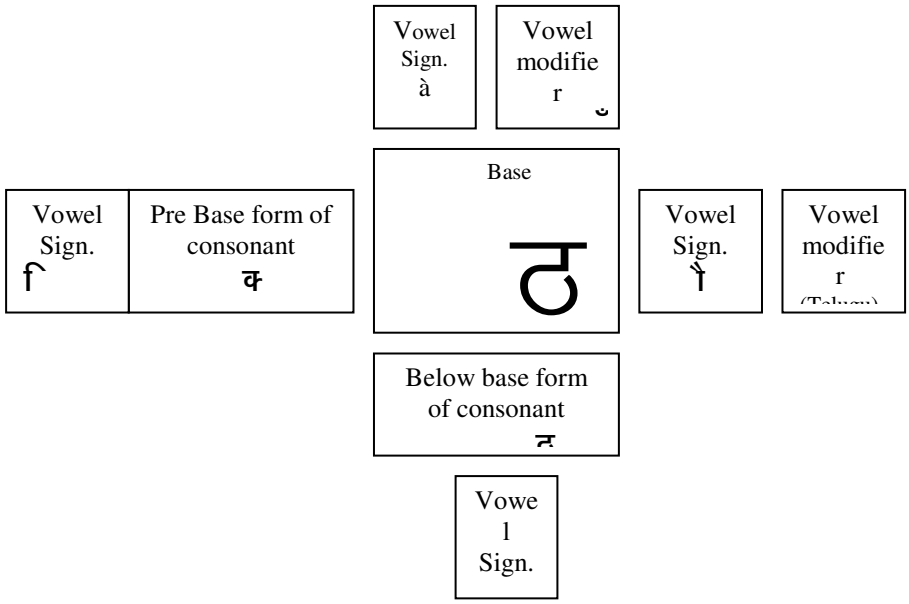

Figure 6 The formation of a syllable in Devanaagari or Telugu script (Can be extended with minor modifications to other Brahmi based Indian scripts). The REPH is treated as a special case and is not included here.

A syllable can thus be broken down into the following parts:-

Vowel Signs.

Base: In Devanaagari the base is usually a consonant or a conjunct such as (ह्य, ह्म). In Telugu the base can also be a part of a consonant which combines with vowel signs and other marks to form syllables e.g. ర glyph is the base in each of ర్, రా, రీ.

Pre base forms of consonants: eg. ె in Telugu.

Below-base forms of consonants: Consonants belonging to the ta class in Devanaagari (ट ठ ड ढ) require the consonant that follows them to appear in the below base form. Almost all consonants in Telugu require the below-base form (when a consonant follows another pure consonant).

Vowel Modifiers: ं ँ in Devanaagari and ంin Telugu.

As a result of the above decomposition of syllable, a glyph need not be described as part of all possible syllables. We can now describe it as a part of one of the above.

3.1.2 Language for Description of Glyphs

A notation is thus proposed for recording each glyph, so that it can be converted to ISCII/Unicode. Different syntax is used for different categories of glyphs. Each of them is discussed in the following sections.

3.1.2.1 All Glyphs Except Those of Below/Post Base Forms of Consonants and Except a Few Others

Three pieces of information are stored along with each glyph:-

For each glyph we identify each logical unit it is a part of. So for ఽ of the Telugu script we specify that it is a part of any one of ఀ ఁ ం ః ఄ అ ఆ ఇ ఈ ఉ ఊ ఋ ఌ ఎ ఏ ఐ ఒ ఓ ఔ. Note that we have not included కా క . This is because we need to record only the smallest logical unit a glyph is a part of. కా ఀ can be considered independent because each has its own ISCII/Unicode equivalent.

Merely specifying the unit to which a glyph belongs is not sufficient. Consider ग and ा of the Devanaagari script which are, amongst others, parts of ग.

When the following glyph sequence occurs ग ा ग ा the program does not know whether they are to be converted to one गा or two or four because all it knows is that the glyphs are part of गा. The problem arises because it does not know how many glyphs must be combined to make the complete unit. We must therefore also specify the number of parts the logical unit has been broken into by the font. This is also required to distinguish it from multiple decompositions (into different number of glyphs) of the same logical unit.

Finally we also record with each glyph the position it goes in the whole. This is required to distinguish one part from another. The name given to the position can be anything, though it should preferably reflect the glyph's position to improve readability.

Here is an example. We have the following description for the glyph ग which can either be the left (L) part of गा of two parts (@2) or the half consonant ग्. i.e. &{L}गा@2|ग्; ('|' is used to separate the alternatives.)

We therefore arrive at the following notation for glyphs:-

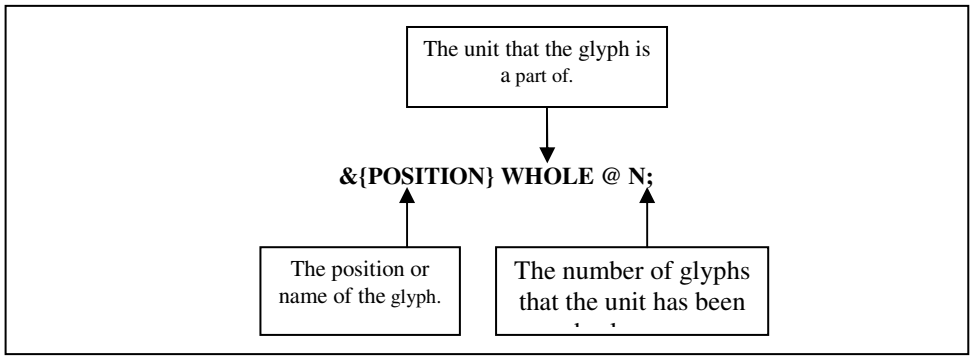

Figure 7 The syntax of mnemonics to be used for describing glyphs. '&', '{', '}', '@', ';' are separators used to identify mnemonics and parts within mnemonics

Except glyphs of below base form of consonants, we found that this encodes all the necessary information for encoding glyphs for converting them to ISCII/Unicode.

Glyph descriptions are specified in a Context Free Grammar [32] which is read by a Java Cup [33] based parser.

3.1.2.2 Glyphs of Below-Base Forms of Consonants.

Glyphs of below/post base forms of consonants are described by surrounding the consonant with angle brackets. For example ङ् is described as < ङ >. The angle brackets are used to distinguish it from the regular/base form of the consonant which in this case is ङ.

3.1.2.3 Glyphs that are Logical Units

If byte code 24 looks like भ then it is recorded with Unicode for भ. If the glyph to be described is a syllable with more than one character then the Unicode

codes of the characters that make up the syllable are given e.g. क्र that is composed of क ् र is described using Unicode codes for them.

3.1.2.4 Combination of Logical Unit and a Glyph

It may so happen that the glyph is a combination of glyphs that have ISCII codes and glyphs which can only be described by mnemonics. In such cases the glyphs are described by combining the descriptions of the logical unit and the glyph, one after another. ಌ glyph of the vaartha [24] font can be described using the mnemonics for ಎ followed by the mnemonics for ಁ

3.1.2.5 Special Cases

The notation correctly and succinctly encodes majority of the glyphs commonly found in fonts for Devanaagari and Telugu. To keep the encoding brief some glyphs have to be given special consideration. A common characteristic of all these special cases is that, the target encoding for the special glyphs does not appear in the position where the glyph appears.

Telugu ు This is encoded as <PreBase-ఁ>. The program then converts it to the Unicode for ఁ and moves it to its appropriate position in the target Unicode. For example consider క్ష. This gets converted to the Unicode for క్ష followed by Unicode for halant and finally Unicode for ఁ.

Devanaagari ि Encoded as < ि >. Although this has a Unicode equivalent describing it, using that will be incorrect. For example consider कि, converting it to Unicode for ि followed by Unicode for क would be incorrect. Describing it as < ि > allows the program to treat it as a special case and move it after the following syllable, in the target encoding.

Devanaagari र् Encoded as <REPH>, because like ि it doesn't appear the position required by the target encoding. The program takes care of the movement to the appropriate position.

Incorporating these into the notation, without treating them as special cases, is possible but at the cost of complicating the description of other glyphs. Since such cases are few in number, we chose to treat them as special cases. For the algorithm to work for other Brahmi based scripts other such cases may to be considered specially.

It may also be noted that the notation is general enough to allow one to not treat the above glyphs as special cases. However one would then have to include the glyphs for these special cases in the logical units whose parts glyphs are described to be. For example if ि were not to be treated as a special case, both ि and क will have to be described also as being parts of कि क would then be described using not just its Unicode equivalent but also as &{RIGHT}कि@2;. The glyph ि would then have to be described as being part of all syllables containing the ि sign. This increases the alternative wholes a glyph can belong to and thus the special treatment.

3.1.3 The Font Glyph Description Table

Using the above notation a file of glyph codes and their description is created. We refer to this file as the font glyph description table. It consists of two columns - byte code and the description separated by a tab stop. Those glyphs that are not marked at all are copied as they are to the output. Most of the fonts contain a glyph for a space that is smaller than the normal space (used for marking word boundaries). The entry for this "glyph" must contain only the code for the glyph. The target is left blank. For example if the small space has code 92 a section of the Font Glyph Description Table would look like:-

.
.
.
24 भ
.
.
.
57 &{L}ग@2।ग्;
.
.
.
92
. .
.
.

Some characters can be grouped into classes and instead of specifying the individual characters everywhere; the description could use the class codes. For

example in Telugu the consonants that have a ‘talakattu’ and in which vowel signs attach by replacing the ‘talakattu’ can be grouped into one class. Consider క గ చ డ త ద న ర ళ వ శ which can be grouped into one class. Instead of giving the description &{T}[క,గ,చ,డ,త,ద,న,ర,ళ,వ,శ]@2; for ్ one could just write &{T}C1@2; where C1 is defined to contain the string [క,గ,చ,డ,త,ద,న,ర,ళ,వ,శ]. A preprocessor then replaces these class codes by the constituent characters.

