A Generic Spell Checker Engine for South Asian Languages

Arif Billah Al-Mahmud Abdullah,
3rd semester student, Computer Science and Engineering,
BRAC University, Dhaka, Bangladesh
E-Mail : proshno@bracuniversity.ac.bd
&
Ashfaq Rahman,
Graduate Student, Computer Science Department, Fu Foundation School of Engineering & Applied Science,
Columbia University, NY, USA.
Lecturer, Computer Science & Engineering, BRAC University, Dhaka, Bangladesh
E-Mail : ashfaqr93@hotmail.com

Abstract :

Now-a-days, spell checker has become an essential part of any application software. Today, undoubtedly it can be acknowledged that Microsoft Office is the most widely used software bundle all over the world. But unfortunately it provides the spell checking tool only for some limited languages. South Asian languages are deprived more in this case though more than 100 million people live in this region and use their very rich local languages like Bangla, Hindi, Tamil etc. These languages are far different from Western languages in phonetic properties and grammatical rules. This is why the existing algorithms and techniques that are being used to check the spelling and to generate efficient suggestions for miss-spelt words of English and other Western languages are not actually suitable for South Asian languages; rather they need different algorithms and techniques for expected efficiency. This paper is an approach to present a new algorithm based on the existing algorithms to generate suggestions for miss-spelt words along with a complete design and implementation of a spell checker which is much efficient for South Asian languages.

***Index Terms*--Automation Client, Automation Server, Circular List, COM, CSAPI, Double MetaPhone, EditDistance, Hashing, MetaPhone, Microsoft Office Automation, SoundEx, Recursive Simulation.**

I. INTRODUCTION

Microsoft would have provided an API named *CSAPI* (*Common Speller API*) for international developers to create the spell checker for their local language. Using *CSAPI,* developers could easily develop MS-Office enabled spell checking utility. But, this API was open till Office 97. After this version of MS-Office i.e. since MS-Office 2000, Microsoft has stopped the free distribution of any kind of tool or API by which one could develop any customized spell checker for MS-Office [7]. Under these circumstances, the need for a useful and a practical technique and a method to develop the generic spell checking utility for MS-Office began to appear stronger and stronger day by day all over the world. But still there are very few open source solutions for this greatly desired tool.

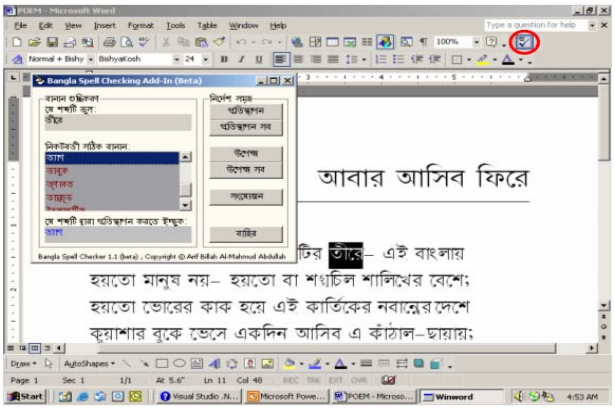

Fig 1: Spell Checker Add-In

II. CREATING A SPELL CHECKER USING STAND-ALONE APPLICATION VS. ADD-IN COMPONENT

The I/O basic of a spell checker for MS-Office is *Microsoft Office Automation.* The whole process of *Office Automation* is performed by two distinct modules; these are - *Automation Server* and *Automation Client. Automation Server* is the host office application (e.g. Word or Excel) upon which different kinds of external actions are imposed and *Automation Client* is the external software development tools (e.g. Visual C++ or Borland C++ Builder) which develop and control the actions to be exposed on to *Automation Server*. It implies that the spell checking utility works as an *Automation Client* by fetching the words or phrases from the Word documents, Excel work-sheets etc (i.e. *Automation Server*), identifying the incorrectly spelt word and eventually replacing the miss spelt word by a suggested correct word [7]. *Office Automation* applications can be both stand-alone software and embedded add-in components. Creating Add-In is comparatively more complex and time consuming than that of a stand-alone software. Yet we selected add-in because it is definitely important to highlight the miss spelt word (i.e. blocking that word) while showing the suggestion dialog-box for the incorrect word on the screen. Highlighting the incorrect word and displaying the appropriate suggestions in the dialog-box may not be simultaneously possible by stand-alone or external applications. But it can be easily implemented by using the Add-In component. Hence the reason for choosing add-in component for

this utility is to ensure the ease and effectiveness of use of the spell checker.

