Indian Language to Indian Language Machine Translation

System

(ILMT) System

Software Requirement Specifications

version 1.03

October - 2008

IIIT Hyderabad
Table of Contents

1 Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, acronyms, and abbreviations
1.4 References
1.5 Overview

2 Overall description
2.1 Product Perspective
2.2 Product functions
2.3 User Characteristics
2.4 Constraints: System Structure/Architecture
2.5 Assumptions and dependencies
2.6 Software Engineering Approaches (Reference: s/w engg process for ILMT)
2.7 ILMT System: Accuracy / User satisfaction
    2.7.1 ILMT System: Accuracy / User
2.8 Methodology

3 Specific Requirements
3.1 System Structure
    3.1.1 Common Representation SSF
3.2 ILMT System Architecture
3.3 Information Flow
3.4 Process Description
3.5 Specification of Individual Modules
    3.5.1 Preprocessor
    3.5.2 Tokenizer
    3.5.3 Sandhi Splitter
    3.5.4 Morph Analyzer
    3.5.5 POS Tagger
    3.5.6 Chunker
        3.5.6.1 Chunking
        3.5.6.2 Pruning
3.5.6.2.1 Morph Pruning
3.5.6.2.2 Guess Morph
3.5.6.2.3 Pick one Morph
3.5.6.3 Head computation
3.5.6.4 Inherit head features
3.5.7 Local Word Grouper/Splitter
  3.5.7.1 Local Word Grouper / Vibhakti Computation
3.5.8 Named Entity Recognizer (NER)
3.5.9 Simple Parser
3.5.10 Lexical Sense Disambiguation.
3.5.11 SL to TL Transfer
  3.5.11.1 Transfer Engine Module
  3.5.11.2 Lexical transfer Engine
  3.5.11.3 Transliteration
3.5.12 Put target language features Agreement Feature
3.5.13 Inter-chunk Agreement
3.5.14 Intra-chunk Agreement
3.5.15 TAM Vibhakti Splitter
3.5.16 Agreement Distribution in split vibhakti
3.5.17 Assign Default Features
3.5.18 Word Generation

3.6 Evaluation

4 System Integration and Testing
  4.1 Dashboard
  4.2 Graphical User Interface (GUI)

References

Appendix
Appendix A: XML File Structure for Corpora Creation (.cml files)
Appendix B: Shakti Standard Format: BNF with Brief Description

Revision History
1 Introduction

1.1 Purpose

Indian Language to Indian Language Machine Translation System (henceforth referred as ILMT System) will be a bidirectional machine translation system, to be developed for nine Indian language pairs, and hence, there would be 9 ILMT Systems, one for each language pair.

To distinguish these nine products, we will use three character code for each language as suffixes to name the product, i.e.,

- ILMT_Tam-Hin for Tamil Hindi
- ILMT_Tel-Hin for Telugu Hindi
- ILMT_Mar-Hin for Marathi Hindi
- ILMT_Ben-Hin for Bangla Hindi
- ILMT_Tam-Tel for Tamil Telugu
- ILMT_Urd-Hin for Urdu Hindi
- ILMT_Pan-Hin for Punjabi Hindi
- ILMT_Mal-Tam for Malayalam Tamil
- ILMT_Kan-Hin for Kannada Hindi

ILMT System is being developed by a consortium of academic & research institutions working in the field of Natural Language Processing (NLP) technology. All these institutions are active in the field of technology development for NLP in general & Indian Language Machine Translation in particular, for many years. They have developed and accumulated almost all components/modules that can be adapted, modified, or enhanced (as the need may be) for ILMT System being developed.

ILMT System being conceived will be a very large and complex system, and it will be inappropriate to design it afresh. The consortium has decided that the available versions of all modules would be taken as the ‘initial versions’, and then each module would be engineered step-wise, to make it a component of a maintainable ILMT software product.

This SRS is being written in the light of the constraints described in the earlier para. So the primary purpose of writing this SRS are

- It will provide a baseline for design & development of ILMT System w.r.t. functionality
- It will provide a baseline for validation & verification of the ILMT System
- It will help in the field deployment & maintenance of the ILMT System
• It will provide a basis for the **transfer of technology & enhancement** of ILMT System (as 
  the need arises in future).

1.2 Scope

The ILMT system will provide a web interface for translation. It will work on web pages or text material from 
books, magazines, newspaper etc, written in standard language. 
ILMT System is to be developed for two distinct user domains, i.e., *tourism & health*. So each of the nine 
products mentioned above would have three distinct package, viz.,

- ILMT System for General Purpose
- ILMT System for Tourism
- ILMT System for Health

1.3 Definitions, acronyms, and abbreviations

NLP – Natural Language Processing  
ILMT System – Indian Language to Indian Language Machine Translation System  
CML – Copora Markup Language  
SSF – Shakti Standard Format  
DSF – Dictionary Standard Format  
API – Application Program Interface  
nounp (CAT_) - It return true if the value of CAT_ is a lexical catogry of type noun.  
{other_}s – More than one feature structure,  
other abbreviation --

- the major list of abbreviation which is used in SSF to define the format is available in Appendix B.  
- the other major list of abbreviation which is used to define the attribute and its value for POS and Morph is available in references notes for standards for POS and Morph

1.4 References

- IEEE 830-1998  
- SSF, Dr. Rajeev Sangal, IIIT Hyderabad  
- Dashboard, Dr. Rajeev Sangal, IIIT Hyderabad  
- Notes of SRS, Expert Software Consultants Ltd., New Delhi
1.5 Overview

This SRS is organized into 3 main sections. They are Introduction, Overall description, and Specific requirements. The SRS ends with a list of Appendixes to make it complete in itself. Specific Requirements sections describes the functional as well as non-functional requirements.
2

Overall Descriptions

2.1 Product perspective

As a product ILMT System is to be developed for three distinct usage scenarios, i.e., general purpose, tourism domain, and health domain. The product will be used by the users on the web using a browser. So the system must be able to handle the web content appropriately.

2.2 Product Functions

The ILMT System will be based on the analyze-transfer-generate paradigm. The input text is first pre-processed (collected, cleaned, and formatted). Then, the analysis of the source language text is carried out. After source language analysis, transfer of vocabulary and analyzed structure is carried out. And finally the target language output is generated. The major product functions (or sub functions) of ILMT System will be

- Preprocessor
  - Collector
  - Cleaner, and
  - Formatter
- Source Language Analyzer
  - Tokenizer
  - Morph analyzer
  - Sandhi splitter (optional)
  - POS tagger
  - Chunker
  - Pruning
  - Head Computation
  - Vibhakti Computation
  - Named Entity Recognizer
  - Simple parser
- Source to Target language Transfer
  - Transfer Grammar
  - Lexical substitution
  - Transliteration
- Target language generator
2.3 User Characteristics

As a product ILMT System is to be developed for three distinct usage scenarios, i.e. general purpose, tourism domain, and health domain. The aim of the ILMT System is to develop a translation system where the following holds good with respect to the users,

- the user does not know the source language
- the user is a native speaker of target language,
- translated output is **comprehendable** i.e., the human user can make a meaning out of the translated output
- there is no major **ambiguity**, and lastly
- it is a **usable** system

Moreover, ILMT System presumes that the user can read target language script.

2.4 Constraints: System Structure/Architecture

The proposed ILMT System is being developed in a consortium mode (11 participating organizations). The researchers have developed over a period of time lot of Natural Language Processing (NLP) modules. These modules are all **functional** in a **specific environment** and in a **limited scope**. Since these NLP modules are complex they cannot be re-written/modified overnight. They are developed using various programming languages (like, C, Perl, lex, Java, C++, Python, etc.) and different paradigm or formalism. Due to the complexity of any NLP System, and the heterogeneity of the available modules, it is decided that ILMT System will be developed using Blackboard Architecture to provide inter-interoperability between heterogeneous modules. Hence all the modules will operate on a **common data representation called Shakti Standard Format (SSF)** either in-memory or stream.

All the modules that are being developed (or re engineered) will need to comply (or adapt) to the specifications of the blackboard. With the blackboard architecture it will be easy to configure & setup ILMT System with the heterogeneous modules.

In view of the above constraints, a separate application (called Dashboard) is being developed in parallel at IIIT Hyderabad, which will provide a framework for setting up and configuring ILMT.
System using blackboard architecture.
ILMT System validation can be done only against Test Suites, and hence, for each of the nine products every one of the three versions, there have to have test suits developed in advance. Further, each product is bidirectional, and hence, there should be a distinct test suite for each direction of Machine Translation Testing. As there are only 9 languages in all, we still need Test Suites for each of the nine languages, and for each of their three versions, i.e.,

We need to develop 54 Test Suits in all for this project. Following the same scheme as proposed for the products test suits will be named as

- TS_HIN_TEL
- TS_HIN_TAM
- TS_HIN_PAN
- TS_HIN_BEN
- TS_HIN_MAR
- TS_HIN_KAN
- TS_TAM_MAL
- TS_TAM_TEL
- TS_HIN_URD

Each of the above nine suits will have six distinct versions for three usage scenario and each direction.

