NATIVE LANGUAGE PROCESSING

A language processor for understanding languages compliant with the grammar of Hindi language and extension to a QA system

Anand Bora, Aman Kumar
B.Tech. (2006)
Computer Science & Engineering
SASTRA Deemed University
Thanjavur

September 10, 2006

ABSTRACT

This paper aims at developing a very basic NLP (Natural Language Processing) system for the Hindi language. Also this paper proposes a format which can make the system compatible with the languages supporting the grammar of Hindi Language. This includes languages like Punjabi, Guajarati and the different type of dialects in which Hindi is spoken in different parts of India. The flexibility is possible due to the flexible word structure proposed. Due to the closeness of the system with the Indian Languages, the project has been named as **NATIVE LANGUAGE PROCESSING**. The project is based on XML file access and parsing. In addition to all the conventional parts an NLP system uses, the system has an optional answering part for generating a proper response to the given output. The system has been modelled in such a manner that it can process a given sentence and generate the outputs at different levels of the language processor. Moreover the structure of the system makes it possible to be compatible with most of the modern languages.

1. INTRODUCTION

Proposals for mechanical translators of languages pre-date the invention of the digital computer. The first recognisable NLP application was a dictionary look-up system developed at Birkbeck College, London in 1948. American interest is generally dated to a memorandum written by Warren Weaver in 1949. His idea was simple: given that humans of all nations are much the same (inspite of speaking a variety of languages), a document in one language could be viewed as having been written in code. Once this code was broken, it would be possible to output the document in another language. From this point of view, German was English in code. As a research idea, this caught on quickly. The key developments during the 1980's were in the fields of Augmented Transition network, Case Grammar and Semantic representations. Thus, a way was found to get around the semantic information bottleneck. Though a lot of research has been carried out after that, a generic system which should be capable of processing most of the Indian languages has not yet been developed. We try to find a way of for this by making a generic system

which will be capable of processing most of the Indian languages.

Anusaarka Project

This project is the project meant for Morphological Analysis of Hindi Language. Joint research operations are performed in the field of NLP related to Indian Languages by different professors and scholars under this project. We have taken massive references from this project and project's site. This source can also be used in the morpheme analysis phase.

2. LANGUAGE PROCESSOR STRUCTURE

The project (paper) is based on XML file access and parsing. It consists of all the conventional parts of a language processor *viz.* Lexical Analyzer (for tokenization, word validation and morpheme analysis), Syntactic Analyzer (Syntax Analysis done by Recursive Transition Network Parser), Semantic Analyzer (for POS tagging and disambiguation), Analyzer (for understanding the meaning) and an *optional* answering part for generating a proper response to the given input. The system has been modelled in such a manner that it can process a given sentence and generate the outputs at different levels of the language processor.

The system is a platform independent system. The main programming language used for the development of this system is C++. Different modules generate different outputs which are fed into the final analyzer stage. The important part of the system is the XML file structure. Various Xml files are needed to keep track of the words in the language and also the grammar codes. Since the analyzer system learns new words on runtime, the manipulation of these files is done adequately. In addition, different algorithms have been used to learn new words and reject inadequate words. Learning is done in the conventional manner of asking questions from the user. The system becomes more and more intelligent as it is trained through time. The fundamentals of learning in the project are deriving proper keywords for unknown words from the answers given by the user.