III. Steps and Tasks in the Development of a Spell Checker

Before starting the elaborate discussion about the Spell Checker Add-In, it is necessary to understand two important aspects of a spell checker :

A. *Generation of the Binary Dictionary File from Standard Database*

At first, it is necessary to store the whole word list in a standard database table of two fields. One will contain the word string and another will remain empty for the time being. Later on, a program named **Weight Generator** will generate numeric weight using customized *Hashing* algorithm for each word in such a way that after generating the weight for each word in the database, if we sort out the word list according the weight field in ascending order, the words will also be sorted just as in the dictionary. After sorting, a simple binary file containing all the words will be generated using the word database along with their corresponding weight. This binary file will be distributed as the main dictionary file.

B. *Registering the Spell Checker Add-In Component*

Before using the Spell Checker, the Add-In component must be registered in the system registry of Windows.

IV. Structure of the Spell Chekcer Add-In

In order to build up a complete Spell Checker Add-In for MS-Office, two distinct steps must be completed. These are:

A. *Creation of a General Add-In for the Customized Spell Checker*

Here in this step, an ordinary add-in object which will appear in a new tool bar is to be created. This tool bar will contain a button indicating the customized Spell Checker utility of the Office application. To create such an add-in, three indispensable interfaces as following must be manipulated:

1. *Extensibility Interface*

Microsoft provides an extensibility interface from MS-Office 2000 for its add-in components in order to connect to and disconnect from any Office host application. Using this new extensibility interface in Office 2000, any MS-Office enabled add-in solution with any programming language that can create *COM* (*Component Object Model*) components may be developed. This extensibility interface is available as *IDT_Extensibility2* in the Microsoft Add-In Designer (MSADDNDR.TLB).

2. *COM Interface*

In fact, all the Office add-ins are a kind of specialized COM objects. A COM component is an in-process COM server (DLL) which can run within the context of one or more Windows based applications [7]. COM add-in provides a flexible, efficient and uniform method of extending the MS-Office 2000 environment. In order to create the COM component, the interface called *IClassFactory* can be used.

3. *User Interface*

The user activates the Spell Checker by means of this interface. It determines when user wants to check out his/her document by raising the OnClick event of the Add-In. The interface named *IDispatch* is very useful to handle the user interface part of the Spell Checker add-in.

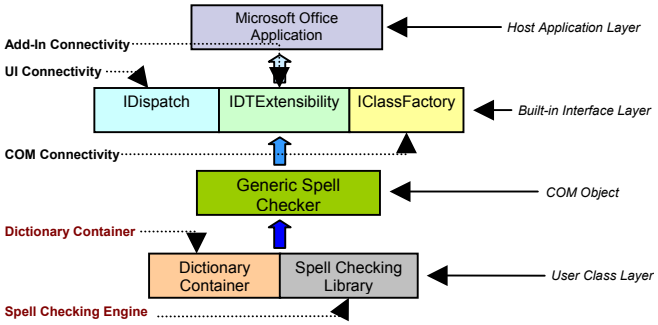

Fig 2: Bangla Spell Checker Add-In Architecture

B. *Attaching the Word Manipulation Module with the AddIn*

The 'Word Manipulation Module' is the source code which will be used as the processing part of the Spell Checker. It consists of the functions and libraries to accomplish the following tasks:

1. *Loading the dictionary database*

Here, the whole word list (both the words and their corresponding weights) from both the main and the custom dictionaries will be loaded in the linked lists.

2. *Fetching the word*

In this part, words will be fetched one by one from the currently open document. To perform this function effectively, the list of delimiting characters should be stored previously so that the proper starting and ending position of any word may be determined precisely while fetching, comparing each character of the document with that list. Because the character sets vary from language to language, the delimeters of different languages may also vary.

3. *Matching the word*

Here, the fetched word is checked for matching whether it exists either in the main or the custom dictionary. Binary search method is applied for the matching because it is assumed that the word list is sorted.