2.5 Assumptions and dependencies

The proposed ILMT System will be based on analyze-transfer-generate paradigm. First, analysis of the source language text would be done, then a transfer of vocabulary and analyzed structure to target language would be carried out, and finally the target language would be generated.

Because Indian languages are similar and share grammatical structures, shallow parsing would be done. The transfer grammar component would be kept simple requiring only a simple parser. Domain specific aspects would be handled by building suitable named-entity recognizers, suitable dictionary entries, etc.

Shallow parsing would involve three major tasks: Morphological analysis, POS tagging and Chunking. The first would be rule based, the second statistical and the third a combination of the two. The overall processing task has been broken up into many modules, each one of them will typically perform a small logical task. The overall architecture of the system is given in Figure 1.

Functionality performed by each of the modules will be described in Specific requirements section. Each module does the core task mentioned against it, however besides the core task, it might also carry out some preprocessing and post-processing as may be required. For example, chunker module identifies the chunks. Thus, the chunker module does the core task of chunking. However, it might also perform post-processing in which the head of the chunk is identified and the head features copied as the chunk features.

All modules of ILMT operate on a common data representation called SSF. All modules read data in SSF and generate output in SSF. If a module is successful in its task, it typically, modifies value of one or more attributes.
in its output.

2.6 Software Engineering Approach (References: s/w engg process for ILMT)

As an ILMT System is a very large and complex system, it is difficult and hence \textit{inappropriate to design it afresh}. The consortium has decided that the available versions of some modules would be taken as the \textit{initial versions}, and then each module would be \textit{engineered step-wise}, to make it a component of a \textit{maintainable ILMT software product}.

The ILMT system is being built by contributions of various NLP research groups. The large task of building ILMT system is sub-divided into smaller tasks. Each participant group will take up one or more task and work in parallel.

The ILMT system consists of many modules, each one of which typically performs a logical task, usually the task is also small, so that when there is a change it is easier to make it on a small sized module.

Most of the modules are subdivided into a \textit{language independent engine} and \textit{language specific data}. This allows a software group to concentrate on building the engines. Similarly, a language group can contribute data for a specific language. The same engine works for different languages.

\textbf{SSF}

All modules operate on a stream of data whose format is fixed. This format is known as Shakti Standard Format (SSF). If a module is successful in its task, it adds a new attribute or analysis to output stream (in the same SSF). This approach helps to keep the local complexity of individual modules under tight control for most of the modules.

\textbf{Transparency}

The use of SSF format in ILMT also helps to achieve unprecedented transparency for input and output of every module. The textual SSF output of a module is not only for human consumption, but it is also used by the subsequent module in the data stream as its input.

\textbf{Handle Failure}

The SSF format also facilitates to dealing with failure at every level of module. For example, in case of failure to produce full analysis and some attributes remain unfilled by a module, the SSF output is still produced and, the downstream modules continue to operate on the data stream, albeit less effectively.

\textbf{ILMT System Platform}

The ILMT system will run on a Linux Server to provide translation service to clients connected to the web over intranet. The following are the ILMT System server and browser specifications.

\textbf{Server (ILMT)}

\begin{itemize}
  \item Operating System : Fedora Core 4 (Kernel - 2.6.11)
  \item Web Server (Apache) : httpd-2.0.49
  \item Data Base (GDBM) : gdbm-1.8.0
\end{itemize}
2.7 ILMT System: Accuracy / User satisfaction

Fully working version of the domain specific MT will be deployed with performance of 85% - 90%, and general purpose MT will be deployed with a performance of 80 – 85 %.

2.7.1 ILMT System : Test Suites

NLP System Testing can be done only against Test Suites, and hence, for each of the nine products every one of the three versions, there have to be test suits developed in advance. Further, each product is bidirectional, and hence, there should be a distinct test suite for each direction of Machine Translation Testing!

As there are only 9 languages in all, we still need Test Suites for each of the nine languages, and for each of their three versions, i.e.,

We need to develop 27 Test Suits in all for this project. Following the same scheme as proposed for the products we should name Test Suits as TS_Hin, TS_Tam, TS_Tel, TS_Ben, TS_Mar, TS_Urd, TS_Kan, TS_Pun & TS_Mal. Each of the above nine suits will have three distinct versions for three usage scenario.

2.8 Methodology

The ILMT system is being built by contributions of various NLP research groups. The large task of building ILMT system is sub-divided into smaller tasks. Each participant group will take up one or more task and work in parallel.

Most of the modules are subdivided into a language independent engine and language specific data. This allows a software group to concentrate on building the engine. Similarly, a language group can contribute data for a specific language.
3

Specific Requirements

3.1 System Structure

The proposed Indian language to Indian language machine translation system will be based on analyze-transfer-generate paradigm. First, analysis of the source language would be done, then a transfer of vocabulary and analyzed structure to target language would be carried out, and finally the target language would be generated.

Because Indian languages are similar and share grammatical structures, only shallow parsing would be done. The transfer grammar component would be kept simple. Domain specific aspects would be handled by building suitable named-entity recognizers, suitable dictionary entries, etc.

Shallow parsing would involve three major tasks: morphological analysis, POS tagging and chunking. The first would be rule based, the second statistical and the third a combination of the two. The overall processing task has been broken up into many modules, each one of them will typically perform a small logical task. The overall architecture of the system is given in Figure 1.

Each of the important modules is described below. Each module does the core task mentioned against it, however besides the core task, it might also carry out some preprocessing and post-processing. For example, chunker module identifies the chunks. Thus, the chunker module does the core task of chunking. However, it might also perform post-processing in which the head of the chunk is identified and the head features copied as the chunk features.

3.1.1 Common Representation SSF

All modules of ILMT operate on a common representation called SSF. All modules read data in SSF and generate output in the same format. If a module is successful in its task, it typically, modifies value of one or more attributes in its output.

**Introduction to Shakti Standard Format (SSF)**

The Shakti Standard Format (SSF) is used for representing the analysis of a sentence. The SSF representation is designed to keep both rule-based as well as statistical approaches together. Each module knows what to do when its desired information is available and uses default when it is not available. If a module is successful in its task, it modifies the relevant attributes, or in other words, adds its analysis to the representation. In section 3.2 example output of each module for a sample Hindi/Telugu sentence is shown. It is especially designed to represent the different kinds of linguistic analysis, as well as different levels of analysis. It can routinely represent partial analysis.
Two kinds of analyses are usually done:

- Constituent level analysis
- Relational-structure level analysis

The former is used to store simple phrase level analysis (called chunking) and the latter for storing relations between the simple phrases. Feature structures are used to store attribute-value pairs for a phrasal node as well as for a word or a token. Attribute value pairs are also used to store relations, as will be seen later. Outputs of many other kinds of analysis, such as grammatical relations, TAM computation, case computation, dependency relations, word sense disambiguation etc. are stored using feature-structures.

Though, the SSF format is fixed, it is extensible to handle new features. It also has a text representation, which makes it easy to read the output.

The following example illustrates the SSF. The following English sentence,

Children are watching some programmes on television in the house. - (1)

Contains the following chunks (enclosed by double brackets),

((Children)) [[are watching]] ((some programmes)) ((on television)) ((in the house))

All the chunks are noun phrases, except for one ('are watching') which is a verb group and is shown enclosed in square brackets.

Though, the SSF format is fixed, it is extensible to handle new features. It also has a text representation, which makes it easy to read the output.

In the sentence above, all the chunks are noun phrases, except for one ('are watching') which is a verb group and is shown enclosed in square brackets.

The above sentence chunks can be represented in the form of tree structure as shown in figure 2.

![Figure 2: Tree structure of a sentence](image)

The SSF representation of the above tree structure would be as follows as shown in figure 3 which stores the tree or forest address of each word or group.

The below figure has two columns which describe following property of tree structure.

- Column1 stores the node address. (It is mainly for human readability.)
Column 2 stores the word or word-group (chunk found in sentence). It includes the symbol ‘((‘ to represent the start of the word or word-group and the symbol ‘))’ to represent the end of the word or word-group.

Column 3 stores the chunk name, it also stores the POS tag of the words occurred in sentence (here POS is not shown which will be added in this column after POS analysis, as describe after diagram)

If the sentence were passed through POS analysis and chunking, part-of-speech tag for each word as well chunks would be marked as follows:

```
(Children_NNS)
[are_VBP watching_VBG]
(some_DT programmes_NNS)
(on_IN television_NN)
(in_IN the_DT house_NN)
```

The SSF representation of the above tree structure would be as follows as shown in figure 4 along with chunk labels and POS tag, in the third column.
The example below shows other features of each word of sentence, such as root, lexical, category, gender, and head of chunk etc, in the fourth column called $fs$ or feature structure:

$$1 \quad \text{[} \text{NP} \quad \{ \text{fs root=child, locat=np, gend=m, num=p, pers=3, head=children} \} \text{] children NNS} \quad \{ \text{fs root=child, locat=n, gend=m, num=p, pers=3, case=8, name=children} \}$$

- The number “1” in “head=1” in $fs$, represents the local node number of the chunk which helps to know the head of the sub-tree without traversing every leaf of sub-tree of chunk.
- Some frequently occurring attributes (such as root, cat, gend, etc.) may be abbreviated using a special attribute called ‘af’ or abbreviated attributes, as follows:

$$1 \quad \text{[} \text{NP} \quad \{ \text{af=child, n, m, p, 3, 8, } \} \text{] children NNS} \quad \{ \text{af=child, n, m, p, 3, 8, } \}$$

Note:

For noun, attribute name is cm
For verb, attribute name is tam
The field for each attribute is at a fixed position, and a comma is used as a separator. Thus, in case, no value is given for a particular attribute the field is left blank, e.g. last two fields in the above example.

