ISSN 2320 2629

Volume 1, No.2, November – December 2012
International Journal of Information Technology Infrastructure
Available Online at http://warse.org/pdfs/ijacst04112012.pdf

Normalization of Non Standard Words for Kannada Speech Synthesis

Jagadish S Kallimani[1], Srinivasa K G[2], Eswara Reddy B[3]
[1]Research Scholer, Department of Computer Science and Engineering, JNTU Kakinada, AP, India, jsk_msrit@rediffmail.com
[2]Department of Computer Science and Engineering, M S Ramaiah Institute of Technology, Bangalore, India, srinivasa.kg@gmail.com
[3]Department of Computer Science and Engineering, Jawaharlal Nehru Technological University, Anantapur, Andhra Pradesh, India, eswarcsejntu@gmail.com

Abstract: The purpose of summary of an article is to facilitate quick and accurate identification of the topic of published document. The objective is to save a prospective reader's time and effort in finding the useful information in a given article.
This paper considers the task of text normalization in concatinative Text To Speech (TTS) synthesis for Kannada language. The main focus is to have a single document summarization tool based on statistical approach. This deals on how non standard Kannada words - acronyms, abbreviations, proper names derived from other languages or clutters, phone numbers, decimal numbers, fractions, ordinary numbers, sequence of numbers, money, dates, measures, titles, times and symbols - are preprocessed before passing it to the TTS system as an input. The paper also discusses about the methodology used to normalize the non Kannada text present in the input text to get an equivalent Kannada as output. The method uses a fast lexical analyzer, *Jflex* to scan the input to find the non standard words in the given input document.

Keywords: Grapheme to Phoneme (G2P), Linear Predictive Coding (LPC), Non Standard Words (NSW), Text-To-Speech (TTS) Synthesis.

INTRODUCTION

A TTS synthesizer generates speech from a given text. Although TTS is not yet able to replicate the quality of recorded human speech, it has improved greatly in recent years. There exist different synthesis technologies suitable for different applications. A non-general system could have a limited vocabulary support and limitations in the length of spoken utterances.

Multilingual speech processing has become an interesting area to the research community for many years and the field is receiving renewed interest owing to two strong driving forces [1]. Technical advances in speech recognition and synthesis are posing new challenges and opportunities to researchers. For instance, discriminative features are seeing wide application by the speech recognition community, but additional issues arise when using such features in a multilingual setting. Another situation is the apparent convergence of speech recognition and speech synthesis technologies in the form of statistical parametric methodologies. This convergence enables the investigation of new approaches to unified modeling for automatic speech recognition and TTS synthesis as well as cross-lingual speaker adaptation for TTS. The second driving force is the impetus being provided by both government and industry for technologies to break down domestic and international language barriers. Speech signal of an utterance in a language s the only physical event that can be recorded and reproduced. The signal can be further processed in two directions – signal and linguistic processing. During linguistic processing, signals are cut into chunks of varying degrees of abstraction such as acoustic-phonetic segments, allophones, phonemes, morphophonemes, etc, will be ultimately correlated with the letters in the script of a language.

Basically, there is no simple metric that could be applied to any TTS system and which would reveal the overall quality of the system. One reason for this is that it is usually not very meaningful to assess TTS systems in isolation, but it is often more useful to evaluate them in different applications in which the system would be used in practice. Different applications have differing needs from a TTS system.

The easiest way to create synthetic speech is to concatenate audio samples of natural speech, such as individual words or sometimes phrases. This concatenation method guarantees high quality and genuineness, but usually limited by vocabulary and usually available in one voice [2]. This technique is very suitable for some broadcast and information systems. However, it is quite obvious that creating a database of all words and common names from the entire world will be such a hard task. Thus, for unlimited speech synthesis using real TTS technology, we have to operate shorter samples of speech signal, such as phonemes, syllables and diaphones.