4. *Generating the appropriate suggestions for the miss spelt word*

This function is to be called when the fetched word exists neither in the main nor in the custom dictionary. To gather the appropriate suggestions, *SoundEx, MataPhone, Double MetaPhone* or other algorithms can be applied. *MetaPhone* or *Double MetaPhone* algorithm manipulates each letter of the alphabet in order to code a word phonetically by reducing it to a certain number consonants sounds [2,3]. Undoubtedly it is an excellent method for reducing matching problems from wrong spelling. But these algorithms have to work with all the letters of the alphabet separately to simulate all cases of spelling. Therefore, these algorithms become very large and complex. For English, French, German, Greek and some other languages having few letters (20 - 30) in the alphabet, these algorithms are quite enough to generate

appropriate suggestions for miss spelt words. But some other languages like Bangla, Hindi, Tamil (i.e. generally the South Asian languages) etc have much bigger set of alphabet.

Consider, Bangla language. The alphabet consists of 50 letters (12 vowels and 48 consonants!). Moreover there are 10 vowel-symbols and 3 consonant-symbols and many more compound consonants (more than 150!). Just think, if *MetaPhone* or *Double MetaPhone* algorithm is implemented for getting suggestions against any miss-spelt word of Bangla language, how complex and large it will be!
So, it is quite clear that application of *MetaPhone* or *Double MetaPhone* algorithms will become extremely complex and difficult for the Bangla language. Besides, these algorithms will not be as efficient as they will be for English and other Western languages. Because in Bangla alphabet set, every letter has its own and completely distinct phonetic characteristics. In a Bangla word, one letter usually never influences the phonetic properties of other letters of that word. But in English it occurs in many cases. For example, in the word EDGE, the spelling of the last 3 letters is like 'J' i.e.

EDGE→E+(DGE)→E+J→EJ
So, the spelling of EDGE and EJ are similar.
In the same way,

PHONE→(PH)+(ONE)→F+ON→FON
So, the spelling of PHONE and FON are similar.

XEROX→X+ER+OX→Z+ER+OKS→ZEROKS
So, the spelling of XEROX and ZEROKS are similar.

In Bangla language, these types of similarities are seldom seen. Rather, in Bangla, there are some set of letters which are spelt similarly but do not influence one another. For example:

Set of letters	Spelling
R, j, h, i	Between 'S' and 'SH'
S, e	Between 'JA' and 'JO'
F, I, m	Between 'RA' and 'RO'
P, p	Similar to 'ANG'
A, n	Between to 'AA' and 'AO'
C, D	Similar to 'EE'
E, F	Between 'UO' and 'YO'
Z, _	Between to 'NA' and 'NO'

Table 1: List of phonetically similar letters

There are also some vowel-symbols. Some of these symbolic forms of vowels are used before the main letters and some others are used after the letters. They are as following:

Main Vowel	Symbolic Form	Spelling	Position
B	◌া	Similar to 'AA'	After the letter
C	◌ি	Similar to 'EE'	Before the letter
D	◌ী	Similar to 'EE'	After the letter
E		Similar to 'UO'	After the letter
F		Similar to 'UO'	After the letter
G		Similar to 'REE'	After the letter
H	ে	Similar to 'E'	Before the letter
I	ৈ	Similar to 'OY'	Before the letter
J	ো	Similar to 'O'	Both sides of the letter
K	ো ে	Similar to 'OU'	Both sides of the letter

Table 2: List of vowel-symbols (known as *kaar*)

Likewise there are few consonant-symbols which act exactly like vowel-symbols. These are shown below:

Main Consonant	Symbolic Form	Spelling	Position
E		Similar to 'JA'	After the letter
F		Similar to 'RA'	After the letter
F		Similar to 'RA'	Before the letter

Table 3: List of consonant-symbols (known as *folaa & reff*)

Except these, there are more than 150 compound letters formed by joining two or more consonants. Some of them are as following:

Compound Letter	Formation of the Letter	Spelling
f	L + i	Similar to 'KHA'
ঙ	P + N	Similar to 'ONGO'
~	L + [	Similar to 'OKTO'
‡	_ + [	Similar to 'ONTO'
o	i + V	Similar to 'OSHTO'
»	i + W	Similar to 'OSHTHO'

Table 4: Some compound letters and their formation

Now, in the following table some very commonly used Bangla words and their miss-spelt forms are mentioned for further analysis:

Correct Word	Miss-spelt Form of the Word	Pronunciation of both Correct & Incorrect Forms of the Word
LvNS	LvNe	KAAGOJ
hvjL	hvhL, jvhL, jvjL	SHAASHOK
Svi[in	evi[in, evwi[n, evi[iA, evwi[A, Svwi[n, Svi[iA, Svwi[A	JAATIYO
Aph	Api, Apj, APh, API, APj	ANGSHO
Dio	Cio, Ci[, Cjo, Cho, Cj[, Ch[, Djo, Dho, Dj[, Dh[	EESHOT
h'Ì	j'Ì, jòÌ, jÿÌ, hòÌ, hÿÌ, i'Ì, iòÌ, iÿÌ	SHOOSHKO