3.1.4 The Algorithm to Convert a Font Encoded Text to ISCII

Preprocessing: Replace the character class codes by the characters as defined by the user.

Reduction of glyph “senses”: Remove those glyph senses for which the complement glyphs are not found in neighboring positions.

Find all possible mnemonic sequences: Some glyphs may still have multiple senses. Generate all possible mnemonic sequences that the sequence of glyphs can form.

Combine mnemonics that form a logical unit: If all mnemonics in the word combine to yield one or more logical units treat this mnemonic sequence as a valid mnemonic sequence.

Use dictionary to disambiguate: Some glyphs such as ఌ ఎ are problematic because their logical unit cannot be determined from context alone. For example ఌ can be used to make న or ఽ. The difference lies in terms of:-

- which glyph of the identical looking glyphs (talakattu ్ in this case) is used and
- the presence/absence of a small space. (in this case whether uses a small space and talakattu or another talakattu)

- Such information is difficult for the user to provide merely by looking at the glyphs. For such cases where ambiguity remains, use dictionary.

Use map table to disambiguate: Sometimes the dictionary may not contain any of the ambiguous words that a glyph sequence gets converted to or may contain more than one of them. In such cases use the mappings of the glyphs successfully disambiguated so far.

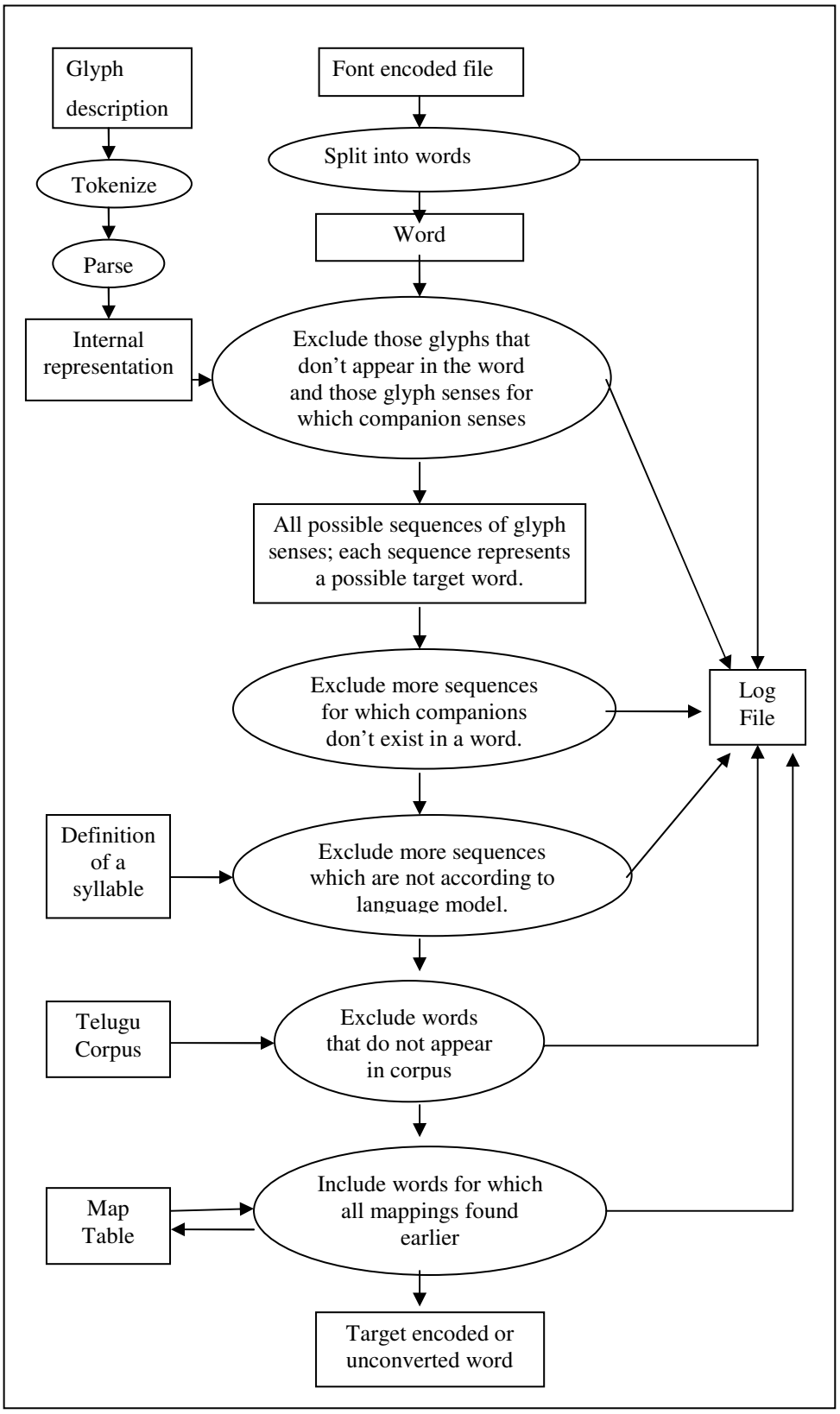

Figure 8 block diagram showing conversion using the glyph description table

To illustrate the algorithm we will convert the word మధ్య to Unicode. This word is composed of the 7 glyphs ఎ ా ధ ◌్ య. The two talakattu's have different glyph codes and are thus different! For conversion to Unicode, each glyph that exists in the font must be present in the font glyph description table (see 3.1.2.1).

The algorithm starts by converting each of these glyphs to their mnemonic equivalents. The mnemonic equivalents are obtained from the font glyph description table.

The glyph	The logical units it can be a part of.
ఎ	ఘు ఘి ఘీ ఘె ఘే ఘ్ ఘై ఘొ ఘో ప పి పీ పె పే ప్ పై ఫ ఫి ఫీ ఫె ఫే ఫ్ ఫై మ మి మీ మె మే మ్ మై మొ మో వ వా పి పీ వె వే వొ వో వౌ వ్ వై హ హా హి హీ హె హే హ్ హై
◌్	క గ ఘు చ ఛ ఝు ఠ డ ఢ త థ ద ధ న ప ఫ భ మ య ర ళ వ శ ష స హ హా
ా	◌ా ఘు ఘి ఘీ ఘె ఘే ఘై ఘొ ఘో ఘ్ ఝు ఝి ఝీ ఝె ఝే ఝై ఝొ ఝో ఝ్ మ మి మీ మె మే మై మొ మో మ్ య యి యీ యె యే యై యొ యో య్
ధ	థ థా థి థీ థె థే థై థొ థో థౌ థ్ ద దా ది దీ దె దే దై దొ దో దౌ ద్ ధ ధా ధి ధీ ధె ధే ధై ధొ ధో ధౌ ధ్
◌్	క గ ఘు చ ఛ ఝు ఠ డ ఢ త థ ద ధ న ప ఫ భ మ య ర ళ వ శ ష స హ హా
◌	థ … and others that contain a dot at the bottom
◌్య	<య>

Table 10 Glyphs in an input word and their possible meanings.

This information is encoded by the font glyph description table as shown in the following table:-

Glyph	The mnemonic based description of the glyph, to encode the logical units it can be a part of.
ఎ	&{B}ఘు@4;|&{B}ఘి@4;|&{B}ఘీ@4;|&{B}ఘె@4;|&{B}ఘే@4; |&{B}ఘ్@4;|&{B}ఘై@5;|&{B}ఘొ@5;|&{B}ఘో@5; |&{B}ప@2;|&{B}పి@2;|&{B}పీ@2;|&{B}పె@2;|&{B}పే@2;|&{B}ప్@2; |&{B}పై@3; |&{B}ఫ@3;|&{B}ఫి@3;|&{B}ఫీ@3;|&{B}ఫె@3;|&{B}ఫే@3;|&{B}ఫ్@3;| &{B}ఫై@4; |&{B}మ@3;|&{B}మి@3;|&{B}మీ@3;|&{B}మె@3;|&{B}మే@3;|&{B}మై@ 4;|&{B}మొ @4;|&{B}మో@4;|&{B}మ్@3; |&{B}వ@2;|&{B}వా@2;|&{B}పి@3;|&{B}పీ@3;|&{B}వె@2;|&{B}వే@2;| &{B}వై @3;|&{B}వొ@2;|&{B}వో@2; |&{B}వౌ@2;|&{B}వ్@2; |&{B}హ@3;|&{B}హా@3;|&{B}హి@3;|&{B}హీ@3;|&{B}హె@3;|&{B}హే @3;|&{B}హ్@3;|&{B}హై@4;
ఁ	&{T}క@2;|&{T}గ@2;|&{T}ఘు@4;|&{T}చ@2;|&{T}ఛ@3;|&{T}ఝు@5;|&{T}ఠ @3;|&{T}డ@2;|&{T}ఢ@3; |&{T}త@2;|&{T}థ@4;|&{T}ద@2;|&{T}ధ@3;|&{T}న@2;|&{T}ప@2;|&{T}ఫ@ 3;|&{T}భ @3;|&{T}మ@3; |&{T}య@4;|&{T}ర@2;|&{T}ళ@2;|&{T}వ@2;|&{T}శ@2;|&{T}ష@2;|&{T}స @2;|&{T}హ@3;|&{T}హా@3;
ఎ	ఎ|&{L}ా@2; |&{R}ఘు@4;|&{R}ఘి@4;|&{R}ఘీ@4;|&{R}ఘె@4;|&{R}ఘే@4;|&{R}ఘై @5;|&{R1}ఘొ@5;|&{R2}ఘొ@5; |&{R}ఘో@5;|&{R}ఘ్@4; |&{R1}ఝు@5;|&{R2}ఝు@5;|&{R1}ఝి@5;|&{R2}ఝి@5;|&{R1}ఝీ@5;| &{R2}ఝీ@5;|&{R1}ఝొ@5;|&{R2}ఝొ@5; |&{R1}ఝే@5;|&{R2}ఝే@5;|&{R1}ఝై@6;|&{R2}ఝై@6;|&{R1}ఝొ@2 ;|&{R2}ఝొ@2;|&{R3}ఝొ@2; |&{R1}ఝో@2;|&{R2}ఝో@2;|&{R1}ఝ్@2;|&{R2}ఝ్@2; |&{R}మ@3;|&{R}మి@3;|&{R}మీ@3;|&{R}మె@3;|&{R}మే@3;|&{R}మొ @2;|&{R}మో@2;|&{R}మ్@2; |&{R1}య@2;|&{R2}య@2;|&{R1}యి@2;|&{R2}యి@2;|&{R1}యీ@2; |&{R2}యీ@2; |&{R1}యె@2; |&{R2}యె@2;|&{R1}యే@2;|&{R1}యే@2;|&{R1}యై@2;|&{R2}యై@2; |&{R1}యొ@2;|&{R2}యొ@2;|&{R3}యొ@2;|&{R1}యో@2;|&{R2}యో @2;|&{R1}య్@2;|&{R2}య్@2;