MOTIVATION

The text input to the TTS system may not be pure Kannada text. It may contain some Non-Standard Words (NSW) like acronyms, abbreviations, proper names derived from other languages or clutters, phone numbers, decimal numbers, fractions, ordinary numbers, sequence of numbers, money, dates, measures, titles, times and symbols [3]. The natural language processing module of an advanced TTS should be able to handle such NSW also. Standard words are those, whose pronunciation can be obtained from the Grapheme to Phoneme (G2P) rules. A G2P converter maps a word to a sequence of phones. All the NSW must be expanded into the corresponding Kannada grapheme form before sending to the G2P module for phonetic expansion. This module should also take a decision of how a NSW is being pronounced. For example, a phone number should not be read like an ordinary number. Each digit in the phone number must be treated as a single number and must be read in isolation.

PROPOSED SYSTEM

It is an attempt to analyze and normalize the input Kannada text to get the efficient speech output. The major issue involved in normalizing the Kannada text is to handle NSW particularly.

The objectives are to:

- Understand the complexities of text normalization.
- Understand the various available text normalization systems with their characteristics, functionality and tradeoffs.
- Understand the practical design and implementation issues of text normalization systems for several Indian languages.
- Develop an efficient text normalizer for Kannada language, which can be used for obtaining speech outputs from Kannada TTS system.

TEXT NORMALIZATION

Text normalization is the process of normalizing non-standard form of text such as number, year, date, time, acronym and abbreviation into standard form. For example, *Dr* would sound like *doctor,* 7th would sound like *seventh,* and so on. Moreover, certain numbers have to be pronounced as individual digits or as a whole. For example, a phone number such as 91234567809 will be pronounced *nine one two three four five six seven eight zero nine*, but it will be pronounced as *nine thousand one hundred twenty three crores forty five lakhs sixty seven thousand eight hundred and nine* if it is referred as a measurement.

This section gives description of various text normalization techniques for various languages.

Tokenization and classification

In all languages, whitespace is the most commonly used delimiter between words and is extensively employed for tokenization. But sometimes, the token will not be recognized as a single token, but split up into two or more tokens. For example, consider a telephone number, +91 012 5678 1231. This should be identified as a single token of type *Telephone Number*, but if tokenization is exclusively based on whitespace, then we get four tokens. Later, every token have to go through a token identification process that identifies its token type. This approach might not even be feasible for some languages. For example, Chinese and Japanese do not use any form of whitespace between words.

In our approach to text normalization, tokenization and classification are achieved in a single step. We have used *Flex*, an automatic generator tool for high-performance scanners (Mason, 1990), which is primarily used by compiler writers to develop scanners that break up a character stream into a sequence of tokens in the front-end of a compiler. *Flex* takes a set of regular expressions as input and generates a scanner as output that will scan an input stream for the tokens represented by the regular expression. A scanner works as a lexical analyzer, recognizes lexical patterns in the input text, and thereby groups input characters into tokens. Tokens are specified using patterns. An effort is made to identify various non-standard representations of the words in Kannada text. Various formats of each NSW category are defined through regular expressions. English language characters and Arabic numerals are also processed as they appear frequently in Kannada text.

The input text is chunked into sentences based on the sentence delimiter *PurN Viram*. When the generated lexical analyzer is run on each sentence, it analyses the text looking for strings which match one of its patterns. If it finds more than one match, it selects the one that matches the largest chunk of text. If it finds two or more matches of the same length, the first matching rule is chosen. So, by defining regular expressions that match the formats of the various token types, we can automatically extract the token that best fits the given token description. In case of ambiguity between two or more token types for a particular token, the lexical analyzer has been configured to output the possible categories with the token to facilitate token sense disambiguation at a later stage. Using this approach, we can complete tokenization and classification.