Table 4: Some common Bangla words with their miss-spelt forms

From the data mentioned in the above tables, it can be seen that in Bangla language, every single word may have several numbers of analogous phonetic representations and each representation creates its own pronunciation strictly according to its elementary letters and symbols. No word violates the phonetic properties of its constructing letters. But in English words, it is frequently seen that the pronunciation of the word is much different from the usual phonetic properties of its elementary letters. This is the most prominent difference in spelling between the Western and South Asian languages. Since *MetaPhone* and *Double MetaPhone* algorithms manipulate each letter of the alphabet and work mainly on the pronunciation of the syllabi of any word, they are suitable for detecting suggestions in Western languages which have smaller alphabet of less complexity. But South Asian languages have much bigger alphabet that are also extremely complex. Here, in order to give appropriate suggestions, it is necessary to simulate various

represent-tations of the miss-spelt word on the basis of the sets of similarly spelt letters, vowel-symbols, consonant-symbols and compound letters. But, these algorithms are not designed to satisfy all these requirements. Hence, the algorithms like *MetaPhone* and *Double MetaPhone* are not very suitable and efficient for South Asian and many other languages.

Under these circumstances, a new algorithm which can satisfy all the mentioned criteria and reduce the complexity of implementation of searching the nearest suggestions for the miss-spelt word in Bangla as well as other South Asian languages is needed. Here, an algorithm which may solve this problem partially has been proposed. This algorithm is named ***RecursiveSimulation*** which is still under research and development. In this algorithm, a set of circular lists has been used. Each of the lists consists of letters that phonetically similar. For example, some lists are mentioned below:

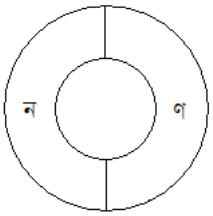

Fig 3.1: Circular list of the letters spelt as 'NA'

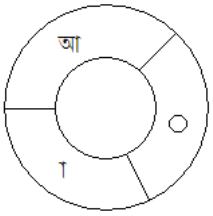

Fig 3.2: Circular list of the letters spelt as 'AA' (here 'O' means null).

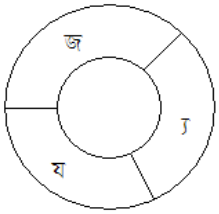

Fig 3.3: Circular list of the letters spelt as 'JA'.

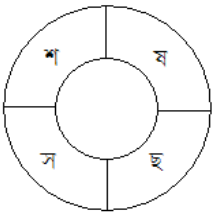

Fig 3.4: Circular list of the letters spelt as 'SA' and 'SHA'

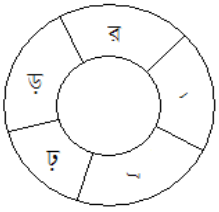

Fig 3.1: Circular list of the letters spelt as 'RA'.

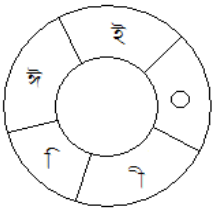

Fig 3.5: Circular list of the letters spelt as 'EE' (here 'O' means null).

If in the miss-spelt word, any letter exists in any of the lists, then that letter will be replaced by all other letters of the corresponding list one by one and then will be checked for a match in the dictionary. When any null character will be found in the list (fig 3.3), then a word representation will be formed where the position of that letter will be simply ignored.

Algorithm 1: **RecursiveSimulation**

Input: W, the miss-spelt word
S, empty array for storing the suggestions
P, current position of the character in the miss-spelt word (default value is the length if W and all external procedures always pass this value)