The representation in SSF of sentence [1] with feature structures is given in Fig. 5 (abbreviated attribute ‘af’ is used).

```
1   [ [   MP  <fs af='child,n.m,p,3,8,' head=children>
1.1  children  NMS  <fs af='child,n.m,p,3,8,' name=children>
   ] ]
2   [ [   VB  <fs af='match,v.m.s,3,8,' aspect=PROG, head=matching>
2.1  are    UBP  <fs af='be,v.m.p,3,8,'/>
2.2  matching  UBG  <fs af='match,v.m.s,3,8,' aspect=PROG name=matching>
   ] ]
3   [ [   MP  <fs af='programme,n.m,p,3,8,' head=progress>
3.1  some   OT   <fs af='some,det,m.s,3,8,'/>
3.2  programmes  NMS  <fs af='programme,n.m,p,3,8,' name=progress>
   ] ]
4   [ [   PP  <fs af='television,n.m.s,3,8,' head=television>
4.1  on    IN    <fs af='on,p.m.c,3,8,'/>
4.1.1 [ [   MP  <fs af='television,n.m.s,3,8,' head=television>
4.1.2  television  NN  <fs af='television,n.m.s,3,8,' name=television>
   ] ]
5   [ [   PP  <fs af='house,n.m.s,3,8,' head=house>
5.1  in    IN    <fs af='in,p.m,s,3,8,'/>
5.2  [ [   MP  <fs af='house,n.m.s,3,8,' head=house>
5.2.1  the   OT   <fs af='the,det,m.s,3,8,'/>
5.2.2  house  NN  <fs af='house,n.m.s,3,8,' name=house>
   ] ]
   ] ]
```

**Fig. 5: Shafe Standard Format**

### 3.2 ILMT System Architecture

The proposed Indian language to Indian language machine translation system will be based on analyze-transfer-generate paradigm. First, analysis of the source language text would be done, then a transfer of vocabulary and analyzed structure to target language would be carried out, and finally the target language text would be generated.

Because Indian languages are similar and share grammatical structures, only shallow parsing would be done. The transfer grammar component would be kept simple. Domain specific aspects would be handled by building suitable named-entity recognizers, suitable dictionary entries, etc.

Shallow parsing would involve three major tasks: morphological analysis, POS tagging and chunking. The first would be rule based, the second statistical and the third a combination of the two.

Due to the complexity of the NLP system and the heterogeneity of the modules, it is decided to design ILMT System using Blackboard architecture. The overall system will be made up of many modules, each one of them will typically perform a small logical task. Each module will operate on a common data structure (either in-memory or stream). The overall architecture of the system is shown in Figure 1.
Figure 1. Architecture of ILMT System

Note: Here Simple Parser does not mean a full sentential parser. It means identifying relation between some of the chunks of the sentence. Some of these relations will depend on the language being analyzed.

The modules will be designed/adapted to access the common in-memory data structure of the blackboard. The common data structure would be SSF. An SSF API shall be provided to access SSF data.
3.3 Information Flow

All modules will operate on a common data representation, i.e., SSF. The Dashboard module will provide an infrastructure to setup, configure and define the dependency of the modules of the ILMT System. Information flow/sharing among the modules of ILMT System is shown in Figure 2. Each module reads one or more system-properties in a given SSF and writes/modifies one or more system-properties of the SSF. The data flow between the

![Diagram of Information Flow](image)

Figure 2 – Information Flow
3.4 Process Description

All modules of ILMT operate on a common data representation i.e., SSF. The SSF representation for a sentence consists of a sequence of trees. Each tree is made up of one or more related nodes. A node has system properties which are given by prop-name and prop-value. All modules read input data in SSF and generate output data in SSF format. If a module is successful in its task, it typically, modifies system-properties of the nodes or the user-defined attributes in the Feature Structures (the fourth system-property).

Thus in the pipe-line of modules shown in the architecture diagram, each module adds or modifies the specified attributes. Other attributes produced by earlier modules are carried forward in the pipeline. Process descriptions of each of the modules is described below.

3.5 Specification of Individual Modules

Specification of each modules is given below. Specification of each module indicates what attribute-value(s) it uses as input and what attribute it uses as output.

3.5.1 Preprocessor

Preprocessor will provide an interface for the ILMT System to the web. This module will collect the input text from the webserver which is in the form of HTTP request coming from the user. It would clean the text. Cleaning would involve stripping the web content of its rich format. It may optionally convert one character encoding to another. Minimally it should support Unicode, WX-Roman, and ISClI.

3.5.2 Tokenizer

Tokenizer converts a given input text into a sequence of tokens (consisting of words, punctuation marks, and other symbols) with a sentence marker after end of each sentence of input text. The output is produced in SSF format. The input to Tokenizer can also be a CML text.

The sentence marker will depend on the type of character representation used for the text input data. It could be either of these (full stop, sign of exclamation, sign of introgation, and PURNA VIRAM depending upon the character encoding).

Tokenizer will be configured for handling special tokens for each language, like, for Hindi language, the tokens like, शी॰, ड, पो॰ will not be split.

Roman symbols inside the text will be left as it is and shall be preceded by an “@” sign in the output.

Input Output Specification

Input to Tokenizer will be Text file or a text data that is in Corpora Markup Language (CML) format.

Output from a Tokenizer will be a SSF having 3 mandatory properties, i.e., ADDR_, TKN_, and CAT_.
**Input**: Input to Tokenizer will be Text file or a CML file.

(TKN_) +

**Output**: Output from a Tokenizer will be a SSF having 3 mandatory properties, i.e., ADDR_, TKN_, and CAT_ with sentence id number. i.e Word level Token and Sentence id in text level SSF.

```
sen_ssf ::= $snt_hdr $row+$ snt_footer
snt_hdr ::= 'Sentence id = '''[0-9]+ '">
$snt_footer ::= '</Sentence>'
$row ::= ^$addr_ ($tkn_) ($cat_) ($alt_fs)? \n
   (t) '})'
```

**Example**

**Input**

Usa ladake ko rAXA kA kelA KAnA padZA

**Output**

```
<Sentence id="1">
1 usa unk
2 ladake unk
3 ko unk
4 rAXA unk
5 kA unk
6 kelA unk
7 KAnA unk
8 padZA unk
</Sentence>
```

### 3.5.3 Sandhi Splitter

This module will be used for those languages in which sandhi takes place. It will take a composite word as input and its output will be simple words.

For many of the Indian languages sandhi occurs and two or more words are merged (Sandhi) into a composite word. To perform morphological analysis on such words, it is necessary to first split the composite word into two or more simple words.

For some words even after splitting morphological analysis may not be possible. For such words Sandhi splitter leaves the word as it is with lexical category attribute in FS set to unknown (“unk”).

**Input:Output Specification**

Input to Sandhi splitter is an SSF. It will have 3 properties set, they are ADDR_, TKN_, and CAT_. The fourth property may optionally contain Feature Structure (FS) values.

If there is a successful split of the composite word then new nodes are added to the SSF, and system properties
like ADDR_, TKN_ and CAT_ is added.

**Input:** TKN_, as ssf

```
**sen_ssf** ::= $**snt_hdr** ($**row**)+ $**snt_footer**

$**snt_hdr** ::= ‘<Sentence id =’ \[0-9]\+ ‘>’

$**snt_footer** ::= ‘</Sentence>’

$**row** ::= ^$**addr_** (\t) ($**tkn_**) (\t) ($**cat_**) (\t $salt_fs)? \n
            \n
          \n
          ) (\t ‘)’)
```

**Output:** ADDR_, TKN_, CAT_

( alternate way to write a R.E of SSF having column attributes Address, token, Category )

**Example:** (Here the example is shown from Telugu and not for Hindi )

**Input**

```
<Sentence id="1">
1 peVIYlikUwuru unk
2 ABaraNALannI unk
3 cakkagA unk
4 alaMkariMcukuMxi unk
5 .
</Sentence>
```

**Output**

```
<Sentence id="1">
1 peVIYlikUwuru unk
2 ABaraNALu unk
3 annl unk
4 cakkagA unk
5 alaMkariMcukuMxi unk
6 .
</Sentence>
```

### 3.5.4 Morph Analyzer

The morph analyzer identifies the root and the grammatical features of a given word. For languages that are not rich in inflections, a simple lookup dictionary that contains all the word forms would be sufficient. But creating such a dictionary for inflectionally rich languages like Tamil or Telugu is nontrivial and requires huge storage and high performance computing. The best alternative is to have a dictionary of root words and employ an analyzer to split the word form into its root and grammatical features.