ధ	&{B}థ@2;|&{B}థా@2;|&{B}థి@2;|&{B}థీ@2;|&{B}థు@2;|&{B}థూ@2; |&{B}థె@2;|&{B}థే@2;|&{B}థై@2; |&{B}థొ@2;|&{B}థో@2;|&{B}థౌ@2;|&{B}థ్@2;|&{B}ద@2;|&{B}దా@2 ;|&{B}ది@2;|&{B}దీ@2;|&{B}దు@2; |&{B}దూ@2;|&{B}దె@2;|&{B}దే@2;|&{B}దై@2;|&{B}దొ@2;|&{B}దో@2 ;|&{B}దౌ@2;|&{B}ద్@2;|&{B}ధ@2; |&{B}ధా@2;|&{B}ధి@2;|&{B}ధీ@2;|&{B}ధు@2;|&{B}ధూ@2;|&{B}ధె@2 ;|&{B}ధే@2;|&{B}ధై@2;|&{B}ధొ@2; |&{B}ధో@2;|&{B}ధౌ@2;|&{B}ధ్@2;
◌	&{D}థ@2; … and others that contain a dot at the bottom
◌	<య>

Table 11 Mnemonic based description of glyphs that appear in the example word.

The first step in the algorithm is to get rid of those mnemonics whose companions don't exist in their vicinity. (Part name B is used for Base, D for Dot, T for Top, L for left, R1, R2, R3 for first, second and third Right parts and R for Right)

Reduction of glyph "senses": Each glyph shown in the first column can potentially be used to form any of the logical units to its right. However, for a given logical unit, not all constituent glyphs will be found in the vicinity. For example although in general the RP can be a part of NRP, here it cannot be because the glyph N does not exist near it.

Glyph	Mnemonics to represent its various usages
ఎ	&{B}ప@2;|&{B}మ@3;|&{B}వ@2;
◌	&{T}ప@2;|&{T}మ@3;|&{T}వ@2;
◌	◌|&{R}మ@3;
ధ	&{B}ధ@3;
◌	&{T}ధ@3;
◌	&{D}ధ@3;
◌	<య>

Table 12 Mnemonics whose companions exist within the example word

Find all possible mnemonic sequences: Having reduced the number of target logical units we can now form all possible words from these:-

&{B}ప@2; &{T}ప@2;ు &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}ప@2; &{T}ప@2; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}ప@2; &{T}మ@3;ు &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}ప@2; &{T}మ@3; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}ప@2; &{T}వ@2;ు &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}ప@2; &{T}వ@2; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}మ@3; &{T}ప@2;ు &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}మ@3; &{T}ప@2; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}మ@3; &{T}మ@3;ు &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}మ@3; &{T}మ@3; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}మ@3; &{T}వ@2;ు &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}మ@3; &{T}వ@2; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}వ@2; &{T}ప@2;ు &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}వ@2; &{T}ప@2; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}వ@2; &{T}మ@3;ు &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}వ@2; &{T}మ@3; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}వ@2; &{T}వ@2;ు &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

&{B}వ@2; &{T}వ@2; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు>

Combine mnemonics that form a logical unit: Each mnemonic sequence is now taken and the mnemonics combined to form logical units. However except for the entries shown in bold the mnemonic sequences do not contain all constituent parts for the logical units. The program therefore selects only those mnemonic sequences which have all the required mnemonics. This leaves us with

&{B}మ@3; &{T}మ@3; &{R}మ@3; &{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు> = వుధ్య

&{B}వ@2; &{T}వ@2;ు&{B}ధ@3; &{D}ధ@3; &{T}ధ@3; <యు> = ము ధ్య

Use dictionary to disambiguate: The dictionary is used next to select one of these. It contains only the second alternative and hence the algorithm successfully converts the glyph sequence ము ధ్య to Unicode.

Use map table to disambiguate: Since no more ambiguity remains the use of map table is unnecessary. The mappings obtained here are also added to the map table so that they can be used later if disambiguation with dictionary does not work with other words.

3.2 MACHINE LEARNING APPROACH

3.2.1 Problems with the Previous Approach:

To describe the glyph grammar one needs not only the knowledge of the script but also the nuances of the script. For example one should know that the right part of क in Devanaagari can also be used to make the right parts of ऊ क्‍ क्त and फ then by looking at the font table, and optionally looking at the display one has to decide on the possibilities allowed in that particular font. In Telugu like scripts it is very difficult to write these grammars since different glyphs with same shape will be provided and the origin metric determines where to place that glyph. Since the origin metric value is not available to the user, it becomes difficult to come up with perfect description.

3.2.2 Machine Learning Techniques

Here we explore the possibility of using machine learning methods for automatic generation of font converters. We have pointed out in section 1.5 the similarity of the problem of font conversion with that of Machine Translation. So we first look at the existing machine learning techniques that are being used for Machine Translation.

Statistical as well as non statistical machine learning techniques are being used for building Machine Translation systems. Statistical techniques try to model the translation problem using probability

P (target sentence | source sentence) = P(target sentence)*P(source sentence | target sentence)

according to the Bayes' rule. The translation that gives the maximum probability for the left hand side is chosen. The probabilities on the right are estimated using different language models such as trigram models [34] or probabilistic context free grammars [35]. Neural Networks as tools for building machine translation systems have been explored, but no realistic systems have been built using them [36]. Finite state translation models have been used in limited domain translation to automatically generalize from a set of examples.

The problem with any statistical machine learning technique is, human being can not 'read/understand' what the machine has learnt. So, in case machine goes wrong, human being will not have 'control' over the system to fix the output. At the same time, the statistical techniques require a huge parallel corpus for training. In case of font converters, a parallel corpus is not readily available.

Non statistical machine learning techniques, on the other hand provide solutions, which are human interpretable. This enables a human not only to 'understand' what machine has learnt, but also 'modify' what machine has learnt. Transducer learning algorithms used by EUTRANS [37]and Brill's rule based tagger [38] are the best examples of non-statistical machine learning algorithms, used in the area of Machine Translation and part of speech tagging respectively, in NLP. Brill's tagger uses a initial set of rules, and a training corpus, and learns through a bootstrapping process. In case of font glyphs, the initial set of rules will be font dependent, and hence this method can not be used for building font converters.

The finite state transducer model, just requires a parallel corpus, and hence seemed to be feasible. One of the advantages of using finite state transducers is that they can be automatically learnt from examples. Moreover these transducers offer a clear advantage over other statistical machine learning techniques, in that the models that they depict are easily interpretable by humans. As a result they are

potentially modifiable to suit one's needs. Moreover domain knowledge about the input or the output can be incorporated to improve their performance.

In the following sections, we describe a transducer learning algorithm, used by EUTRANS. 3.2.5 Contains the terminology used in EUTRANS, the machine translation system whose algorithms have been adapted for our problem. Section 3.2.6 contains an example to illustrate the working of the original algorithm with an example from glyph translation. In section 3.2.7 we point out that the original algorithm needs to be modified to avoid the overgeneralization and suggest modifications.

3.2.3 Using Machine Learning Techniques for Generation of Converters.

Of the different example based machine learning techniques, we used one in which finite state transducers can be automatically learnt from examples. These transducers can be considered as programs that change the input in some way to give the output. Transducer learning algorithms have been used in the EUTRANS project which aims at using example based approaches for the automatic development of machine translation systems for limited domain applications and have been shown to give a low word error rate.

The following example shows the transducer for translating Spanish phrases to English built from 10 examples.

(trescientos, three oh oh),
(seiscientos, six oh oh),
(trescientos diez, three one oh),
(trescientos cincuenta, three five oh),
(trescientos cincuenta y uno, three five one),
(seiscientos cincuenta y siete, six five seven),
(seiscientos ochenta, six eight oh),
(seiscientos ochenta y cuatro, six eight four),
(seiscientos veintitres, six two three)

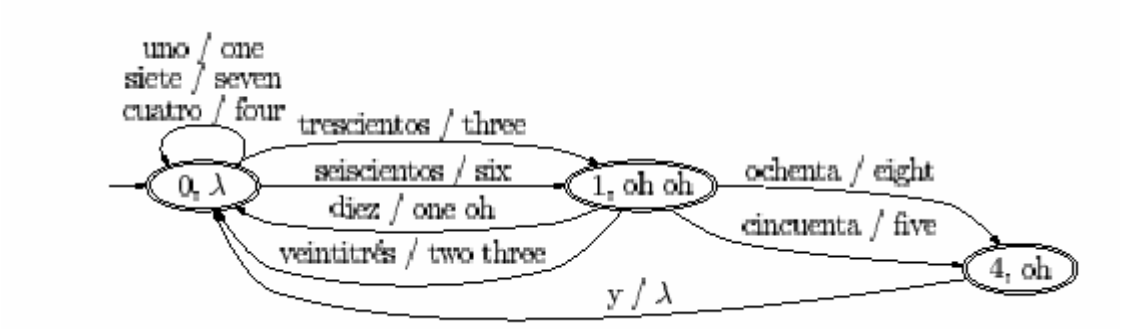

Figure 9 The finite state transducer learnt from 10 example phrases of Spanish to English translation.