Token sense disambiguation

Once the tokens are extracted from the input text, the type of each tokens need to be identified. Identification of token type involves high degree of ambiguity. For example, *1977* could be of the type *Year*, or of the type *Cardinal Number* and *1.25* could be of the type *Float*, or of the type *Time*. Disambiguation is generally handled by manual, hand-crafted and context-dependent rules. However, such rules are very difficult to write, maintain, and adapt to new domains. Token sense disambiguation can be mapped to a general homograph disambiguation problem (Yarowsky, 1996). We have used decision tree based data-driven techniques to address this issue.

Decision trees and decision lists

Decision trees are models based on self learning procedures that sort the instances in the learning data. The decision tree algorithm selects both the best attribute and the question to be asked about that attribute. The selection is based on what attribute and question about it divide the learning data in order to get the best predictive value for the classification. When a token is issued to the tree for disambiguation, a decision is made by traversing the tree starting from the root, taking various paths satisfying the conditions at intermediate nodes, till the leaf. The path taken depends on various contextual features defined for the token. The leaf node contains the predictive value for the decision. Decision lists are a special class of decision trees. Decision lists may be the simplest model for hierarchal decision making. Despite their simplicity, they can be used for representing a wide range of classifiers. A decision list can be viewed as a hierarchy of rules. When a classification is needed, the first rule in the hierarchy is addressed. If this rule suggests a classification, then its decision is taken to be the classification of the decision list. Otherwise, the second rule in the hierarchy is addressed. If that rule fails to classify as well, the third rule is addressed, and so on. Often, programmers prefer presenting decision lists as sequences of if-then-else statements, intended for classifying an instance x.

Tokenization

The tokenization undergoes three levels such as:

- Tokenizer
- Splitter and
- Classifier.

The whitespace is used to tokenize a string of characters into a separate token. Punctuation and delimiter were identified and used by the splitter to classify the token. Context sensitive rules written as whitespace is not a valid delimiter for tokenizing phone numbers, year, time and floating point numbers. Finally, the classifier classifies the token by looking at the contextual rule. Different forms of delimiters are removed in this step. For each type of token, regular expression are written in *.jflex* format. Then using *JFlex* toolkit a Lexer file is generated. In this way the whole tokenization process is performed. All regular expressions are designed according to predefined semiotic classes and the rules of the context that are obtained in the previous semiotic class identification phase. This study is unique as decision tree and decision list are used for disambiguation. The generated Lexer file is used in the token expansion phase. The generated Lexer is a java class file which is then invoked by a driver class to get the list of tokens. According to the tag in the list, each type of token expander class is then invoked for expanding the token.

Verbalization & disambiguation

The token expander expands the token by verbalizing and disambiguating the ambiguous token. Verbalization or standard word generation is the process of converting non natural language text into standard words or natural language text. A template based approach such as the lexicon is used for cardinal, ordinal, acronym, and abbreviations. For expanding the cardinal number, calculate the position of the digit rather than dividing by 10. To expand the cardinal number token:

- Traverse from right to left.
- Map first two digits with lexicon to get the expanded form (For instance, 100 as *hundred*).
- After the expanded form of the third digit, insert the token *hundred*.
- Get expanded form of each pair of digit after third digit from the lexicon.
- Insert the token *thousand* after the expanded form fourth and fifth digit and *lakh* after expanded form of sixth and seventh digit.

These processes continue for each seven digits. Each seven digit is divided as a separate block. After each of the second block insert the token *crore*. So the expanded form of token 39019 is *thirty nine thousand and nineteen*.

The detailed functional requirement system of the proposed system is given in Table 1.

Table 1: Functional Requirements

Use Case Name	Enter Text in Kannada
Trigger	The User runs the Kannada TTS Normalizer
Basic Path	1. The User enters the Kannada text in the text box provided. 2. The User clicks the input to file button. 3. The User then runs the JFlex tool to tokenize the input text. 4. The System gets locked to prevent further in puts by the user. 5. The System generates the equivalent normalized text. 6. The System generates speech file of the normalized text. 7. The speech file is played out.
Alternative Paths	Not Applicable.
Post condition	Kannada written in English output for the normalized text.
Exception Paths	Error message is displayed in case of exceptions.
Other	GUI which is user friendly.