Paradigm-based approaches for Morph Analyzers have given good success rates for Indian languages.

( Appendix : Morph standard Workshop Document)
**Input Output Specifications**

The input to Morph analyser is an SSF, comprising of 3 mandatory properties, ADDR_, TKN_, and CAT_. The output from a Morph Analyser will be an SSF. Feature Structure (FS) system property will get updated by Morph. Feature Structure is a list of attribute-value pairs. The attributes could be root, lcat, gend, num, pers, case, cm, suff, head, name, etc. The attribute-value pairs like (root, lcat, gend, num, pers, case, cm, suff) can also be represented as a composite attribute list called *Abbreviated Features* (af).

**Input:** TKN_ as

$\text{row} ::= ^{^\text{addr}_} (t) (^{_{\text{tkn}_}}) (t) (^_{\text{cat}_})$

**Output:** FS, the attributes are, root, lcat, gend, pers, num, case, cm, suff, and other user defined attribute-value pairs, or term as \{other\}.

$\text{row} ::= ^{^\text{addr}_} (t) (^{_{\text{tkn}_}}) (t) (^{_{\text{cat}_}}) (^{_{\text{alt_fs}}})$

where, $\text{fs} ::= \langle \text{fs} \rangle$ (key = \{alt_val\}) + \{\rangle

**Example:**

**Input**

```xml
<Sentence id="1">  
1 usa unk  
2 ladake unk  
3 ko unk  
4 rAXA unk  
5 kA unk  
6 kelA unk  
7 KAnA unk  
8 padZA unk  
</Sentence>
```

**Output**

```xml
<Sentence id="1">  
1 usa unk <fs af='vaha,pn,any,sg,3,0,,'>  
2 ladake unk <fs af='ladakA,adj,m,sg,3,0,,'>  
3 ko unk <fs af='ko,psp,,,,,,'>  
4 rAXA unk <fs af='rAXA,n,f,sg,3,,,,'>  
5 kA unk <fs af='kA,psp,,,,,,'>  
6 kelA unk <fs af='kelA,n,m,sg,3,d,,,'>  
7 KAnA unk <fs af='KAnA,n,m,sg,3,0,,'> <fs af='KAnA,n,m,sg,3,d,,,'>  
8 padZA unk <fs af='padZA,v,m,sg,any,,yA,yA'>  
</Sentence>
```
3.5.5 POS Tagger

Part of Speech tagging is the process of assigning a part of speech tag to each word (token) in the sentence. Identification of the parts of speech such as nouns, verbs, adjectives, adverbs for each word of the sentence helps in analyzing the role of each word (token) in a sentence. There are a number of approaches, such as rule-based, statistics based, transformation-based etc. which are used for POS tagging. Here we propose to use statistical approaches. (HMM based)
(Appendix: POS Standard Workshop Document)

Input/Output Specification
Input SSF to POS Tagger must have 3 system properties (ADDR_, TKN_, CAT_) defined for each of the nodes. Having the value of CAT_ is “unk” as unknown. It reads the TKN_ property and updates the CAT_ property by a valid lcat(CAT_).

Input: TKN_
As $row ::= ^addr_ (t) (tkn_) (t) (cat_) (t alt_fs) \n -->{cat_ = "unk" nk}$
(The morphological features used will vary slightly for different languages.)

Output: CAT_ as pcat
   $row ::= ^addr_ (t) (tkn_) (t) (cat_) (t alt_fs) \n$
The input specifications above means that it uses or requires property TKN_ to be available in the input SSF. For example, the tokens in the input sentence are given.
The output specifications above says that attribute ‘CAT_’ will get defined or modify by this module.

Example:
Input
<Sentence id="1">
1 usa unk <fs af='vaha,pn,any,sg,3,0,,,'>
2 ladake unk <fs af='ladakA,adj,m,sg,3,o,,,'>
   |<fs af='ladakA,adj,m,pl,3,d,,,'>
   |<fs af='ladakA,adj,m,pl,3,o,,,'>
   |<fs af='ladakA,n,m,pl,3,d,,'>
3 ko unk <fs af='ko,psp,.....,'>
4 rAXA unk <fs af='rAXA,n,f,sg,3,,,'>
5 kA unk <fs af='kA,psp,.....,,'>
6 kelA unk <fs af='kelA,n,m,sg,3,d,,,'>
7 KAnA unk <fs af='KAnA,v,any,any,any,,nA,nA'>
   |<fs af='KAnA,n,m,sg,d,,'>
8 padZA unk <fs af='padZA,v,m,sg,any,,yA,yA'>
</Sentence>
3.5.6 Chunker

This module is sub-divided into four parts

3.5.6.1 Chunking

Chunking identifies simple noun phrases, verb groups, adjectival phrases, and adverbial phrases in a sentence. This involves identifying the boundary of chunks and the label.

Input-Output Specification:
Input SSF to Chunker must have TKN_ and CAT_ system properties set for each node. After chunking new nodes will get added to the SSF. These nodes will be identify the chunks. The phrasal category will be assigned to the system-property CAT_ for these new nodes.

Input: ADDR_, TKN_, CAT_,
Output: ADDR_ (This mean restructuring takes place, because this is when value of ADDR_ changes), TKN_, CAT_

newrow()->{CAT_ , TKN_}

The input specifications above means that it uses or requires property TKN_ and CAT_ to be defined in the SSF. In other words, the words or tokens in the input sentence together with part of speech category are given. The output specifications above mean that attribute ‘ADDR_’ will get re-defined by this module.
Example:

Input

```
1  usa  DEM  <fs af='vaha,pn,any,sg,3,0,,'/>
2  ladake  NN  <fs af='ladakA,adj,m,sg,3,o,,'/>
    |<fs af='ladakA,adj,m,pl,3,d,,'/>
    |<fs af='ladakA,adj,m,pl,3,o,,'/>
    |<fs af='ladakA,n,m,pl,3,d,,'/>
    |<fs af='ladakA,n,m,sg,3,o,,'/>
3  ko  PSP  <fs af='ko,psp,,...,'>
4  rAXA  NN  <fs af='rAXA,n,f,sg,3,,,'/>
5  kA  PSP  <fs af='kA,psp,,...,'>
6  kelA  NN  <fs af='kelA,n,m,sg,3,d,,'/>
7  KAnA  VM  <fs af='KA,v,any,any,any,,nA,nA'>
    |<fs af='KAnA,n,m,sg,,d,,'/>
8  padZA  VAUX  <fs af='padZa,v,m,sg,any,,yA,yA'>
```

Output

```
1  ((
1.1  usa  DEM  <fs af='vaha,pn,any,sg,3,0,,'/>
1.2  ladake  NN  <fs af='ladakA,adj,m,sg,3,o,,'/>
    |<fs af='ladakA,adj,m,pl,3,d,,'/>
    |<fs af='ladakA,adj,m,pl,3,o,,'/>
    |<fs af='ladakA,n,m,pl,3,d,,'/>
    |<fs af='ladakA,n,m,sg,3,o,,'/>
1.3  ko  PSP  <fs af='ko,psp,,...,'>
   ))
2  ((
2.1  rAXA  NN  <fs af='rAXA,n,f,sg,3,,,'/>
2.2  kA  PSP  <fs af='kA,psp,,...,'>
   ))
3  ((
3.1  kelA  NN  <fs af='kelA,n,m,sg,3,d,,'/>
   ))
4  ((
4.1  KAnA  VM  <fs af='KA,v,any,any,any,,nA,nA'>
    |<fs af='KAnA,n,m,sg,,d,,'/>
4.2  padZA  VAUX  <fs af='padZa,v,m,sg,any,,yA,yA'>
   ))
```

Description: Simple NPs are grouped as noun chunks, and verb sequences are grouped as verb chunks (VG), Similarly for adjectival chunks (JJP) and adverbial chunks (RBP).
3.5.6.2 Pruning

Pruning identifies the most appropriate feature structures out of different structures available after Morph Analysis by checking compatibility with the CAT_, guessing etc. This module is further divided into three parts.

3.5.6.2.1 Morph Pruning

It takes that feature structure where the lcat value matches with the CAT_ value. All those feature structures which whose lcat is compatible with the pos tag those are retained as possible outputs for the given token. Rest of the feature structures will be pruned(removed) by this module. In case there isn't any feature structure whose lcat is matching with the CAT_ then all the feature structures are retained and a new attribute value pair poslcat="NM" is added to every feature structure. NM stands for “not matched”.

Input Output Specifications

Input to Morph Pruning is an SSF with one or more FS values for each of the TKN_.
If CAT_ and any of the lcat match then the matched ones are kept.

If CAT_ doesn't match with any of the lcat (in all the feature structures), then

\[
\text{lexcat}(\text{CAT}_-) \rightarrow \{\text{poslcat}="NM"\}
\]

The notation used in output specification says that if CAT_ is a lexical category (predicate ‘lexcat’) then the attribute value pair poslcat="NM” gets added when there is no match.