3.2.4 Minimum Unit for Translation

In the font conversion problem, the input is the word in the font encoding and output is the word in ISCII/Unicode encoding. However, if we take the analogy of machine translation, a word in font glyphs is analogous to a sentence in a language string, and a syllable in font glyphs is analogous to the word in a language string. Secondly, we observe that the syllables are just concatenated to form words, and their shape is independent of other syllables in the proximity. Hence, the minimum unit of translation in font converters is a syllable, and not a word.

Therefore, in case of font converters training set consist of a parallel corpus consisting of syllables as a sequence of glyphs and their translation into ISCII. Further to reduce the burden of learning, the order of matra and the reph sign (in case of Devanaagari script) are kept the same as that of font glyphs. Since the rules for these transformations are deterministic one can recover this information at a later stage also.

3.2.5 Definitions

A finite state transducer is composed of states and edges connecting those states. Each edge has associated with it an input symbol and an output string. The parsing of an input string begins from a distinguished state (the initial state) and proceeds by consuming input symbols one by one. Every time an input symbol is matched following an adequate edge, the symbol is consumed, the string associated with that edge is output and a new state is reached. This process continues on until the whole input is consumed; then, additional output may be produced from the last state reached in the analysis of the input.

Subsequential finite state transducers are a special class of finite state transducers that satisfy the determinism condition i.e. for a given input string there is at most one valid path and therefore at most one translation

Onward subsequential finite state transducers are a further specialization of subsequential finite state transducers. In these, for each input string prefix, the output string associated to it by the transducer is the longest common prefix of the output strings corresponding to the input strings that begin with this input prefix.

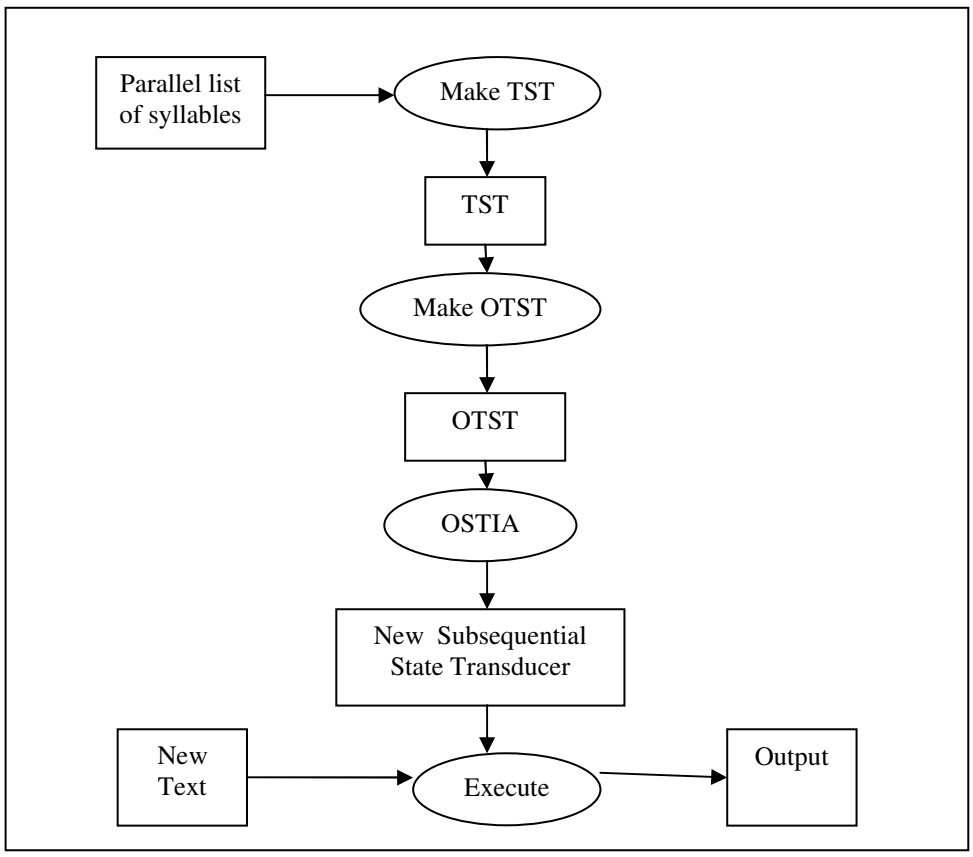

Figure 10 Block diagram of the system that is trained from examples:

In the next section we look at an example to understand the original algorithm and the improvements due to the modification.

3.2.6 An Example to Illustrate the Original Algorithm

Suppose the following examples are given for training:-

Glyphs			ISCII
ग	ा		गा
ग	ा	ं	गां
ग्र	ा		ग्रा
ग	ा	ी	गी
ग	म	ा	गमा
ग	ा	ा	गाा
म	ग	ा	मगा

Table 13 Examples for training used by OSTIA

Stage 1: In the first stage a Subsequential Finite State Transducer (SST) is created in which all outputs are in the final stage. There are no outputs on intermediate states or edges.

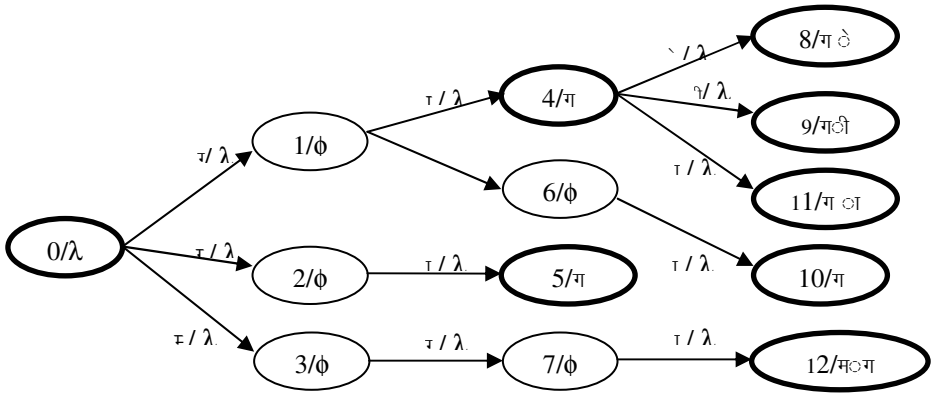

Figure 11 Tree Subsequential Transducer built directly from examples using algorithm MakeTST given in page 72. Final states have darker boundaries. '/' is used to separate input from output. 'ϕ' is the null string. λ is the empty string.

Whenever an input string is seen, the appropriate path on the Transducer is taken and outputs (if any) on the edges are emitted. When the input string is finished outputs (if any) on the current state then, is also emitted. If the current state is not a final state, then the input string is said to be rejected. Consider e.g. the input string गा converted using the transducer in Figure 6. The string consists of two symbols 'ग', 'ा'. When the first symbol is seen the edge from 0 to 1 is followed and state 1 is reached. Since there is no output on the edge no output is given. From state 1, on the next input symbol 'ा' the edge from 1 to 4 is taken and again no output is given. Note that since the input string has not yet finished the output

on state 4 is not emitted. Finally on 'ा' the edge from 4 to 11 is followed. The input string has finished. Therefore the output on 11 i.e. ग ा is given.

Stage 2: In the second stage output strings are pulled towards the start state as much as possible. The SST now becomes Onward Tree Subsequential Transducer (OTST) as is shown in the following figure.

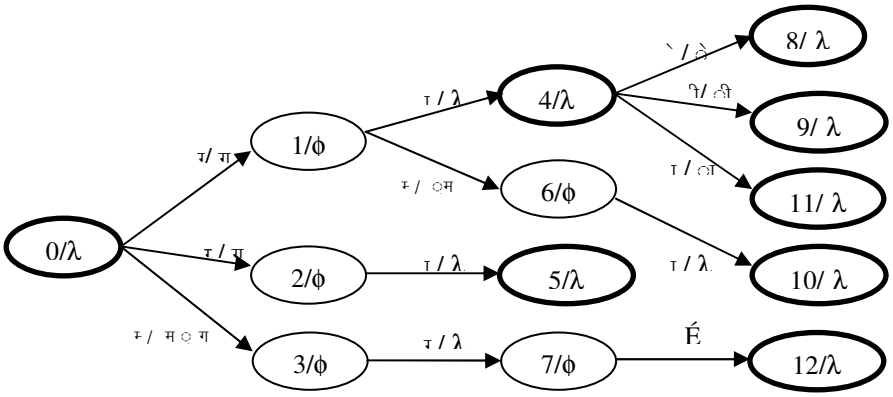

Figure 12 Onward Tree Subsequential Transducer built from Tree Subsequential Transducer using algorithm MakeOTST given in Appendix. The longest common prefixes of the output strings have been moved level by level from the final states towards the root. The transducer translates all and only the examples given in input and no other.