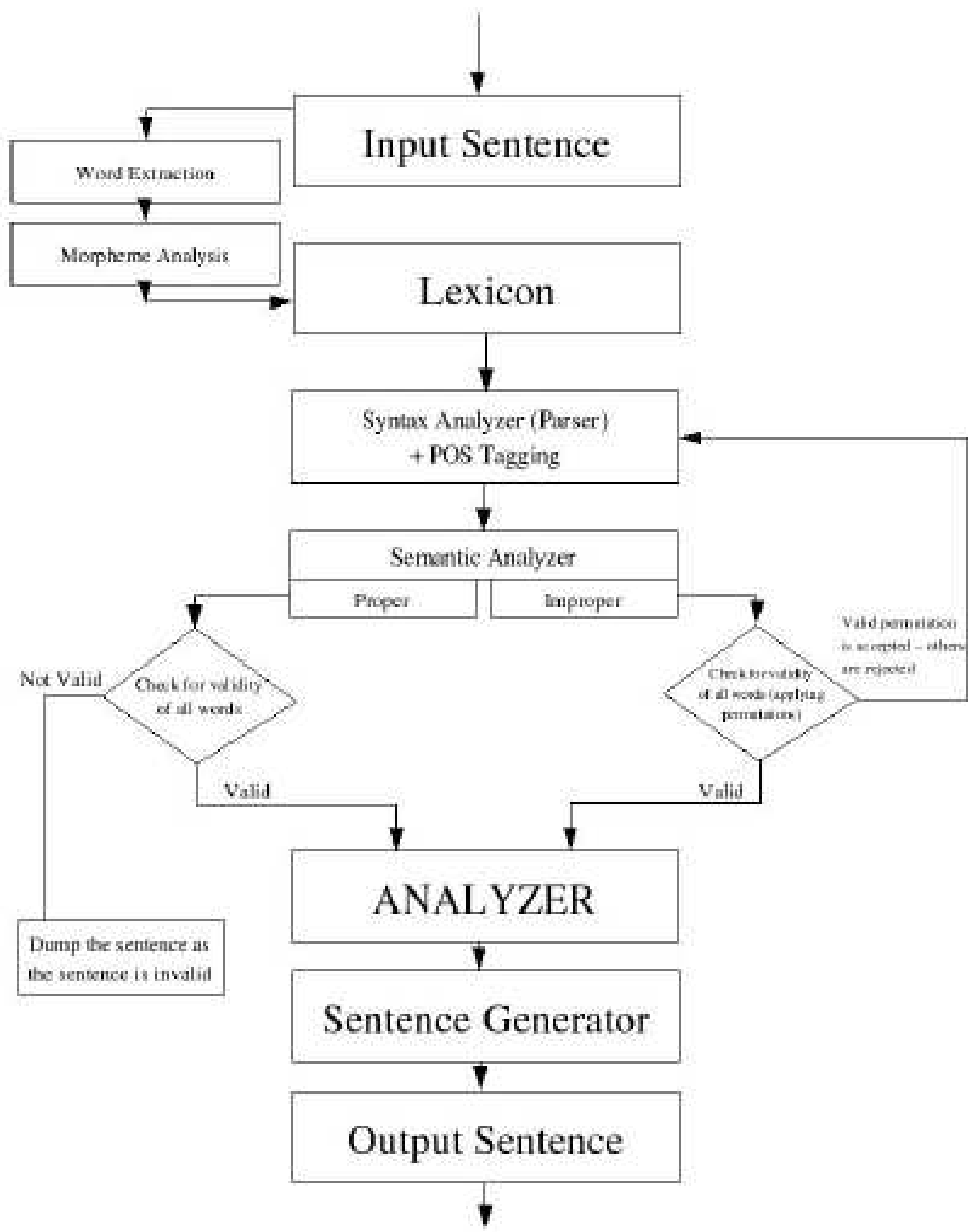

Figure 1.

The modules shown in Fig. 1 are described as follows
.

2.1 PROJECT IMPLEMENTATION DETAILS: Basic implementation

Two methods of morpheme analysis have been implemented, viz., Basic Implementation and the Anusaarka Project[1]. Our basic implementation scheme mirrors that of NLP - Akshar Bharati Book[2] with our modifications. The root words are extracted only from those words that are included inside the given word. Most of the words in our analyzer are mapped in this manner. We have used XML tags for the morphemes as well. This means that for each word there can be a morpheme entry in the XML wordlist file. If the given word is derived from the root word after adding the morpheme, then the word is considered valid and is already available in the wordlist. For any other case, the word is treated as an out of dictionary word. The general structure is shown in the Lexicon discussion section of the article.

The algorithm used for morphological analysis is that used by NLP - A Panian Perspective[2]. It is mentioned here

ALGORITHM

```
1. L:=empty set
2. If w is in DI with entry b
        then add b to set L.
3. For i:=0 to length w do
        let S=suffix of length i in w
        if reverse(s) in RST
                then for each entry b
        associated with reverse(s) in RST
        do
                proposed-root = (w –
        suffix s) + string added from root
                if( proposed-root is in
        dictionary of roots)
                then
                        construct lexical
                entry 1 by combining
                features given for the
                proposed root in DR
                        add 1 to L
                end for each entry
        end for I
4. If L is empty then return (“unknown
word w”) else returns (L)
```

END ALGORITHM

2.2 UNKNOWN INVALID WORD REJECTION

This is a small part in which the system is trained beforehand. The engine is trained for a particular number of words in the language. This wordlist is passed through the engine. Then the transition matrix for the given set of words is formed. This transition matrix gives a pattern for the new words being fed into the system. Apart from this, a recursive filtering module is designed which further enhances the word filtering. This is done by adding a number of test cases.