Input: \{other\}_s
Output: \{other\}_s \rightarrow \{pcat_ ==lcat\}

\{other-, poslcat='NM'} \rightarrow \{\text{ALL}{\other}_s \rightarrow \{pcat =! lcat\}\}

Example:

Input:

\[
\text{<Sentence id=}'1'>
\]

1 (((
1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'>
1.2 ladake NN <fs af='ladakA,adj,m,sg,3,o,,'>
\mid<fs af='ladakA,adj,m,pl,3,d,,'>
\mid<fs af='ladakA,adj,m,pl,3,o,,'>
\mid<fs af='ladakA,adj,m,pl,3,o,,'>
\mid<fs af='ladakA,n,m,sg,3,o,,'>
\mid<fs af='ladakA,n,m,sg,3,o,,'>
1.3 ko PSP <fs af='ko,psp,,,'>
))
2 (((
2.1 rAXA NN <fs af='rAXA,n,f,sg,3,,'>
2.2 kA PSP <fs af='kA,psp,,,'>
3.5.6.2.2  Guess Morph

This module uses heuristics to reduce the multiple numbers of feature structures left by the morph pruning. This module will be defined by each individual language groups for their respective languages.

Input-Output Specification:

Input: {other_}s

Output:{other_}s --> { R1=" True ", ....}
Example:
Rule: W1 and W2 are consecutive words.
\[ \text{num}(W1) = \text{sg} \Rightarrow \text{num}(W2) = \text{sg} \]

Once the above rule is applied on a sentence with word W1 having num=sg, then these feature structures of W2 are retained which also have num=sg. (Note: Each language group may write such rules as programs depending on their requirement.)

The fs of \textit{ladake} which has the number=“sg” is retained based on the rule in the example below.

\begin{verbatim}
Input
<Sentence id="1">
1  (( NP
 1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'>
 1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,,'>
      |<fs af='ladakA,n,m,sg,3,o,,'>
 1.3 ko PSP <fs af='ko,psp,,,,,'>
 )
 2  (( NP
 2.1 rAXA NN <fs af='rAXA,n,f,sg,3,,,,,'>
 2.2 kA PSP <fs af='kA,psp,,,,,'>
 )
 3  (( NP
 3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,'>
 )
 4  (( VGF
 4.1 KAnA VM <fs af='KAnA,v,any,any,any,,nA,nA'>
 4.2 padZA VAUX <fs af='padZA,v,m,sg,any,,yA,yA'>
 ))
</Sentence>

Output:
<Sentence id="1">
1  (( NP
 1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'>
 1.2 ladake NN <fs af='ladakA,n,m,sg,3,o,,'>
 1.3 ko PSP <fs af='ko,psp,,,,,'>
 )
 2  (( NP
 2.1 rAXA NN <fs af='rAXA,n,f,sg,3,,,,,'>
 2.2 kA PSP <fs af='kA,psp,,,,,'>
 )
 3  (( NP
 3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,'>
 )
 4  (( VGF
 4.1 KAnA VM <fs af='KAnA,v,any,any,any,,nA,nA'>
 ))
</Sentence>
\end{verbatim}
3.5.6.2.3 Pick one Morph

It will pick only one feature structure based on the selection definition given to it. By default it will pick the first feature structure.

The attribute value pair poslcat = NM will remain there, if it is tagged once.

**Input-Output Specification:**
Input to Pruning is an SSF with one or more FS values for each of the TKN_. Output will be only one feature structure.

**Input:** {other_}1,{other_}2,{other_}3,.....
**Output:** {other_}1

**Example:**

**Input:**
```
<Sentence id="1">( ( NP
1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'/>
1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,,'/>
1.3 ko PSP <fs af='ko,psp,......,'/>
 ) )

2 ( ( NP
2.1 rAXA NN <fs af='rAXA,n,f,sg,3,,,,'/>
2.2 kA PSP <fs af='kA,psp,......,'/>
 ) )

3 ( ( NP
3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,,,'/>
 ) )

4 ( ( VGF
4.1 KAnA VM <fs af='KA,v,any,any,any,,nA,nA'>
4.2 padZA VAUX <fs af='padZa,v,m,sg,any,,yA,yA' ></fs>
 ) )
</Sentence>
```

**Output:**
```
<Sentence id="1">( ( NP
1 ( ( NP
1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'/>
1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,,'/>
1.3 ko PSP <fs af='ko,psp,......,'/>
 ) )

2 ( ( NP
2.1 rAXA NN <fs af='rAXA,n,f,sg,3,,,,'/>
2.2 kA PSP <fs af='kA,psp,......,'/>
 ) )

3 ( ( NP
3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,,,'/>
 ) )

4 ( ( VGF
4.1 KAnA VM <fs af='KA,v,any,any,any,,nA,nA'>
4.2 padZA VAUX <fs af='padZa,v,m,sg,any,,yA,yA' ></fs>
 ) )
</Sentence>
```
3.5.6.3 Head computation

This module computes the head of the chunk. A child node is identified as head of the chunk. A new feature called name is added to this child node with attribute as name. Attribute name is then set to an arbitrary but unique string that sounds close to root for readability. The chunk node inherits the features of the head child. All the features are copied from the head-child to parent chunk except ‘name’. A new attribute called head is added to the feature of the chunk node. Whose value is the name-string just assigned to the head child.

Input-Output Specification:
Input SSF must have TKN_, CAT_, and FS defined for each of the child nodes in each of the chunks. In case of chunk node, CAT_ must be defined.
Output SSF will have one of the child node in each chunk identified as head of the chunk. Each chunk node will have head attribute set in its FS.
Input: TKN_, CAT_, FS
Output: lexcatp(CAT_) -> {name}
chunkcatp(CAT_) -> {head}

The notation used in output specification above says that if CAT_ is a lexical category (predicate ‘lcat’) then the value of attributes name may get added AND if it is a chunk (‘bagcatp’), then the attribute head may get added.

Example:
Input
<Sentence id="1"> 1  ((   NP 2  (( rAXA NN <fs af='rAXA,n,f,sg,3,,,,'> 2.1 kA PSP <fs af='kA,psp,,,,,,,' > 2.2 )) 3  (( 3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,'> 3.2 )) 4  (( 4.1 KAnA VM <fs af='KA,v,any,any,any,,nA,nA'> 4.2 padZA VAUX <fs af='padZa,v,m,sg,any,,yA,yA'> 4.3 )) </Sentence>
3.5.6.4 Inherit head features

All the features are copied from the head-child to parent chunk except ‘name’.

Input-Output Specification:

Input: ANY_

Output: bagcatp(CAT_) → ANY_ (copied attributes)

Example:

Input

<Sentence id="1">
1 (( NP <fs af='ladakA,n,m,pl,3,d,,' head="ladake"> 1.1 usa DEM <fs af='vaha,pn,any,sg,3,0,,'>
1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,,' name="ladake"> 1.3 ko PSP <fs af='ko,psp,.....,'>
))
2 (( NP <fs af='rAXA,n,f,sg,3,,,,' head="rAXA"> 2.1 rAXA NN <fs af='rAXA,n,f,sg,3,,,,' name="rAXA"> 2.2 kA PSP <fs af='kA,psp,.....,'>
))
3 (( NP <fs af='kelA,n,m,sg,3,d,,' head="kelA"> 3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,' name="kelA">)
))
4 (( VGF <fs af='KA,v,any,any,any,,nA,nA' head="KAnA"> 4.1 KAnA VM <fs af='KA,v,any,any,any,,nA,nA' name="KAnA"> 4.2 padZA VAUX <fs af='padZa,v,m,sg,any,,yA,yA'>
))
</Sentence>
Output
<Sentence id="1">
1 ( () NP <fs af='ladakA,n,m,pl,3,d,,' head="ladake"> 
1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,'>
1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,,' name="ladake"> 
1.3 ko PSP <fs af='ko,psp,......,'>
) 
2 ( () NP <fs af='rAXA,n,f,s,3,,' head="rAXA"> 
2.1 rAXA NN <fs af='rAXA,n,f,s,3,,' name="rAXA"> 
2.2 kA PSP <fs af='kA,psp,......,'>
) 
3 ( () NP <fs af='kelA,n,m,sg,3,d,,' head="kelA"> 
3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,' name="kelA"> 
) 
4 ( () VGF <fs af='KA,v,any,any,any,,nA,nA' head="KAnA"> 
4.1 KAnA VM <fs af='KA,v,any,any,any,,nA,nA' name="KAnA"> 
4.2 padZA VAUX <fs af='padZa,v,m,sg,any,,yA,yA'> 
) 
</Sentence>

(In the current implementation tasks in head computation and head feature inheritance are performed as part of head computation).

3.5.7 Local Word Grouper/Splitter

3.5.7.1 Local Word Grouper / Vibhakti Computation

Local word grouper does the technical task of vibhakti computation. The main task here is to group the function
words with the content words based on local information. One of the techniques to achieve this task is using kriya rupa charts. These charts specify the groups to be formed out of the sequence of verbs which denote a single action. Auxiliary verbs that appear as separate words are not interpreted as individual content words themselves. This is because they specify grammatical features of the chunk whose head is the main verb. The same is true for postpositions called noun vibhakti, which appear as separate words in Hindi. The local word grouper will handle this problem efficiently with appropriate complex and compound word group formation rules. It will group case/tam marker with the root for noun and verbs.