Stage 3: States are now merged one after another. The only property that these states must verify is that all paths departing from them which share the same sequence of input symbols must also share the same sequence of output strings

In order to generalize from the set of examples, the system now starts merging states, trying first to merge 1 into 0. However 0 and 1 have outgoing arcs with the same input symbol (्) but different output symbols (म्ग and ्म). More merging may be required in order to keep the Transducer deterministic, i.e. only one outgoing arc from a state for a given input symbol. See Figure 13.

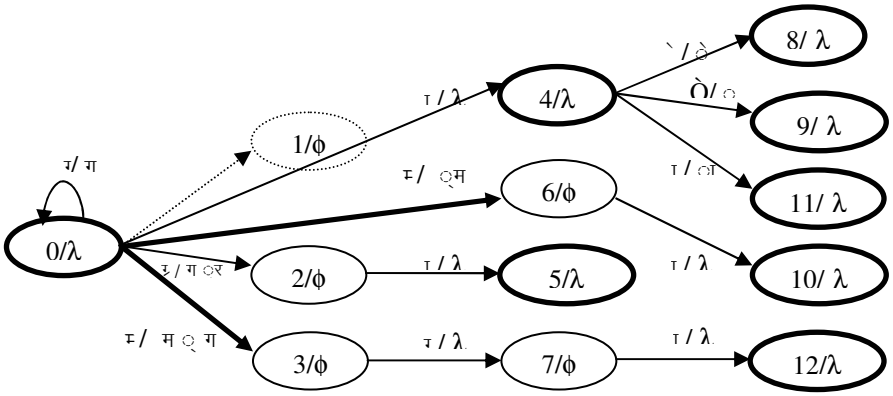

Figure 13 State 1 is being merged into state 0.

The longest common prefix of the two strings मग and म and ग (which is the empty string in this case) is retained on the arc and the remaining is pushed back on the outer edges. The result can be seen in Figure 14.

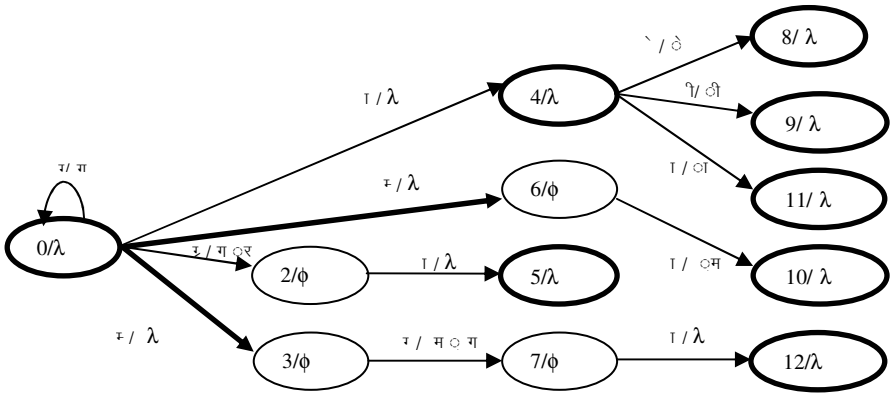

Figure 14 State 1 is being merged into state 0. To make the arcs with the same input symbol, have the same output strings the longest common prefix of the output string is retained and the rest is pushed back. To keep the Transducer deterministic, i.e. only one outgoing arc with a given input symbol, states 3 and 6 also will be merged.

Before finally merging 1 into 0, states 3 and 6 are tried for merging in order to make a single outgoing arc with म. Since they do not have outgoing arcs with common input symbols they can be merged without pushing back of parts of output strings. The final transducer after merging 0 and 1 (and also 3 and 6) is shown in Figure 15.

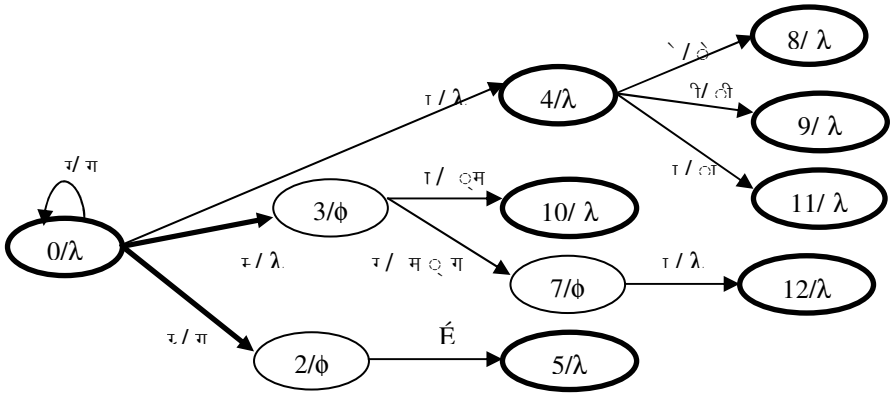

Figure 15 The merging of 0 and 1 is now complete, as a result of merging of 6 into 3. The transducer still converts the examples correctly. However, it has now learnt (incorrectly) how to convert the glyph र into the ISCII code for ग. It should have learnt the ISCII code for ग ्.

As can be seen overgeneralization from examples has taken place. We will see later how to stop this by putting more conditions on when two states can be merged. Merging doesn't stop here and it continues until the following is obtained.

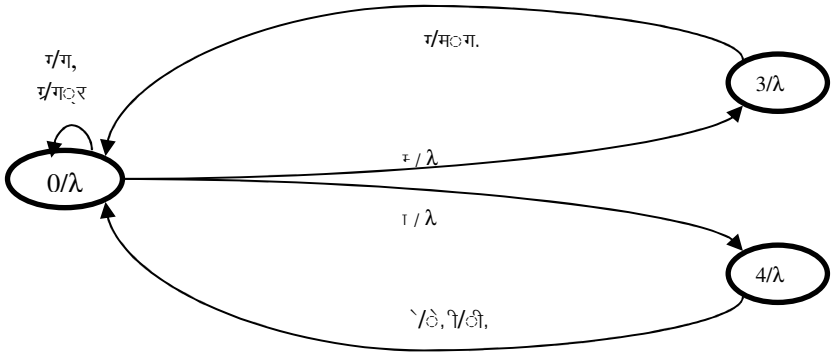

Figure 16 The final transducer obtained as a result of merging all possible states. Looking only at the arcs we see that overgeneralization has taken place, even though the examples are still converted correctly

Given below are some new and incorrect conversions that the transducer in Figure 16 does:-

Glyphs		ISCII
ग		ग
ग्र		ग ् र
म	ग	म ् ग
म	ा	् म

Table 14 Examples of incorrect translations done with the original algorithm

3.2.7 Modification in the Original Algorithm to Prevent Overgeneralization.

If we look at Figure 12 in the previous section, which shows the transducer before 1 was merged into 0, we find that 1 is a non final state and as such the transducer converts ग into ग only when it is followed by a ा. This behavior is correct. The problem arose in Figure 15 because by merging 1 into 0, we made a non final state into a final one. We therefore make a small change to the algorithm so that it doesn't try to merge final states with non final ones. Therefore the condition changes from:-

/* If first state is not final or second state is not final or both have the same output */

If $\sigma'(r) = \phi$ or $\sigma'(s) = \phi$ or $\sigma'(r) = \sigma'(s)$ then

…

To :-

/* If first state is final and second state is final and both have the same output */

If $\sigma'(r) \neq \phi$ or $\sigma'(s) \neq \phi$ or $\sigma'(r) = \sigma'(s)$ then

…

When this change is made to the Merge_States algorithm the following transducer results from the same set of examples:-

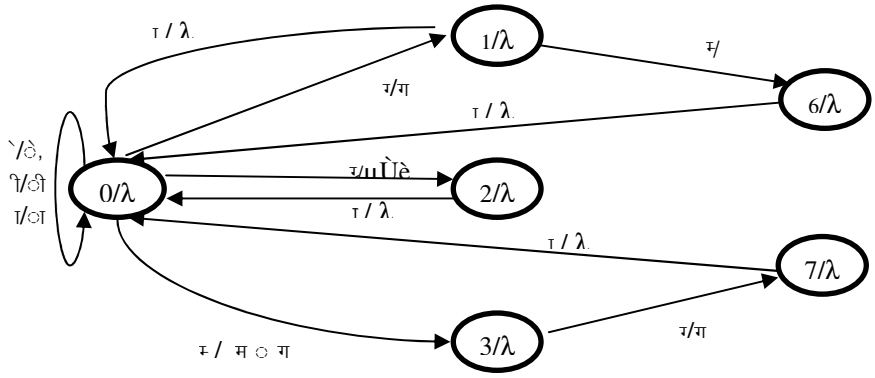

Figure 17 The final transducer obtained as a result of merging all possible "final" states. The algorithm has correctly learnt vowel signs as can be seen from the arc from state 0 to itself

The transducer can now correctly convert vowel signs. Since vowel signs combine with other characters to form new syllables, this learning becomes important.

CHAPTER 4

URDU – HINDI ACCESSOR

4.1 INTRODUCTION

Urdu, as is well known, is one of the official languages in India and has the official language status in the states of Andhra Pradesh and Jammu Kashmir. Its use with Hindi in the spoken form is common in the northern regions of the country and the two languages together are referred to as Hindustani [39].

Rather than saying that Urdu and Hindi are two different languages, they are indeed two different ways of using the same language. Masica [40] points out that they are not even two dialects: they are exactly the same dialect used by two different communities. It is well known that Premchand wrote his stories in Urdu script and got them transcribed into Devanaagari. With the help of this software, now one can access the original Urdu text, without any distortions The differentiation of languages in the mind of public is influenced regrettably by politics and prejudice. Hindi and Urdu together as Hindustani, collectively form the third most populous language (after Chinese and English) in the world. However the use of two different scripts viz Perso-Arabic for Urdu and Devanaagari for Hindi has divided the world of Hindustani into two.