This part can be modified by using new soft computing techniques. An example for this kind of word rejection is the following: Hindi language cannot have ‘xx’ anytime or anywhere. The transition value is always zero as we have trained the system for ‘n’ number of words. So when this type of string is encountered it is straightaway rejected. It is also passed through this filtering module.

2.3 WORD – LIST ARCHITECTURE

The rich wordlist that has been used in the project is implemented in XML.

Sample tree of the wordlist formed

```
- <wordlist>
    - <m>
        - <mota>
        - <type>
            <adjective />
            <noun />
          </type>
          <gender>M</gender>
          <number>S</number>
          <person>T</person>
        - <morphemes>
        - <i>
            <gender>F</gender>
          </i>
        - <ey>
            <number>P</number>
          </ey>
          </morphemes>
```

First of all indexing has been maintained according to alphabetical order i.e., all the words starting with alphabet ‘a’ has been tagged inside the alphabet ‘a’. Inside the alphabet tagging various other tagging has been maintained such as gender, type, morpheme, noun, adjective, person, and so on. The XML architecture has been implemented in order to make parsing simpler. For the XML architecture of word list please refer to the code section.

2.4 SYNTAX ANALYSIS (PARSER)

Syntax analysis is a process in compilers that recognizes the structure of programming languages. It is also known as parsing. The parser used in the project is a simple *Recursive Transition Network*. This parser though not efficient is enough for parsing basic sentences. We have considered simple sentences and have eliminated the need for complex grammar evaluation. For designing the grammar of basic Hindi Language, we considered different basic sentences which have been enlisted later.

James Allen has described RTN parser beautifully in his book for Natural Language Understanding[3]. This can be described as follows: Simple transition networks are often called finite state machines (FSMs). Finite state machines are equivalent in expressive power to regular grammars, and thus are not powerful enough to describe all languages that can be described by a CFG. To get the descriptive power of CFGs, you need a notion of recursion in the network grammar. A recursive transition network (RTN) is like a simple transition network, except that it allows arc labels to refer to other networks as well as word categories. Thus, given the NP network in figure 1, a network for simple English sentences can be expressed as shown in figure 2. Uppercase labels refer to networks. The arc from S to S1 can be followed only if the NP network can be successfully traversed to a pop arc. Although not shown in this example, RTN’s allow true recursion—that is, a network might have an arc labelled with its own name.

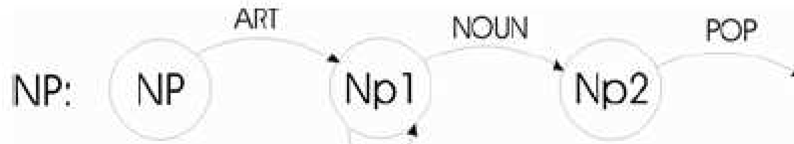

Figure 1

Figure 2

A separate parse code file is maintained which keeps track of the different grammars possible. For every call of the expandable Non Terminal, the RTN calls the related tag recursively. This means that at each call the next word is matched for its properties. If the category of the next word matches the given parse tree formed then the parser proceeds else it pops out returning error. We have used a very basic list of grammar code. This grammar code can be changed dynamically if grammar learning module is also incorporated in the system. The sample parse code structure which we have used is in the following manner:

SAMPLE PARSE CODE LIST

```
- <codes>
      - <S>
            - <code>
                  <NP />
                  <VP />
            </code>
            - <code>
                  <N />
                  <V />
            </code>
            - <code>
                  <A />
                  <ADJ />
                  <N />
            </code>
      </S>
            - <NP>
</codes>
```

2.5 SEMANTIC ANALYSIS – OUR PROPOSED APPROACH AND PARTIAL IMPLEMENTATION

For any sentence provided, we locate the root verb in the sentence. Then this verb is mapped for the tense form in which it currently exists. This way we can identify the occurrence time of the activity (karaka). Also what has been done can be known by the root word of the verb. In the Anusaarka project[1], this phase is referred as the Vibhakti phase. We have implemented our own algorithm for the same. This root word comes from the XML wordlist provided. Thereafter, the worker (karta of the activity is identified. This can be a pronoun or a noun. We map the person (i.e. first, second or third) for the karta and identify the main worker. Then the karma is identified which can be another noun or pronoun. This is the other person involved in the conversation. This can also be an object. This is identified from the meaning tag of the Xml file. After all the three parts have

been identified, we move on to the karma phase. Though we have already located the root verb, the mapping is yet to be done. Now the system identifies that the karta and karma are related to each other through the karma part.