This module computes the case/tam features of noun/verb chunks and adds them to FS.

**Input-Output Specification:**

**Input:**

\[
\begin{align*}
\text{TKN}_-, \text{CAT}_- \\
\text{nounp } (\text{CAT}_-) & \rightarrow \{ \text{Cm}, \text{suffix}, \text{gend}, \text{num}, \text{pers}\} \\
\text{verbp } (\text{CAT}_-) & \rightarrow \{\ldots \text{.tense}\}
\end{align*}
\]

**Output:**

\[
\begin{align*}
\text{NPp}(\text{CAT}_-) & \rightarrow \{\text{Cm}, \text{partcle}, \text{gend}, \text{num}, \text{pers}\} \\
\text{VGp}(\text{CAT}_-) & \rightarrow \{\text{tam}, \text{neg}, \text{emph}, \text{gend}, \text{num}, \text{pers}\}
\end{align*}
\]

The above value shows that if \text{CAT}_- is noun then attributes Cm/tam and particle are in output list. Similarly for verb ( \text{CAT}_-).

The input specification states that this module will take \text{TKN}_- and \text{CAT}_- and other features as its input.

If value of \text{CAT}_- is a noun, it reads the value of gender, number etc and if value of \text{CAT}_- is verb, it reads tense in addition to the feature as gend, pers, etc.

**Example:**

**Input**

\[<\text{Sentence id="1">\]

1. (( NP <fs af='ladakA,n,m,pl,3,d,,,' head="ladake">  
1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'>  
1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,,,' name="ladake">  
1.3 ko PSP <fs af='ko,psp,,,,,'>

2. (( NP <fs af='rAXA,n,f,sg,3,,,,' head="rAXA">  
2.1 rAXA NN <fs af='rAXA,n,f,sg,3,,,,' name="rAXA">  
2.2 kA PSP <fs af='kA,psp,,,,,'>

3. (( NP <fs af='kelA,n,m,sg,3,d,,,' head="kelA">  
3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,,' name="kelA">  

4. (( VGF <fs af='KA,v,any,any,any,,nA,nA head="KAnA">  
4.1 KAnA VM <fs af='KA,v,any,any,any,,nA,nA' name="KAnA">  
4.2 padZA VAUX <fs af='padZa,v,m,sg,any,,yA,yA'>

</Sentence>\]
Output

3.5.8 Named Entity Recognizer (NER)

Named Entity recognition plays an important role in Language analysis phases. The Named Entity Recognition task requires entities mentioned in the document to be detected, their sense disambiguated, select the attributes to be assigned to the entity and represent it with a tag. An entity is an object or set of objects in the world. The Named entity hierarchy is divided into three major classes; Name, Time and Numerical expressions. The task of an NER engine is to identify the class in which an entity falls and give the relevant tag.

Input-Output Specification

Input: Pre-processed Text, having the name of noun.

\[
\text{nounp(tkn_)}
\]

Output: The output will have attribute-value pair of attribute named by ENAMEX_TYPE, SUBTYPE_1

\[
\{\text{other_}\} \rightarrow \{\text{ENAMEX_TYPE, SUBTYPE_1}\}
\]

Example:

input:

\[
\begin{align*}
1 & (( NP \quad <fs af='ladakA,n,m,pl,3,d,0_ko,' head='ladake'> \\
1.1 & usa \quad DEM \quad <fs af='vaha,pn,any,sg,3,o,,'> \\
1.2 & ladake \quad NN \quad <fs af='ladakA,n,m,pl,3,d,,' name='ladake'> \\
2 & (( NP \quad <fs af='rAXA,n,f,sg,3,,0_kA,' head='rAXA'> \\
2.1 & rAXA \quad NN \quad <fs af='rAXA,n,f,sg,3,,,' name='rAXA'> \\
3 & (( NP \quad <fs af='kelA,n,m,sg,3,d,,' head='kelA'> \\
3.1 & kelA \quad NN \quad <fs af='kelA,n,m,sg,3,d,,' name='kelA'> \\
4 & (( VGF \quad <fs af='kAnA,v,any,any,any,,nA_padZa+yA,' head='kAnA'> \\
4.1 & kAnA \quad VM \quad <fs af='kAnA,adj,m,sg,,d,,' name='kAnA'>
\end{align*}
\]
3.1 kelA NN <fs af='kelA,n,m,sg,3,d,, name="kelA"> )
4 (( VGF <fs af='KA,v,any,any,any,,nA_padZa+yA,' head="kAnA"> 4.1 kAnA VM <fs af='kAnA,adj,m,sg,,d,, name="kAnA"> )
</Sentence>

OUTPUT:
<Sentence id="1"> 1 (( NP <fs af='ladakA,n,m,pl,3,d,0_ko,' head="ladake"> 1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'> 1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,, name="ladake"> )
2 (( NP <fs af='rAXA,n,f,sg,3,,0_kA,' head="rAXA"> 2.1 rAXA NN <fs af='rAXA,n,f,sg,3,, name="rAXA" ENAMEX_TYPE="PERSON" SUBTYPE_1="INDIVIDUAL"> )
3 (( NP <fs af='kelA,n,m,sg,3,d,, name="kelA"> 3.1 kelA NN <fs af='kelA,n,m,sg,3,d,, name="kelA"> )
4 (( VGF <fs af='KA,v,any,any,any,,nA_padZa+yA,' head="kAnA"> 4.1 kAnA VM <fs af='kAnA,adj,m,sg,,d,, name="kAnA"> )
</Sentence>

3.5.9 Simple Parser

This module identifies relation between some of the chunks in a sentence. Some of these relations will depend on the language being analyzed.

Input-Output Specification:

Example:

Input
<Sentence id="1"> 1 (( NP <fs af='ladakA,n,m,pl,3,d,0_ko,' head="ladake"> 1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'> 1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,, name="ladake"> )
2 (( NP <fs af='rAXA,n,f,sg,3,,0_kA,' head="rAXA"> 2.1 rAXA NN <fs af='rAXA,n,f,sg,3,, name="rAXA" ENAMEX_TYPE="PERSON" SUBTYPE_1="INDIVIDUAL"> )
3 (( NP <fs af='kelA,n,m,sg,3,d,, name="kelA"> 3.1 kelA NN <fs af='kelA,n,m,sg,3,d,, name="kelA"> )
</Sentence>
The attribute 'drel' is introduced to state the relation between two chunks.

3.5.10 Lexical Sense Disambiguation

The root words identified by the morphological analyzer are disambiguated and the sense id of each word in the input text is identified. These words are then substituted by their target language equivalents as available in the Multilingual Dictionary.

**Input-Output Specification:**

**Input:** Chunked file in SSF format in the source language.

**Output:** Chunked file in SSF format with the source language words replaced by their lexical equivalents in the target language.

This module disambiguates each word in the input text and returns the sense tagged text as output.

**Example:**

**Input**: Hindi (source) file

```xml
1  (( NP  
1.1 उ PRP <fs af='व,pn,m,sg,,,'>
1.2 लड़के NN <fs af='लड़का,n,m,sg,,,'>
</Sentence>
```

```xml
Output:
1  (( NP  
1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'>
1.2 ladake NN <fs af='ladaKA,n,m,pl,3,d,0_ko,' head="ladaKE">  
</Sentence>
```

```xml
1.1 उ PRP <fs af='व,pn,m,sg,,,'>
1.2 लड़के NN <fs af='लड़का,n,m,sg,,,'>
</Sentence>
```
3.5.11 SL to TL Transfer

There are three modules in performing the transfer from source language to target language:

a) Transfer structure module which includes the language independent transfer engine and transfer grammar which are specific to language pairs.

b) Lexical substitution module which does the word substitution looking up an appropriate bilingual synsets or bilingual word dictionaries.

c) Transliteration module which does script level transliteration for untranslated words.

3.5.11.1 Transfer Engine Module

Normally in language processing, sentences are parsed to identify the syntactic structure of a sentence. There are more similarities than differences between Indian languages. For example Tamil and Hindi language pair does not require a full parse. In this project, MT would be performed without a full sentential parser. The structural transformation is required when the source language structure does not have an equivalent structure in the target
language. A partial parse or shallow parse is sufficient to identify the specific constituents in the sentence that has to undergo transformation.

The syntactic differences between the languages can be found in complex sentence constructions for example participles between Tamil and Hindi. This would be bridged by means of the transfer grammar.

This component has three sub-modules. Transfer Grammar, Lexical Transfer & Transliteration.

A module which can perform transliteration among Indian languages, including Urdu, needs to be developed.

Transliteration allows a word or words to be rendered in the script of the reader. For example, if a person who knows Hindi reads Bangla text in Devanagari(s) he can still understand some parts of the meaning.

It can be seen that even when translation fails for a word or a chunk, transliteration can still allow a reader to try to read and understand. Indian language share a large number of lexical items, and simply by a change in the script the reader can understand quite a few things.