Output: S, array containing the found suggestions

1. **RecursiveSimulation (W, S, P)**
2. C ← P^{th} character of W
3. L ← number of letters in alphabet that are phonetically similar to C
4. T ← W
5. *while* L > 0 *do*
6. M ← empty string

```
7.        ReplaceLetter(T, M, P)
8.        if M exists in the dictionary and M does not exist in S then
9.            append M to S
10.       if P > 1 then RecursiveSimulation (M, S, P – 1)
11.       T ← M
12.       L ← L – 1
13.   end
14.   if P = length of W and P > 1 then do
15.       W ← leftmost (P – 1) characters of W
16.       P ← P - 1
17.       while P^th character of W is any vowel-symbol or any consonant-symbol and P>0
18.       do
19.           W ← leftmost (P – 1) characters of W
20.           P ← P – 1
21.       end
22.       RecursiveSimulation (W, S, P)
23.   end

1.    ReplaceLetter(T, M, P)
2.    M ← leftmost (P – 1) characters of T
3.    L ← P^th character of T
4.    N ← next phonetically similar character to L
5.    if N is a vowel-symbol then do
6.        if N is grammatically used before the letter then
7.            if L is a vowel-symbol then
8.                if L is grammatically used before the letter then
9.                    append  N to M
10.               else
11.                   if P > 1 then insert N at (P – 1)^th position of M
12.           else
13.               if P > 1 then insert N at (P – 1)^th position of M
14.       else
15.           if L is a vowel-symbol then
16.               if L is grammatically used before the letter then
17.                   if P < length of T then do
18.                       P ← P + 1
19.                       append  P^th character of T to M
20.                       append  N to M
21.                   end
22.               else
23.                   append N to M
24.           else
25.               append N to M
26.   end
27.   else if N is a vowel then do
28.       if L is a vowel-symbol then
29.           if L is grammatically used before the letter then
30.               if P < length of T then do
31.                   P ← P + 1
32.                   append  P^th character of T to M
33.                   append  N to M
34.               end
35.           else
36.               append N to M
37.       else
38.           append N to M
39.   end
40.   else
41.       append N to M
```

This algorithm will simulate a suggestion list as following for the word 'Åhdå':

Primary simulation tree:

```
Åhdå
+---Åhd
!       +---Åjd
!       !       !---jÅd
!       !       !
!       !       !---jd (feasible)
!       !       !
!       !       !---jCd
!       !       !
!       !       !---jDd
!       !       !
!       !       !---Åjd
!       !
!       +---Åid
!       !       !---iÅd
!       !       !
!       !       !---id
!       !       !
!       !       !---iCd
!       !       !
!       !       !---iDd
!       !       !
!       !       !---Åid
!       !
!       +---ÅRd
!       !       !---RÅd
!       !       !
!       !       !---Rd
!       !       !
!       !       !---RCd
!       !       !
!       !       !---RDd
!       !       !
!       !       !---ÅRd
!       !
!       +---Åhd
!               !---hÅd
!               !
!               !---hd (feasible)
!               !
!               !---hCd
!               !
!               !---hDd
!               !
!               !---Åhd (feasible)
!
+---ÅhdB
!       +---ÅjdB
!       !       !---jÅdB
!       !       !
!       !       !---jdB
!       !       !
!       !       !---jCdB
!       !       !
!       !       !---jDdB
!       !       !
!       !       !---ÅjdB
!       !
!       +---ÅidB
!       !       !---iÅdB
!       !       !
!       !       !---idB
!       !       !
!       !       !---iCdB
!       !       !
!       !       !---iDdB
!       !       !
!       !       !---ÅidB
!       !
!       +---ÅRd
!       !       !---RÅdB
!       !       !
!       !       !---RdB
!       !       !
!       !       !---RCdB
!       !       !
!       !       !---RDdB
!       !       !
!       !       !---ÅRdB
!       !
!       +---Åhd
!               !---hÅdB
!               !
!               !---hdB
!               !
!               !---hCdB
!               !
!               !---hDdB
!               !
!               !---ÅhdB
!
+---ÅhdÅ
        +---ÅjdÅ
        !       !---jÅdÅ (feasible)
        !       !
        !       !---jdÅ
        !       !
        !       !---jCdÅ
        !       !
        !       !---jDdÅ
        !       !
        !       !---ÅjdÅ
        !
        +---ÅidÅ
        !       !---iÅdÅ
        !       !
        !       !---idÅ
        !       !
        !       !---iCdÅ
        !       !
        !       !---iDdÅ
        !       !
        !       !---ÅidÅ
        !
        +---ÅRdÅ
        !       !---RÅdÅ
        !       !
        !       !---RdÅ
        !       !
        !       !---RCdÅ
        !       !
        !       !---RDdÅ
        !       !
        !       !---ÅRdÅ
        !
        +---ÅhdÅ
                !---hÅdÅ
                !
                !---hdÅ
                !
                !---hCdÅ
                !
                !---hDdÅ
                !
                !---ÅhdÅ
```