The system can be explained by the following example:

Raama ne raavana ko tiira se maaraa.
Ram ergative Ravan accus. Arrow instr. Killed.
(Ram killed Ravan with an arrow).

The other important point is the tracking of the postpositions. They are very important at pointing the exact world position and activity scenario. Each postposition (parsarg) has been tracked and the karta and karma are identified adequately. The different postpositions taken into consideration are:
'mein', 'ka', 'ki', 'ke', 'ko', 'se', 'ne', 'par' and 'tak'

For identifying the type of the sentences different signals to the system has been provided. These signals can be enumerated as:

Statement, Negation, Command, Question & Exclamation.

The signals are triggered only when the system identifies proper words related to the given sentence. For instance, a negation statement will always have any of the following three words in the sentence – 'na', 'nahin' or 'mat'. A question will always have words like 'kaun', 'kya' etc. Apart from this a question asked can also be checked by the existence of special symbols in the sentence. An example for this is the '?' in any question. Logical statements are built on the basis of understanding of the parameters talked about earlier. Two important phases of the proposed semantic analyzer are **PROPER** and **IMPROPER**.

The **PROPER** phase parses the sentence in the manner described earlier. The primary condition her is that the sentence must be valid in the parser phase. In other words, the sentence must follow grammar which complies with the available grammar code list.

The **IMPROPER** phase is for understanding those sentences which are structurally not correct but they are semantically correct. We have considered a very basic implementation of this phase too. In this phase, the grammatical construction of sentence is taken. Then different valid permutations are applied on the grammar. Thereafter, the new structured grammar is compared with already available grammar rules. If the rules are available then they are compared and if valid, the sentence is declared valid. Otherwise, the sentence is checked for all possible cases. The sentence is dumped if none of the case matches. The improper phase can also be expanded to incorporate lots of other features.

2.6 PROPOSED PART FOR IMPLEMENTATION ANALYZER PROCESS

This process was earlier intended for both word learning and grammar learning. Word learning has been done in the conventional method of asking questions. The system asks the user about any unknown word. The system maps some keywords from the given response and searches its wordlist for the word. If it finds relevant data available it

stores the word in the Xml file with the meaning tag containing the keywords providing the meaning of the word. This word is stored at the proper tag in the Xml file. This means that the searching is much faster.

This idea can further be classified by the following conversational scenario:

User: main toffee kha raha hoon.
System: mujhe 'toffee' ke bare mein nahin janta.
User: Toffee ek khane ki cheez hai jo ki bahut meethi hoti hai.
[Translation in English]
{
User: I am eating toffee.
System: I don't know about 'toffee'.
User: Toffee is an edible thing which is sweet.
}

In the above case 'khane', 'cheez' and 'meethi' are available in the wordlist. The words have their own respective meanings which provide the required information for the current word being processed. An important point for the analysis is deciding about the noun on runtime. Certain words form different type of relations with real world entities. A typical example can be the 'instance' relation. In the above example toffee will be an instance of object having the different properties. Thus the system will always keep in mind the different aspects about the object. For storing the meaning of the particular word, the word entry is done in the Xml wordlist in the format referred earlier in the lexicon section.

If we talk about making the system generic and extending it to different compatible languages, the constraint which comes initially is the different word forms to be dealt and the different modes of speech. Our system has been basically built for the Hindi Language but this can be modified for different languages by replacing the proper words from different languages in this Expert system.

Grammar learning facilitates learning grammar on the runtime. This is a very difficult part and needs enormous research and development. We also propose a basic grammar learning algorithm. As mentioned earlier we have maintained a parse code xml list for different grammars available in the language. To start with, we first store the structure of the grammar of current sentence in a temporary buffer. Obviously, if we have to learn a new grammar, there will be new words in the sentence. Those words are marked as '*' in the grammar structure stored in the buffer. Words are identified by the conventional question asking method and their category is inserted into the buffer replacing the '*'. This new sentence is considered a valid sentence and the grammar is cross checked in the parser. Grammar Learning can also be done by asking questions. These questions asked from the user will only validate the grammar being provided in the new structure.