**Input-Output Specification:**

**Input:** ANY_ (Source Language chunks with their feature structure)

**Output:** ANY_ (Mapped structure of Target language as chunk and their feature structure)

**Example:**

**Input:**

```xml
<Sentence id="1"> 
  1 (( NP <fs af='ladakA,n,m,pl,3,d,0_ko,' head="ladake">  
      1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'>  
      1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,,' name="ladake">  
    ))  
  2 (( NP <fs af='rAXA,n,f,sg,3,,0_kA,' head="rAXA" drel="r6:kelA">  
      2.1 rAXA NN <fs af='rAXA,n,f,sg,3,,, name="rAXA">  
    ))  
  3 (( NP <fs af='kelA,n,m,sg,3,d,,' head="kelA">  
      3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,' name="kelA">  
    ))  
  4 (( VGF <fs af='KA,v,any,any,any,,nA_pad2A+yA,' head="kAnA">  
      4.1 kAnA VM <fs af='kAnA,adj,m,sg,d,,' name="kAnA">  
    )) 
</Sentence>
```

**Output:**

```xml
<Sentence id="1"> 
  1 (( NP <fs af='ladakA,n,m,pl,3,d,0_ko,' head="ladake">  
      1.1 usa DEM <fs af='vaha,pn,any,sg,3,o,,'>  
      1.2 ladake NN <fs af='ladakA,n,m,pl,3,d,,' name="ladake">  
    ))  
  2 (( NP <fs af='rAXA,n,f,sg,3,,0_kA,' head="rAXA" drel="r6:kelA">  
      2.1 rAXA NN <fs af='rAXA,n,f,sg,3,,, name="rAXA">  
    ))  
  3 (( NP <fs af='kelA,n,m,sg,3,d,,' head="kelA">  
      3.1 kelA NN <fs af='kelA,n,m,sg,3,d,,' name="kelA">  
    ))  
</Sentence>
```
3.5.11.2 Lexical Transfer Engine

The root words identified by the morphological analyzer are looked upon in a synset dictionary for the target language equivalent. This dictionary contains the root word equivalents of the target language as well as its category (say noun, verb etc.), and other necessary information. This stage also uses dictionaries to identify the target language equivalence for the source language grammatical suffixes.

Explanation of Logic for Lexical substitution with the help of an example:
Suppose h1 is the input word in source language to be substituted. Following are the synsets:

<table>
<thead>
<tr>
<th>Source Language</th>
<th>Target Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hindi</td>
<td>Telugu</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>Hindi</th>
<th>Telugu</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>{h1,h2,h3}</td>
<td>{t1,t2,t3}</td>
</tr>
<tr>
<td>s2</td>
<td>{h4,h1,h5}</td>
<td>{t4,t3}</td>
</tr>
</tbody>
</table>

Default 1)

step 1. h1  -->  t1 (first word in S1)
            h1  --  t4 (first word in S2)

step 2. select  t1

Rationale for default in the two steps:

1. Sense selection in source language (if WSD not in place): Dominant sense of h1 is s1. Therefore select s1.

2. Target word selection (if word links are not available).

For selected sense s1, select frequent (or common) word t1 in target language.

Input-Output Specification:
Input: root, lcat, CAT_, suffix
Output: root, lcat, TKN_, suffix

Example:
Input:
<Sentence id="1"> 1     NP   <fs af='ladakA,n,m,pl,3,d,0_ko,' head="ladake">
3.5.11.3 Transliteration

Transliteration module is used for converting the word starting with "@" symbol from source language unicode character to target language in unicode form.

The fields which are converting is TKN ,lex,vib,name and head in SSF.

Here in the below example the source language is Hindi and the target language is Telugu.

**Input Output Specification:**

**Input:** Input to Transliteration will be text file in SSF format having token of source language in utf fonts.

**Output:** Output from a Transliteration will generate the token in utf fonts of target language in those places of
SSF where the starting symbol is “@”. The word with “@” symbol in SSF may occur in fields like TKN_, lexroot, vib, name and head in FS. i.e Transliteration module is used for converting the word starting with "@" symbol. The fields which are converting are TKN, lex, name and head in SSF.

**Input:** tkn_ --> {root, name, head }  
utf fonts of source language  

**Output:** tkn_ --> {root, name, head }  
utf fonts of target language

**Example:**

**Input:** Source language is Hindi  

```
<Sentence id="1">
1 (( NP  <fs af='@नयंतरण,,m,s,,0,0,' head="@नयंतरण">  
1.1 @नयंतरण NN <fs af='@नयंतरण,,m,s,,0,0,' name="@नयंतरण">  
1.2 , SYM  
))
2 (( NP  <fs af='अंकुरण,,m,s,,1,0_का,' head="अंकुरण">  
2.1 अंकुरण NN <fs af='अंकुरण,,m,s,,1,' name="अंकुरण">  
))
3 (( NP  <fs af='@स,,m,s,,0,0,' head="@स">  
3.1 @स NN <fs af='@स,,m,s,,0,0,' name="@स">  
))
4 (( VGF  <fs af='ह,,m,p,any,1,,nA' head="ह">  
4.1 ह VNN <fs af='ह,,m,p,any,1,,nA' name="ह">  
))
5 (( NP  <fs af='@ह,,f,s,,1,0_का,' head="@ह">  
5.1 @ह NN <fs af='@ह,,f,s,,1,' name="@ह">  
))
6 (( NP  <fs af='बीजोपचार,,,,,1,0_द्वारा,' head="बीजोपचार">  
6.1 @उ JJ <fs af='@उ,,any,any,,अमच्,,' head="@उ">  
6.2 बीजोपचार NN <fs af='बीजोपचार,,,,,1,' name="बीजोपचार">  
))
7 (( VGF  <fs af=',,f,s,m,,"_ज_स+त_ह,' head="र">  
7.1 र VFM <fs af=',,f,s,m,,any,,व,' name="ऍ">  
7.2 . SYM  
))
</Sentence>
```

**Output:** Target language is Telugu  

```
<Sentence id="1">
1 (( NP  <fs af='@నయంతరణ,,m,s,,0,0,' head="@నయంతరణ">  
1.1 @నయంతరణ NN <fs af='@నయంతరణ,,m,s,,0,0,' name="@నయంతరణ">  
1.2 , SYM  
))
```

Confidential ©ILMT, 1.03 Page 38 of 61
3.5.12 Put target language features

The primary task of this module is to insert the gender feature of the nouns according to the target language. There are some nouns like “paMKA” which is feminine in Telugu where as it is masculine in Hindi. So, for this we have collected list of feminine nouns in the target language(Hindi) and this module marks all the nouns which are in the list as feminine and others as masculine.

Input-Output Specification:
Input: lexp(tkn_)-->{pcat,lgen}
Output: nounP(tkn_) AND Femininelist(tkn_)  ->  lgen = "f "

Example:
Input:
<Sentence id="1"> 1  ((     NP     <fs af='paMKA,n,f,sg,3,d,'> 1.1 paMKA   NN     <fs af='paMKA,n,f,sg,3,d,'> )) 2  ((     VGF     <fs af='calav,f,sg,any,,'> 2.1 cala    VM     <fs af='calav,any,any,any,,'> 2.2 rahl    VAUX <fs af='raha,v,f,sg,any,,'> </Sentence>
2.3 hE VAUX <fs af='hE,v,any,sg,any,,,'>)
</Sentence>

Output:
<Sentence id="1">
1 (( NP <fs af='paMKA,n,m,sg,3,d,,'>)
1.1 paMKA NN <fs af='paMKA,n,m,sg,3,d,,'>)
))
2 (( VGF <fs af='cala,v,f,sg,any,,,,'>)
2.1 cala VM <fs af='cala,v,any,any,any,,,'>)
2.2 rahl VAUX <fs af='raha,v,f,sg,any,,,,'>)
2.3 hE VAUX <fs af='hE,v,any,sg,any,,,'>)
))
</Sentence>

3.5.13 Inter-chunk agreement

The task of this module is to handle the inter-chunk agreement. Inter-chunk agreements like subject-verb agreement are handled in this module. The features from subject are copied to the verb. One can write their own rules for their respective target languages.

Input-Output Specification:

Input: NPp(CAT_) → {gend,num,pers}

Output: VGp(CAT_) → {gend,num,pers}

INPUT:
<Sentence id="1">
1 (( NP <fs af='paMKA,n,m,sg,3,d,,'>)
1.1 paMKA NN <fs af='paMKA,n,m,sg,3,d,,'>)
))
2 (( VGF <fs af='cala,v,f,sg,any,,,,'>)
2.1 cala VM <fs af='cala,v,any,any,any,,,'>)
2.2 rahl VAUX <fs af='raha,v,f,sg,any,,,,'>)
2.3 hE VAUX <fs af='hE,v,any,sg,any,,,'>)
))
</Sentence>

OUTPUT:
<Sentence id="1">

3.5.14 Intra-chunk Agreement

The task of this module is to handle the intra-chunk agreement. Intra-chunk agreements like Noun-adjective agreement are handled in this module. For example, based on gender of noun, gender(attribute lgen) of adjective is set. Each language vertical would write their own rules for their target languages.