With the advent of technology however, now it is possible to bridge the gap between two scripts. There have been attempts to develop transliteration schemes

(e.g. by GIST group at CDAC and at Columbia University) to access Urdu text through Devanaagari. However Urdu characters that didn't have equivalents in Devanaagari were mapped to the closest Devanaagari alternative. We feel that it is instead necessary to extend the Devanaagari script further to account for special Arabic characters used in Urdu.

Thus we see that it is not an easy task to transliterate an Urdu text into Hindi or vice versa. In what follows, we suggest solution to overcome these differences, and also further steps to reduce the gap between Urdu and Hindi, enabling a Hindi person not knowing Arabic script to access Urdu text through Devanaagari.

4.2 BUILDING A URDU-HINDI ACCESSOR

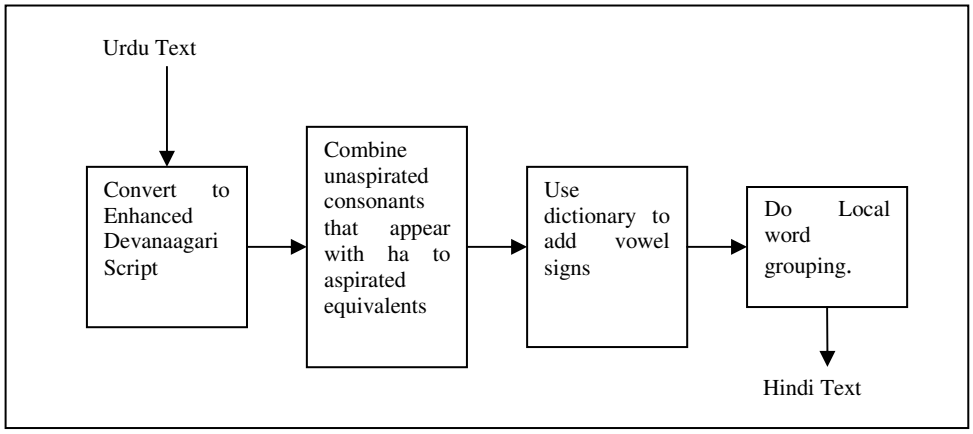

Figure 18 Stages in translation of an Urdu text to Devanaagari

The overall structure of the accessor is shown in Figure 18.

Step 1. Transliteration to Devanaagari

Transliteration to Devanaagari requires a mapping from the Urdu alphabet to the Devanaagari alphabet. However not all characters of the Urdu alphabet have equivalents in Devanaagari. For example The following ظ ض ژ ز ذ ج are variants of

the ja sound and without an Enhanced Devanaagari script will have to be mapped to the single character ज.

The distinction between different ja's is critical since different ja's may give rise to different meanings. In case all the ja's are transliterated to single ja in Devanaagari there will be loss of information, may be leading to catastrophe in understanding. Here are some examples.

Variant of Ja	Devanaagari Unicode Equivalent	Sample Urdu Word	Meaning	Nearest Devanaagari Equivalent of Sample Urdu Word
ز	ज	زلّت	भूल चूक	जिल्लत
ذ		ذلّت	अपमान	
ض		ضلّت	भटक जाना	
ظ		اظہار	प्रकट करना	इजहार
ج		اجہار	जोर से बोलना	
ز		ازہار	दीपक जलाना	

Table 15 Mapping of multiple "ja" sounds of urdu to a single "ja" sound in Devanaagari

As a result where Devanaagari is used to write Urdu (e.g. Urdu Hindi dictionaries) an Enhanced Devanaagari Script is used. Different variants of the script are in use. We use the one from [41] which is also given in the appendix‡

Since most Devanaagari fonts do not contain the new glyphs in the Enhanced Devanaagari Script, we developed a new font to include the additional required glyphs [42]. A filter now maps Urdu characters to enhanced Devanaagari characters.

Step 2. Urdu uses 'ha' to convert a non aspirated consonant into an aspirated one. Devanaagari has special symbols for the non aspirated and aspirated

‡ Unicode has many more characters than are given in 41. Prof. Rahmat Ysuf Zai former head, Dept of Urdu, University of Hyderabad helped us in mapping most (if not all) of them.

ones. A filter is required to convert non aspirated consonants followed by 'ha' to their nearest aspirated equivalents as shown below:-

ग्ह	घ
त्ह	थ
प्ह	फ
च्ह	छ
…	…

Table 16 Filter to convert non aspirated consonants with 'ha' to aspirated

Step 3: The Arabic alphabet is an abjadi (impure). Short vowels are not written, though the long ones are. Urdu, whose script is derived from Arabic, doesn't use vowels or vowel signs in the script. The reader has to determine the presence of vowels from the context. For transliteration to Devanaagari to look natural these vowel signs have to be inserted. Usually there is only one vowel sign that can go in any particular position. A dictionary stores the mappings from Urdu without vowel signs to Devanaagari with vowel signs. There may be more than one possibilities corresponding to a given sequence of consonants. The machine just does a dictionary search and displays all possible combinations.[§] It may be noted that since Urdu often leaves out vowels, even an Urdu native finds it difficult to "guess" the word unless a context is clear.[**]

Some examples of the dictionary entries are:-

§ The dictionary which contains about 1500 such mappings also stores word meanings and is slowly growing. The initial set of word meanings was provided by Prof. M.S. Hayat, Director, Distance Education, University of Hyderabad.

** Even the translations of Premchand's literature by Oxford University Press have mistakes wherein the characters in the novel have been misspelled. e.g. *Suman* as *saman, umanath* as *omanath, balbhadra* as *balbhadar*.

یادگار	यादगार
گڑ	गुड
اس	इस
اس	उस

Table 17 Words without vowel signs in Urdu are translated to words with vowel signs

.

Step 4. Some of the Urdu words are written as two words whereas Hindi writes them as a single. It is necessary that multiple consecutive words that have single equivalents in Hindi be combined. Some such examples are shown below:-

खुद कुशी	खुदकुशी
हमला वर	हमलावर

Table 18 Local word grouping to combine multiple Urdu words into a single Hindi word.

4.3 SAMPLE OUTPUT

The following screenshots show the sample input taken from a Urdu News article on BBC and its conversion to the enhanced Devanaagari script.

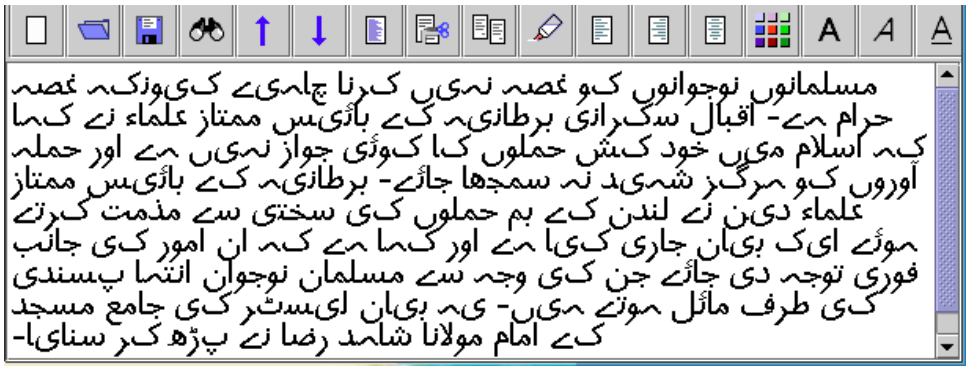

Figure 19 Sample Urdu text taken from the BBC Urdu Website

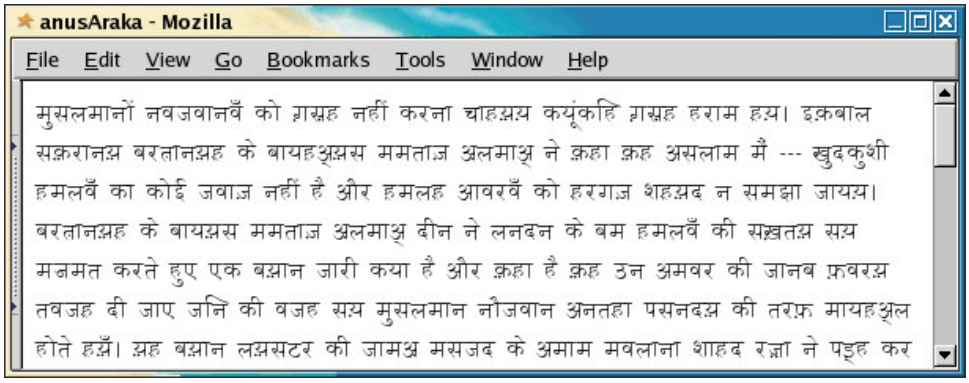

Figure 20 Output of the sample text shown above

The following screenshot shows the step by step conversion from the Urdu to the Enhanced Devanaagari Script (EDS). The first row in each set shows the original Urdu text encoded in Unicode. The last row in each set shows the final Enhanced Devanaagari Script output. The other rows show the intermediate output of conversion. The user interface allows the user to Show/Hide specific rows.