Introduction to *JFlex*

A frequently encountered problem in real life application is that of checking the validity of field entries in a form. For example, a form field may require a user to enter a strong password which usually must contain at least one lower case letter, an upper case letter and a digit. If the user fails to enter password with such specifications, the program should respond by alerting the user with appropriate message. The job of checking the validity of fields in our application thus properly falls to the lexical analyzer [4]. In this case, the Graphical User Interface (GUI) form collects the inputs, constructs an input string from the input fields and supplied values, and channels the input string to the scanner. The scanner matches each segment of the input string against a regular expression and reports its observation. Thus the report is generated and given to the GUI for the user. The user is allowed to correct any erroneous field as long as it appears. *Jflex* is a lexical analyzer generator for Java written in Java. The main advantages of *Jflex* are:

- Full Unicode support
- Fast generated scanners
- Convenient specification syntax
- Platform independent
- *JLex* compatible

The syntax of the lexical rules section is described by the following BNF grammar:

```
Lexical Rules: = Rule+
Rule: = [State List] ['^'] RegExp [Look Ahead] Action
| [State List] '<<EOF>>' Action
| State Group
State Group: = State List' {' Rule+ '}'
State List: = '<' Identifier (',' Identifier)* '>'
Look Ahead: = '$' | '/' RegExp
Action: =' {' Java Code '}' | '|'
RegExp: = RegExp '|' RegExp
```

Figure 1: The Lexical Rules

Methodology

Let us consider different samples of Kannada articles and we can easily find out lots of NSW present within them. When this document is passed to TTS as input TTS skips this words and pronounces only the characters which are in Kannada text. This problem is to be addressed in order to get pleasant and complete speech output.

NSW in Kannada language

From the above mentioned articles, it is clear that around 7 to 8% of data in any article contains NSW which cannot be handled by a normal TTS [5][6]. The different NSW in Kannada articles are:

- Cardinal numbers and Literal Strings
- Ordinal numbers
- Roman Numerals
- Fractions
- Ratios
- Decimal Numbers
- Telephone Numbers
- Date, Year
- E-mail
- Percentage, Alphanumeric strings

The purpose of the design is to plan the solution for handling NSW in any article. This phase is the first step in moving from problem to the solution domain. The design of the system is the most critical factor affecting the quality of the software and has a major impact on the later phases, particularly testing and maintenance.

System design aims to identify the modules that should be in the system. We need to know the specification of these modules and interaction with each other to produce the desired results. At the end of the system design all the major modules in the system and their specification are decided [7] [8]. The following data flow diagrams illustrate the working of overall system. Figure 2 shows the context diagram of the normalization of Kannada TTS system. The system accepts Kannada text as input which requires normalization. It then produces the normalized Kannada text which is passed to the TTS to produce equivalent speech output by reading the corresponding speech file from the speech database.

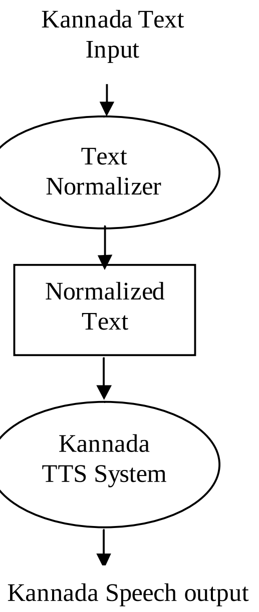


Figure 2: Normalization of Kannada in TTS System

The normalizer phase is divided into two modules, *normalize-input-text* and *process-normalized-text*. The *normalize-input-text* module takes the initial input and finds the characters which needs normalization and normalizes them. Finally *process-normalized-text* module takes the normalized text and finds out the corresponding .wav file to produce speech output.