Secondary simulation tree:

```
Åh
+---Åj
!       !---jÅ
!       !
!       !---j
!       !
!       !---jC (feasible)
!       !
!       !---jD
!       !
!       !---Åj
!
+---Åi
!       !---iÅ
```

5. Sorting the Suggestion List

After gathering the suggestions, this sorting procedure is executed to sort out the suggestion list efficiently so that user may get the suggestions in the most useful format. In order to sort the suggestion list most usefully, we must sort it according to the phonetic similarity of each suggested word with the related miss-spelt word. A word with a pronunciation nearest to the miss-spelt word, will be at the top of the list. As this difference becomes more and more profound, the word also goes downward in the list. This kind of highly-efficient sorting algorithm will not be discussed here. Rather, a very common and ordinary algorithm has been used for the time being. It is the *EditDistance* algorithm. *EditDistance* algorithm determines the priority of words by calculating the smallest number of insertions, deletions and substitutions required to change one string to another [1, 4, 5].

6. Replacing the Miss-spelt Word by a Suggested Word

This function replaces any wrongly spelt word according to the user's choice. User chooses a word for replacement from the suggestion dialog-box.

7. Adding a New Word to the Custom Dictionary

This function is invoked to add a new word along with its weight. This word is considered as a wrong word and it is inserted into the custom dictionary in such a way that the sort order remains correct.

8. Ignoring the Miss-Spelt Word

This function is invoked to keep a miss spelt word intact and look for the next miss spelling.

V. CONCLUSION

Here, we have endeavored to kindle a glimpse in the horizon of ultimate spell checking solution for all languages and especially for South Asian languages. Since, the *RecursiveSimulation* algorithm is sill at its nascent stage, it has many bugs and limitations. All the critical cases that could occur have not yet been considered. Moreover, this algorithm is not that fast and efficient for all the languages across the world and may not be able to manipulate all the compound letters properly. But, through robust research and further development, there is a good chance of overcoming all its limitations and someday it may become a great spell checking and suggestion making algorithm.

REFERENCES

[1] http://www.nist.gov/dads/HTML/editdistance .html
[2] http://www.nist.gov/dads/HTML/doubleMeta phone.html
[3] http://www.nist.gov/dads/HTML/metaphone. html
[4] http://www.codeproject.com/cpp/spellchecke r_mg.asp
[5] http://www.codeproject.com/cpp/spellchecke r_pp.asp

[6] comp.databases.informix (Newsgroup)
[7] Microsoft Developer Network (MSDN – April 2002)
[8] http://www.merriampark.com/ld.htm
[9] Baase, S. (1983). *Computer Algorithms.* Mass.: Addison-Wesley
[10] http://www.languageinindia.com/april2003/ verba lcomp.html
[11] Dalianis, H. (2001). *Evaluating a spelling support in a search engine*. IPLABNADA, Royal Institute of Technology, KTH, Stockholm.
[12] Hodge, Victoria J. and Austin, Jim. (2001a). *An Evaluation of Phonetic Spell Checkers*, Technical Report YCS 338. Department of Computer Science, University of York.
[13] Hodge, Victoria J. and Austin, Jim. (2001b). *An Evaluation of Standard Spell Checking Algorithms and a Binary Neural Approach*. Accepted for, IEEE Transactions on Knowledge and Data Engineering.
[14] Kann, V., Domeij, R., Hollman, J. and Tillenius, M. (1998). *Implementation aspects and applications of a spelling correction algorithm*. NADA report TRITA-NA-9813.
[15] Kukich, K. (1992). *Techniques for automatically correcting words in text*, ACM
Computing Surveys. Vol.24, No. 4 (Dec. 1992)
[16] Siraj, K. (1997). *Adhunik Bhasahijnan.* Dhaka: Bangla Academy
[17] Huq, M. (1991). *Bangla Bananer Niyam.* Dhaka: Jatiya Shahitya Prokashoni
[18] Huq, A. (1989). *Adhunik Bhashatattver Sharup-O-Prajukti*. Dhaka: Bangla Academy
[19] Ghosh, S. M. K. (1986). *Bangla Banan.* Calcutta : De'j Publishing
[20] Quarishi, F. A. (1993). *Bnagla Barnomalar Anatomy.* Dhaka: Shamiksha Prokashoni