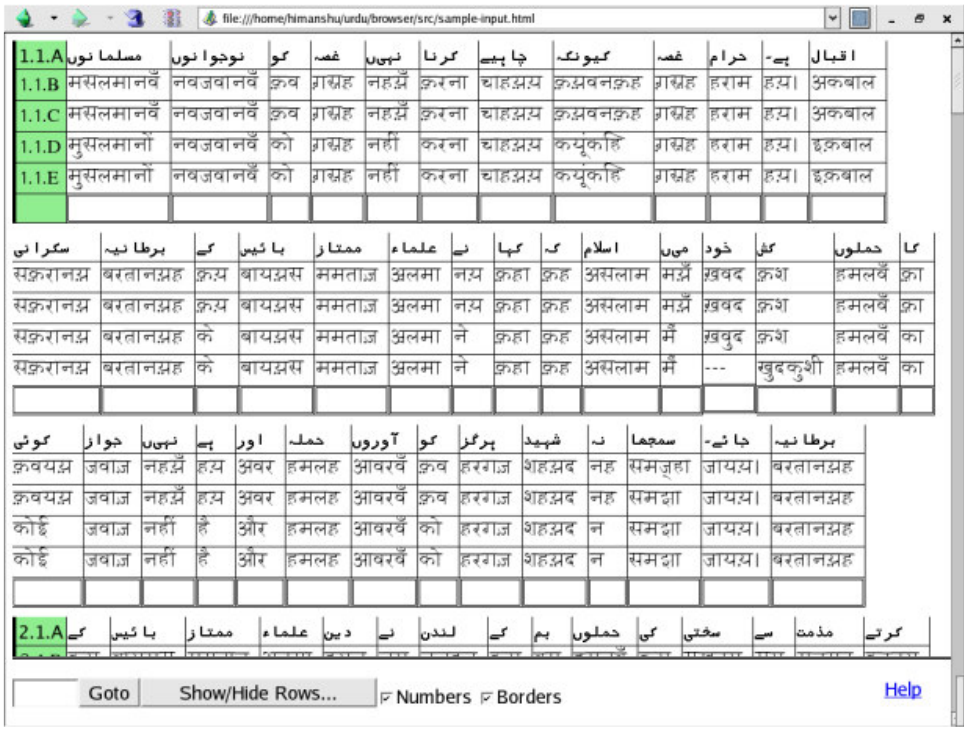

Figure 21 Screenshot of the output of the Urdu-Hindi Anusaaraka in a web browser

The second row contains a character to character conversion from Urdu to EDS. Although not natural but this is still readable. This corresponds to the Step 1 in the conversion process, i.e. Transliteration to Devanaagari.

The third row contains the output after conversion of unaspirated consonants to aspirated ones. See for example the third word from the end where समज्हा is converted to समझा. This is Step 2 of the conversion process.

The fourth row contains the output after insertion of vowel signs or conversion of consonants such as व य to vowel signs. अवर is converted to और, अकबाल to इकबाल and so on. These conversions are done using the dictionary that stores these mappings. The conversion from असलाम to इसलाम could also have been done if the dictionary contained such an entry.

The last row shows the result of local word grouping. The local word grouper in this case has combined the words खुद and कुशी into a single word खुदकुशी. Again an addition of another entry that converts the words हमला and वर to हमलावर would have allowed the program to combine the two into one.

The display of output in stages not only allows the interested user to understand how the output was arrived at but also aids the user or developers to understand why an incorrect output was obtained (if any).

It is important to note that the output is not limited to what we have obtained. The Anusaaraka interface allows one to add more intermediate stages for the purpose of understanding the conversion. More layers can be added between any of the stages mentioned earlier.

4.4 FUTURE WORK

We have thus been able to arrive at a simple system to access Urdu texts through Extended Devanaagari. The only requirement for a full fledged translation from Urdu to Hindi is that of a good coverage dictionary in the electronic format.

CHAPTER 5

RESULTS

5.1 GLYPH GRAMMAR BASED APPROACH

Major chunk of time when using this algorithm is spent in writing down the glyph description table. For scripts such as Devanaagari and Bengali, where a glyph can belong to relatively few logical units, this time is about 2 hours. Once a first draft of the glyph description table is ready it can be given to the program as input along with the text to convert. Often users cannot identify all possible uses for a glyph and such usage is known only when the program leaves some words unconverted. The glyph description table needs to be updated iteratively with the usages not foreseen earlier. Fortunately most of this work can be reused for new fonts of the same script. If the basic glyphs used by the two fonts are same, one merely has to reassign the mnemonic based description to new glyph codes.

Devanaagari: Conversion of over 90% can be obtained after a period of nearly two hours. For the remaining 10% one has to update the font glyph description table. The results were obtained on the jagran, kruti and shusha fonts for Devanaagari.

Telugu: Tests on the eenadu and vaartha [43] font give more than 75% conversion on valid input. The unconverted words can be categorized as follows:-

4% unconverted (updates required to the font glyph description table)

2% too many mnemonic sequences to process. Such cases arise because the number of target logical units for a glyph is too many even after removal of those senses for which companion glyphs are not found nearby.

18% ambiguous, which could not be disambiguated by either the dictionary or the map table.

Devanaagari (jagran/kruti/shusha)	90%
Telugu (eenadu/vaartha)	75%

Figure 22 Results of the grammar based approach in Devanaagari and Telugu

5.1.1 Drawbacks of the Glyph Grammar Based Approach

a) It is difficult to list all possible semantics for a glyph. It becomes necessary to revise the glyph description table with more senses for the glyphs if a word is not converted completely.

b) Some training in understanding the notation (and the nature of script) is required to be able to describe new glyphs.

5.1.2 Possible Improvements

There is currently no support for the user to disambiguate. Such a mechanism will not only disambiguate the word which the user disambiguates but also all other words that are ambiguous because of the same glyphs.

The mappings found are used only at the time of conversion. The program outputs these mappings but there is no support for using these mappings for new conversions.

5.2 THE MACHINE LEARNING APPROACH

The input to the transducer is a parallel list of syllables encoded in the font encoding and the target encoding (ISCII/Unicode). Syllables and not words are required since the performance is higher with syllables. A sample text was taken and 1000 syllables extracted from it. The Unicode equivalents for these syllables were then given. These syllable mappings were given to OSTIA for learning new mappings. The results for Devanaagari and Telugu are shown in the following table:-

Telugu (eenadu/vaartha)	93%
Devanaagari (jagran/kruti/shusha)	98%

Table 19 Conversion of texts when 1000 syllables from a representative sample text are taken

It must be noted that these 1000 syllables used as such for conversion can convert only 75% of text. With new mappings learnt from OSTIA an additional 15% of conversion can be obtained.

The following graph shows the effect of increasing the number of syllables given as input to OSTIA when syllable mappings were obtained for Eenadu to ISCII conversion.

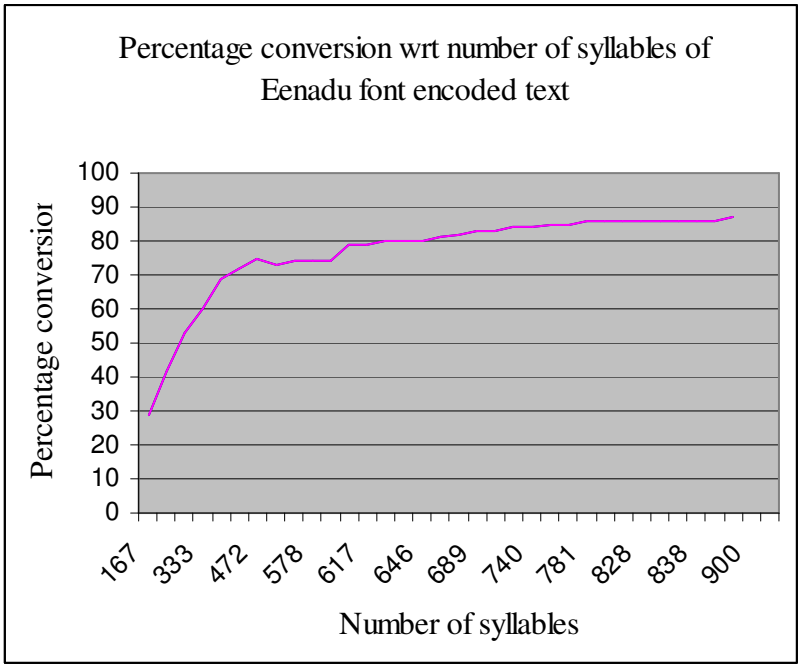

Figure 23 Increase in the percentage conversion on new text with increasing number of syllables.

5.2.1 Drawbacks

One drawback of finite state transducers is that they cannot learn glyph movement. Glyph movement sometimes occurs when the visual position of a glyph is different from the logical position of the character/syllable which it is a part of. One way in which this can be addressed is by moving the characters so that their position coincides with the visual position of their constituent glyphs. Consider the f vowel sign for example. Its logical position is after the consonants but visually it appears before.

Secondly, the examples required for training must be units smaller than whole words. As seen above we had given syllables for training. Although syllable splitting can be automated to some extent, manual splitting is done to minimize errors in examples used for training.

5.2.2 Possible Improvements

Incorporation of domain and range information in the tools: The learning algorithm can incorporate domain and range information, if available. The tool being used doesn't provide this. This information if incorporated is likely to improve accuracy of conversion.

5.3 RESOURCE REQUIREMENTS WITH DIFFERENT APPROACHES

Depending upon the platform, time availability, font availability etc. either of the two approaches could be used. The following table gives a comparison of the two in terms of these factors.

Approach	Operating System/ Platform	Time required for over 90% conversion	Training	Font
Glyph Grammar Based	Java	2 hrs (Devanaagari)	Yes (about 2 hrs or more)	No. A print out of glyph table sufficient.
Machine Learning	C++	8 hrs (Telugu / Devanaagari)	Negligible	Must be installed for splitting to be done manually.

Table 20 Platform, time, training etc. required for font conversion with different approaches

Although the conversion of 90% of font encoded text is less than ideal, it is sufficient to make the output text readable. Moreover a user with little or no programming background can make use of these tools and improve the accuracy with a little more effort when he/she is using the first approach. We have thus been

able to provide a mechanism with which almost any Brahmi based script's font encoded text can be reasonably converted.

One has to now take these tools and help build a repository of font glyph description tables for new fonts of new scripts. This will not only allow conversion of new fonts but also help in improving the tools.