Tokenization, expansion and verbalization of tokens [9] [10] are the major phases, shown in figure 3. In tokenization we have three steps namely, tokenizing, splitting and classifying token into different tags like <*NUM*>, <*FLOAT*>, <*EMAIL*> etc. If the number string is not an ordinary number, a parameter is set according to the type of the number string. If the number string is a decimal number (Ex: 23.8756) the number before the dot (.) is treated as one number and the digits after the dot are spoken in isolation. If the number string is a *date*, the delimiters can be '/' or '-' (Ex: 25-10-1999 or 25/10/1999) for all these things we have regular expression to match these types. In splitter, we are using punctuation mark to split between different types of tokens. We also use white space for splitting between tokens. After token is splitted in to different classes like number, decimal number etc we use rule based system to classify ambiguous tokens.

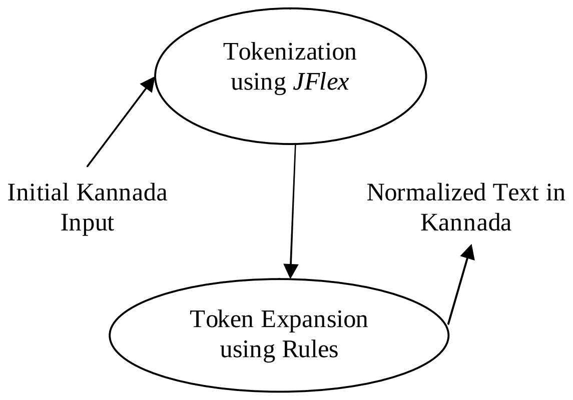

Tokenization using *JFlex*
Initial Kannada Input
Normalized Text in Kannada
Token Expansion using Rules

Figure 3: Tokenization, Expansion and Verbalization

After the normalization of the input text, the *process-character* module takes the normalized Kannada text and breaks it down into words. The words are broken down into characters. The individual characters are the input for the *produce-phoneme* module. The characters are rearranged according to the rules in Kannada language and the output phoneme files are produced. The phoneme files are taken as an input by *identify-audio-files* module. This module consults the phoneme file path and speech database to produce the audio file. The audio file is then fed to the *strip-audio-files* module. This module strips-off the silence in the speech file. After silence removal, the stripped audio file is input to the *merge-audio-file* module. The output of this module is the final concatenated audio file.

THE SYSTEM

The methodology for normalizing Kannada text is rule based system rather than the decision tree. The block diagram for normalizing Kannada language is shown in figure 4. This model is classified into two main groups namely:

- Tokenization using *Jflex*
- Token expansion and verbalization

Tokenization

This phase is subdivided into:

- Tokenizer
- Splitter
- Classifier

Main job of tokenizer is to identify the token present in the given text. In order to indentify the tokens we have to write regular expression for each token in *JFlex* tool. White space character is the mostly used delimiter to identify the tokens in this method. We are also using white space for identifying the different set of tokens. For each type of token, regular expression are written in *.jflex* format. Then using *JFlex* toolkit, a lexer file is generated. If a regular expression is matched then we assign a tag in list[i] and token in list [i+1]. In this way the whole tokenization process is performed. All regular expressions are designed according to our predefined semiotic classes and the rules of the context that are obtained in the previous semiotic class identification phase. This study is unique, where decision tree and decision list are used for disambiguation.