Extension of the system to a QA system:
The Analyzer phase returns some words which are send to the Sentence Generator phase. These words are restructured to form a valid sentence in order to give a reply. Though not implemented, the mood and tone of a person in his talk can be tracked and adequate response can be generated.

2.7 SENTENCE GENERATOR

A *Sentence Generator* (SG) constructs sentences, paragraphs, and even papers that fit a prescribed format. The format is specified by a set of rules called a ***grammar***.

We have proposed a very basic sentence generator which generates simple sentences from the grammar rules available. The sentence generator takes some words from the analyzer phase. These words are identified for their categories. Then a basic structure is found out from the grammar rules available which match the current rules. Henceforth, the sentence for the output to the sentence of the input sentence is generated.

This can be further clarified by the following example: Consider the words returned by analyzer phase are – us(that), jagah (place), gayaa (gone).

The system will try to find a valid set of rules and form a valid sentence. Also the system searches for the simplest grammar possible. In the above case, the simplest grammar possible is: Noun+Pronoun+Noun+Verb or Pronoun+Pronoun+Noun+Verb. Here priority is decided on random basis. But in this case the latter case must be followed.

The first word is obviously the system, as it has been modelled. The other three words are then concatenated. The system is very limited and needs to undergo disambiguation. So the output of the sentence generator is "main us jagah gayaa" or "mera us jagah gaya" and so on.

SOURCE CODE EXCERPTS

```
/*Set of standard structures*/
struct types;
struct retxml;
struct morphemes;
/*The Standard XML structure which is
retrieved by other modules*/
struct types
{
        char* type;
// The word type (Eg. Noun/Adj./Verb
etc.)
        char* meaning;
// The meaning of the word of the given
type
        types* next;
// The next type of same word with
various
        types()
        {
                next = NULL;
                type = NULL;
                meaning = NULL;
        }
};

struct morphemes
{
        char* morpheme;
// The various morphemes of the same
word
        retxml* info;
// The various other infos of the given
morpheme
        morphemes* next;
        morphemes()
        {
                morpheme = NULL;
                next = NULL;
                info = NULL;
        }
};
```

3 FUTURE PROSPECTS

NLP has huge prospects in coming time and our system can provide a huge impetus in forming a full fledged NLP system for Hindi Language. Hindi is spoken in five different types in different parts of India. Apart from this there are innumerous dialects which are similar to the language. Our proposed and basic system has been modelled over Xml architecture which means that changing a single word file can change the whole understanding of the system. We have built our own modules and also used some of the resources available on the NET, especially, Anusaarka[1]. Though the system is not a perfect one it gives us a platform for developing fully enhanced NLP systems in the coming time. The applications are enormous and the system expandability is infinite.

REFERENCES

1. Anusaarka Project – http://ltrc.iiit.ac.in/showfile.php?filename=projects/Anusaaraka/index.php
2. Natural language processing – A Paninian perspective
3. Natural language understanding – James Allen
4. www.iiit.net/ltrc/Publications
5. Information Science Institute – www.isi.edu/natural-language
6. Word sense disambiguation – www.senseval.org
7. Open NLP – www.opennlp.sourceforge.net
8. www.citeseer.com
9. www.sciencedirect.com
10. www.au-kbc.org
11. Bauer, Caurie (2004) A glossary of morphology
12. Adaptivity in Question Answering Using Dialogue Interfaces – Silvia Quarteroni and Suresh Manandhar
13. Learning word meaning by instructions – Kevin Knight
14. A plan-based agent architecture for interpreting natural language dialogue - LILIANA ARDISSONO, GUIDO BOELLA AND LEONARDO LESMO
15. Learning word syntactic sub categorizations interactively – Fernando Gomez
16. Knowledge Extraction from Hindi Text - Shachi Dave, Pushpak Bhattacharyya
17. Chinese Lexical Analysis Using Hierarchical Hidden Markov Model – Hua – Ping Zhang, Qun Liu, Xue – Qi Cheng, Hao Zhang, Hong – Kui Yu.