APPENDIX A

URDU-ARABIC-PERSIAN DEVANAAGARI ALPHABET

Unicode Code Point	Unicode Character	Enhanced Devanaagari Equivalent	Unicode Code Point	Unicode Character	Enhanced Devanaagari Equivalent
0625	إ	अ	0630	ذ	ज
0628	ب	ब	0631	ر	र
067E	پ	प	0691	ڑ	ड़
062A	ت	त	0632	ز	ज़
0679	ٹ	ट	0698	ژ	ज
062B	ث	स	0633	س	स
062C	ج	ज	0634	ش	श
0686	چ	च	0635	ص	स
062D	ح	ह	0636	ض	ज़
062E	خ	ख़	0637	ط	त
062F	د	द	0638	ظ	ज़

0688	ڈ	ड	0646	ن	न
0630	ذ	ज	0639	ع	अ
0631	ر	र	063A	غ	ग़
0691	ڑ	ड़	0641	ف	फ़
0632	ز	ज़	0642	ق	क
0698	ژ	ज	0643	ك	क़
0633	س	स	06AF	گ	ग
0634	ش	श	0644	ل	ल
0635	ص	स	0645	م	म
0636	ض	ज	0646	ن	न
0637	ط	त	0648	و	व
0638	ظ	ज	0647	ه	ह
0639	ع	अ	0649	ى	य
063A	غ	ग़	064A	ي	य
0641	ف	फ़	06D2	ے	य़
0642	ق	क	0629	ة	त
0643	ك	क़			
06AF	گ	ग			
0644	ل	ल			
0645	م	म			

APPENDIX B

CONTEXT FREE GRAMMAR FOR GLYPH DESCRIPTION TABLE

Following is the context free grammar for the glyph description table used in the glyph grammar based approach (See 3.1). Details about such grammars and their processing using the Java programming language can be found in [44]

```
glyph_code_to_senses_map
  -> glyph_code_to_senses_map glyph_description NEWLINE
    | glyph_description NEWLINE
    | NEWLINE
    | glyph_description error NEWLINE
    ;

glyph_description
  -> CODE COLON senses
  | CODE COLON
  ;

senses
  -> senses OR similar_senses
  | similar_senses
  ;

similar_senses
  -> similar_senses part
  | part
  ;

part
  -> char_sequence
  | mnemonics
  | CODE
  ;

mnemonics
  -> AMPERSAND LEFT_CURLY_BRACKET char_sequence
RIGHT_CURLY_BRACKET strings:set AT DIGIT SEMICOLON
  ;

strings
  -> strings substring
```

```
  | substring
  ;

substring
  -> char_class
  | char_sequence
  ;

char_class
  ->   LEFT_SQUARE_BRACKET comma_separated_values
RIGHT_SQUARE_BRACKET
  ;

comma_separated_values
  -> comma_separated_values COMMA char_sequence
  | char_sequence
  |
;

char_sequence
  -> char_sequence CHAR
  | char_sequence DIGIT
  | CHAR
  | DIGIT
  ;
```

APPENDIX C

PUBLICATIONS/PRESENTATIONS

[1] Himanshu Garg, “Automatic Generation of Font Converters for Brahmi based Indian Scripts”, The Linguistic Society of India Platinum Jubilee Conference, Hyderabad India 2005

[2] Presented the Urdu-Anusaaraka system at Kendriya Hindi Sansthan during the Rashtriya Sangoshthi on Feb 11 2004

BIBLIOGRAPHY

[1] Daniel Jurafsky and James H. Martin. "Speech and Language Processing", Prentice Hall, 2000

[2] Ide, N. "Encoding Linguistic Corpora", *In Proceedings of the Sixth Workshop on Very Large Corpora*, 1998, pp. 9--17

[3] Squeak: Squeak, http://www.squeak.org

[4] C. V. Jawahar, Million Mesha and A. Balasubramanian, "Searching in Document Images", *In Proceedings of the Indian Conference on Vision, Graphics and Image Processing (ICVGIP)*, Dec. 2004, Calcutta, India, pp. 622--627

[5] MNSSK Pavan Kumar, S. S. Ravikiran, Abhishek Nayani, C. V. Jawahar and P.J. Narayanan, "Tools for Developing OCRs for Indian Scripts", Proceedings of the Workshop on Document Image Analysis and Retrieval, June 2003, Madison,

[6] Bharati, Akshar, Amba P Kulkarni, Vineet Chaitanya and Rajeev Sangal, "अनुवाद के उपकरण संगणक तथा भाषाएं" (Tools of Translation: Computer and Languages), University of Hyderabad, Distance Education Programme, Hyderabad, India, Feb 1998.

[7] Indian Script Code for Information Interchange – ISCII, UDC 681.3, Bureau of Indian Standards, New Delhi, India, 1991

[8] Holmes, N. "The problem with Unicode," Computer Volume 36, Issue 6, pp. 116, 114 – 115, June 2003

[9] Holmes,N. "Toward decent text encoding," Computer Volume 31, Issue 8, pp. 108 – 109, August 1998

[10] Mudawwar, M.F. "Multicode: a truly multilingual approach to text encoding," Computer Volume 30, Issue 4, pp. 37 – 43, April 1997

[11] "ISFOC Standard for Fonts," http://www.cdac.in/html/gist/standard/isfoc.asp

[12] "ILeap – Indian Language Word processor", http://www.cdac.in/html/gist/products/ileap.asp

[13] Linux Technology Development for Indian Languages, http://www.cse.iitk.ac.in/users/isciig/

[14] Tom Mitchell, "Machine learning", Mc Graw Hill, 1997

[15] Microsoft Typography – Features of TrueType and OpenType, http://www.microsoft.com/typography/SpecificationsOverview.mspx

[16] Can legacy encoded Khmer text be converted to Khmer Unicode ? http://www.bauhahnm.clara.net/Khmer/Welcome.html#LEGACYTOUNICODE

[17] STED – SARA Transliterator and Editor, http://sted.sourceforge.net

[18] SILConverters 2.1 http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=EncCnvtrs

[19] International Components for Unicode, http://www-306.ibm.com/software/globalization/icu/index.jsp

[20] Brian W. Kernighan and Dennis M. Ritchie, "The C Programming Language", 2nd ed. Prentice Hall Inc., 1988.

[21] Bjarne Stroustrup "The C++ Programming Language", 3rd ed. Addison-Wesley, 1997

[22] Arnold, K., and Gosling, J. "The Java Programming Language", 2nd ed. Addison-Wesley, Reading, MA, USA, 1998

[23] Jonathan Kew, "Beyond UTF22: complex legacy-to-Unicode mappings," *In 22nd International Unicode Conference*, September 2002, San Jose, California

[24] Steven Johnson C., "Yacc: Yet Another Compiler Compiler", {UNIX} Programmer's Manual, Volume 2, Holt, Rinehart, and Winston, New York, NY, USA pp. 353 - 387, 1979

[25] padma, http://padma.mozdev.org/

[26] Font Converters, http://ltrc.iiit.ac.in/showfile.php?filename=downloads/FC-1.0/fc.html

[27] Bharati, Akshar, Nisha Sangal, Vineet Chaitanya, Amba P. Kulkarni, and Rajeev Sangal, "Generating Converters between Fonts Semi-automatically", *In Proceedings of SAARC conference on Multi-lingual and Multi-media Information Technology*, September 1998, CDAC, Pune, India

[28] Himanshu Garg, "Generating converters between fonts semi-automatically," B.Tech. Thesis, 2002, IIIT, Hyderabad, India

[29] THE 'PLAIN ROMAN' TRANSLITERATION SYSTEM, http://www.columbia.edu/itc/mealac/pritchett/00ghalib/about/txt_translit.html?

[30] CDAC: GIST - Products – Nashir http://www.cdacindia.com/html/gist/products/nashir.asp

[31] Hindi to Urdu Transliterator http://www.crulp.org/h2utransliterator.html

[32] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools"
Addison-Wesley Pub Co. 1986

[33] Hudson, S. E., Flannery, F., Ananian, C. S., Wang, D., Appel, A. W., "CUP Parser Generator for Java," http://www.cs.princeton.edu/ ~appel/modern/java/CUP/

[34] Peter F. Brown, Jennifer C. Lai, and Robert L. Mercer, "Aligning Sentences in Parallel Corpora," *In Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, 1991, University of California, Berkeley, CA, pp. 169-176,

[35] Barnett J., Cant J., Demedts A., Dietzel T., Gates B., Hays E., Ito Y., and Yamron J., "LINGSTAT: State of the System," *In ARPA Workshop on Machine Translation*, November 1994, Vienna, Virginia,.

[36] Bonnie Dorr, Pamela Jordan, and John Benoit. "A Survey of Current Research in Machine Translation," Advances in Computers, Volume 49, M. Zelkowitz (Ed), Academic Press, London, pp. 1-68, 1999

[37] Amengual J.C., Benedi J.M., Casacuberta F., Castano A., Castellanos A., Jimenez V.M., Llorens D., Marzal A., Pastor M., Prat F., Vidal E., Vilar J.M, "EUTRANS-I Speech Translation System, The," Machine Translation, Volume 15, pp. 75-103, 2000

[38] Eric Brill, "Some Advances in Transformation Based Part of Speech Tagging," *In Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994

[39] Urdu Language, http://en.wikipedia.org/wiki/Urdu_language

[40] Masica, Colin P., "The Indo-Aryan Languages," Cambridge University Press, 1991

[41] "Urdu Hindi Bhuvat", Bhuvanvaani Trust, Lucknow

[42] FontForge: An outline font editor for creating/editing fonts, http://fontforge.sourceforge.net/

[43] http://www.vaarttha.com/font/vaartha.ttf

[44] Andrew, Appel W., "Modern Compiler Implementation in Java" Cambridge University Press, 1998