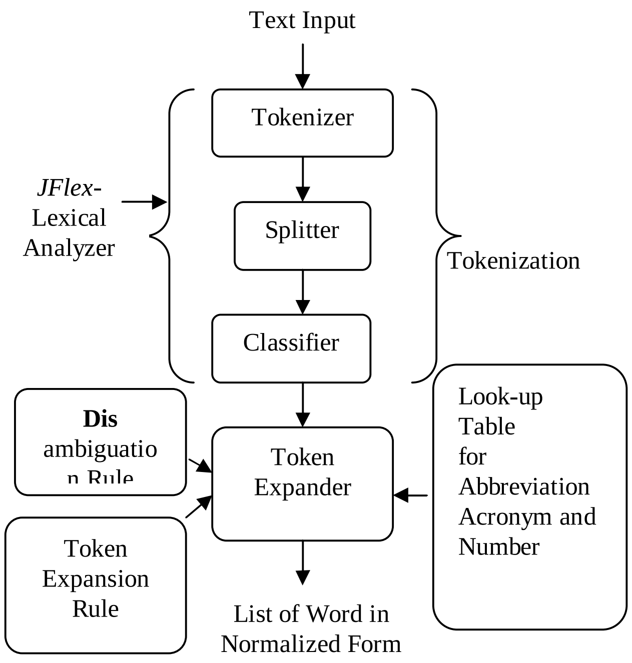


Figure 4: Block Diagram of Text Normalization

Punctuation marks are used to split between the token and context sensitive rules are written to classify these tokens into different tag names like <*NUM*>, <*FLOAT*> etc.
Context sensitive rules are written to classify tokens in to different set of tag names like <*NUM*> tag for all numbers, <*FLOAT*> tag is for all floating point tokens and so on. Classifier does not clear all ambiguity between all the tokens.

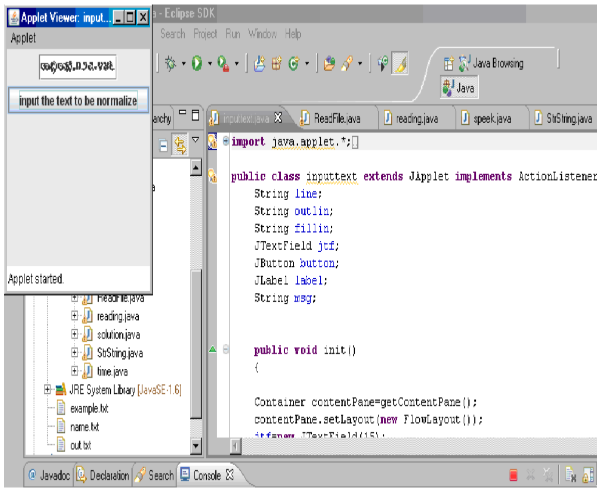
Token expansion and verbalization

The generated lexer file is used in the token expansion phase. The generated lexer is a java class file which is then invoked by a driver class to get the list of the token. According to the tag in the list, each type of token expander class is then invoked for expanding the token. Token expander expands the token using expansion rules. Consider a cardinal number. The rule used is to divide the number by ten and get the remainder. Verbalization or standard word generation is the process of converting non natural words to natural language. Lexicon language is used for expansion of cardinal's ordinals numbers. For expanding ordinal number, we use the rule as divide by 10 and take the position of the numbers. So we scan from the right side and we divide the number into last three digits and later we divide every 2 digits and so on we add string like *nuru* after 3rd digit and after 4th and 5th we use *savira* after 6th and 7th digit we put *laksha* and so on.

Consider the number 12345. when we divide it by ten we get remainder as 5 and verbalization rule checks its position here it is one so don't add any extra string after number 5. Next when we divide the quotient we get 4 but in verbalization it is in 2nd place so add string *hattu*, and for 3 it is *nuru* and so on. Finally we get the string as *hanneradu* ***savirada*** *muru* ***nura*** *nalavattu aidu*.

RESULTS

The process of text normalization for Kannada language has been considered in the development of efficient concatenative TTS synthesis. The obtained results are discussed in this section which shows the GUI developed tokenization through *Jflex* and conversion of NSW to their Kannada form.



```
import java.applet.*;

public class inputtext extends JApplet implements ActionListener
    String line;
    String outlin;
    String fillin;
    JTextField jtf;
    JButton button;
    JLabel label;
    String msg;


    public void init()
    {

    Container contentPane=getContentPane();
    contentPane.setLayout(new FlowLayout());
```

Figure 5: Input to the System

Jflex is a tool which accepts *.jflex* file and convert it to equivalent java file. These java files are mainly used to make tokenization in lexical analysis. For the input,

???? 12345 19-03-2011, abc@def.co.in, 123.456

through *test.txt* input file, the matched tokens generated are shown below in figure 6.

```
List size: 2
Start of tok
Tag: 4 token: ?
Tag: 4 token: ?
Tag: 4 token: ?
Tag: 4 token: ?
Tag: 4 token: 12345
Tag: 4 token: 19-03-2011
Tag: 4 token: ,
Tag: 4 token: abc@def.co.in
Tag: 4 token: ,
Tag: 4 token: 123.456
End of tok
```

Figure 6: Results of the Tokenization

Finally, the output with the normalized text is obtained for the given input. This is shown in below figure 7.

CONCLUSION

In this paper, the method for text normalization for Kannada language using lexical analyzer *Jflex* has been discussed. The paper presents the complexities of Kannada language and the method to normalize the NSW of Kannada. The proposed rule based system is not able to completely classify the tokens (such as pin code number, the phone number, etc) depending on the context.

The presented work is suitable only for some specialized cases of the Kannada language but in future for large amount of complex cases can also be considered. The proposed system does not handle the context specific text which can be addressed later.

```
?
punctuation mark
12345
integer number
hanneradu savirada muru nura nalavattu idhu
19-03-2011
the given nor 19-03-2011
hattombhattu
muru
yeradu savirada hannondu
,
punctuation mark
abc@def.co.in
email id
the given mail id is abc@def.co.in
a b c at d e f dot co dot in
,
punctuation mark
123.456
float number
the given float is 123.456
ondu nura ippattu muru
point
nalku idhu aaru
```

Figure 7: Results after the Normalization

REFERENCES

[1] Hervé Bourlard, John Dines, Mathew Magimai-Doss, Philip N Garner, David Imseng, Petr Motlicek, Hui Liang, Lakshmi Saheer, Fabio Valente, Current trends in multilingual speech processing, *Sa¯dhana¯* Vol. 36, Part 5, October 2011, pp. 885–915._c Indian Academy of Sciences.

[2] Anand Arokia Raj, Tanuja Sarkar, Satish Chandra Pammi, Santhosh Yuvraj, Mohit Bansal, Kishore Prahallad, Alan W Black *Text processing for text-to-speech systems in Indian languages*, 2007.

[3] Cohen M, Giangola J, and Balogh J, *Voice User Interface Design*. Addison Wesley, 2004.

[4] Elliot Berk, JFlex - The Fast Scanner Generator for Java, 2004, version 1.4.1, http://jflex.de.

[5] Flanagan J, *Speech Analysis, Synthesis and Perception*. Springer-Verlag,

[6] *History and Development of Speech Synthesis*, Helsinki University of Technology, Retrieved on November 4, 2006.

[7] Julia Zhang. *Language Generation and Speech Synthesis in Dialogues for Language learning*, master's thesis, http://groups.csail.mit.edu/sls/publications/2004/zhang_thesis.pdf. Section 5.6 on page 54.

[8] Paul Taylor, *Text to Speech Synthesis*. University of Cambridge, 2007. Pp.71-111, (draft), Retrieved (June, 19, 2008). http://mi.eng.cam.ac.uk/~pat40/ttsbook_draft_2.pdf.

[9] Peri Bhaskararao, Salient phonetic features of Indian languages in speech technology, *Sa¯dhana¯* Vol. 36, Part 5, October 2011, pp. 587–599._c Indian Academy of Sciences.

[10] Sproat R., Black A.W., Chen S., Kumar S., Ostendorf M, and Richards C., *Normalization of non-standard words*, Computer Speech and Language, pp. 287–333, 2001.