**Input-Output Specification:**

**Input:** nounp (CAT_) → {lgen}

**Output:** adjp(CAT_) → {lgen}

**Example:**

**Input:**

```
<Sentence id="1">
  1  ((          NP        <fs af='slwa,n,f,sg,3,d,,'>
   1.1     slwa    NNP      <fs af='slwa,n,f,sg,3,d,,'>
  ))
  2  ((          NP        <fs af='laDakl,n,f,sg,3,d,,'>
   2.1     acchA   JJ         <fs af='acchA,adj,f,sg,3,d,,'>
   2.2     laDakl   NN       <fs af='laDakl,n,f,sg,3,d,,'>
  ))
  3  ((          VGF       <fs af='hE,v,any,sg,3,d,,'>
   3.1     hE        VAUX     <fs af='hE,v,any,sg,3,d,,'>
  ))
</Sentence>
```

**Output:**

```
<Sentence id="1">
  1  ((          NP        <fs af='slwa,n,f,sg,3,d,,'>
   1.1     slwa    NNP      <fs af='slwa,n,f,sg,3,d,,'>
  ))
  2  ((          NP        <fs af='laDakl,n,f,sg,3,d,,'>
   2.1     acchA   JJ       <fs af='acchA,adj,f,sg,3,d,,'>
  ))
</Sentence>
```
2.2 laDakI NN <fs af='laDakI,n,f,sg,3,d,'/>
)
3 (( VGF <fs af='hE,v,any,sg,3,d,'/>
3.1 hE VAUX <fs af='hE,v,any,sg,3,d,'/>
)
</Sentence>

3.5.15 TAM Vibhakti Splitter

This module will take ‘vibhakti’ (case marker/tam) and other such information for each local word group or chunk and add function words to the chunk/local word group.

Input-Output Specification;

Input: Sequence of chunks with vibhakti and TAM attributes but without explicit function words.

TKN_,
NPp(CAT_) → cm
VGp (CAT_) → tam

Output: Same sequence of chunks with expanded/inserted function words (with right group features)

TKN_, CAT_

Example:

Input:

<Sentence id="1">
1 (( NP <fs af='rAma,n,m,sg,,o,0_ko,' head="rAma">1.1 rAma NNP <fs af='rAma,n,m,sg,,o,0,' name="rAma">
))
2 (( NP <fs af='mal,n,f,sg,,o,0_meM,' head="mal">2.1 mal NNP <fs af='mal,n,f,sg,,o,0,' name="mal">
))
3 (( NP <fs af='muMbal,n,,,0_se,' head="muMbal">3.1 muMbal NN <fs af='muMbal,n,,,0,' name="muMbal">
))
4 (( NP <fs af='xillI,n,,sg,,o,' head="DillI">4.1 xillI NNP <fs af='xillI,n,,sg,,o,' name="DillI">
))
5 (( VGF <fs af='jA,v,m,sg,any,,yA_padZa_saka+wA_hE,' head="jAnA">5.1 jA VM <fs af='jA,v,m,sg,any,,yA,' name="jAnA">
5.2 . SYM <fs af=',,punc,,,,,'/>
))
</Sentence>

Output:
3.5.16 Agreement Distribution in split vibhakti

The task of this module is to distribute the features from the vibhakti to the words into which it was split. This module is needed because of the following reason. In the analysis part, the local word grouper will compute the features for the local word group from various nodes in the local word group. Then the Local word splitter will split the words but doesn't give any features to the words. So, the main task of this module is to distribute the features of the local word group. One can write their own rules for their specific target language.

Input Output specification:

```
Input: TKN_, CAT_,

NPPp(CAT_) → {gend,num,pers}
VGPp(CAT_) → {gend,num,pers}
```

Output:
```
nounp (CAT_) → {gend,num,pers}
verbp (CAT_) → {gend,num,pers}
```
Example:
Input:
<Sentence id="1">
1  (( NP  <fs af='pariyojanA,n,f,pl,,d,,', head="pariyojan AoM">
1.1  ina  DEM  <fs af='yaha,pn,any,pl,3,o,,', name="ina">
1.2  pariyojan AoM  NN  <fs af='pariyojanA,n,f,pl,,d,,', name="pariyojan AoM">
1.3  ko  PSP  <fs af='ko,pn,,,,,'>
))
2  (( NP  <fs af='sArvajanika,adj,any,any,,,any, head="sArvajanika">
2.1  sArvajanika  JJ  <fs af='sArvajanika,adj,any,any,,,any, name="sArvajanika">
))
3  (( NP  <fs af='BAglxArl,unk,,,,,, head="BAglxArl">
3.1  nijl  JJ  <fs af='nijl,n,f,sg,,o,,'>
3.2  BAgIxArI  NN  <fs af='BAglxArl,unk,,,,,,', name="BAglxArl">
3.3  meM  PSP  <fs af='meM,pn,,,,,'>
))
4  (( JJP  <fs af='sWApiwa,n,m,sg,,o,,', head="sWApiwa">
4.1  sWApiwa  JJ  <fs af='sWApiwa,n,m,sg,,o,,', name="sWApiwa">
))
5  (( VGF  <fs af='kara,v,m,sg,any,,,yA', head="kiyA">
5.1  kiyA  VM  <fs af='kara,v,m,sg,any,,,yA', name="kiyA">
5.2  jA  VAUX  <fs af='jA,v,m,,,,,yA', name="kiyA">
))
</Sentence>
OUTPUT:
<Sentence id="1">
1  (( NP  <fs af='pariyojanA,n,f,pl,,d,,', head="pariyojan AoM">
1.1  ina  DEM  <fs af='yaha,pn,any,pl,3,o,,', name="ina">
1.2  pariyojan AoM  NN  <fs af='pariyojanA,n,f,pl,,d,,', name="pariyojan AoM">
1.3  ko  PSP  <fs af='ko,pn,,,,,'>
))
2  (( NP  <fs af='sArvajanika,adj,any,any,,,any, head="sArvajanika">
2.1  sArvajanika  JJ  <fs af='sArvajanika,adj,any,any,,,any, name="sArvajanika">
))
3  (( NP  <fs af='BAglxArl,unk,,,,,, head="BAglxArl">
3.1  nijl  JJ  <fs af='nijl,n,f,sg,,o,,'>
3.2  BAgIxArI  NN  <fs af='BAglxArl,unk,,,,,,', name="BAglxArl">
3.3  meM  PSP  <fs af='meM,pn,,,,,'>
))
4  (( JJP  <fs af='sWApiwa,n,m,sg,,o,,', head="sWApiwa">
4.1  sWApiwa  JJ  <fs af='sWApiwa,n,m,sg,,o,,', name="sWApiwa">
))
5  (( VGF  <fs af='kara,v,m,sg,any,,,yA', head="kiyA">
5.1  kiyA  VM  <fs af='kara,v,m,sg,any,,,yA', name="kiyA">
5.2  jA  VAUX  <fs af='jA,v,m,,,,,yA', name="kiyA">
))
5.3 hE VAUX <fs af='hE,v,,sg,any,,,hE'>
</Sentence>

3.5.17 Assign Default Features

The task of this module is to assign the default features to the words which don't have features. There are some cases where the morph doesn't give all the features and also the local word splitter doesn't all the features for the function words it added. So, this module gives default features. This is needed so that the word generator (after this module) is able to generate the word. One can write their own rules for their specific target language.

Input Output specification:

Input:  TKN_ -> {lgen = "   ",lnum="   ",lper="   "}
Output: TKN_ -> {m,sg,3}

Example:

Input:

<Sentence id="1">
1    ((  NP  <fs af='yaha,pn,any,pl,3,o,,' head="ina">  
1.1  ina  DEM  <fs af='yaha,pn,any,pl,3,o,,' name="ina">  
))
2    ((  NP  <fs af='pariyojanA,n,f,pl,,d,,' head="pariyojanAoM">  
2.1 pariyojanAoM  NN  <fs af='pariyojanA,n,f,pl,,d,,' name="pariyojanAoM">  
2.2   ko  PSP  <fs af='ko,pn,,,,,'>
))
3    ((  NP  <fs af='sArvajanika,adj,any,any,,,any,' head="sArvajanika">  
3.1 sArvajanika  JJ  <fs af='sArvajanika,adj,any,any,,,any,' name="sArvajanika">  
))
4    ((  NP  <fs af='BAgLxArI,unk,,,,,' head="BAgLxArI">  
4.1  nijl  JJ  <fs af='nijl,n,f,sg,,o,,'>
4.2  BAgIxArI  NN  <fs af='BAgLxArI,unk,,,,,' name="BAgLxArI">  
4.3  meM  PSP  <fs af='meM,pn,,,,,'>
))
5    ((  JJP  <fs af='sWApiwa,n,m,sg,,o,,' head="sWApiwa">  
5.1 sWApiwa  JJ  <fs af='sWApiwa,n,m,sg,,o,,' name="sWApiwa">  
))
6    ((  VGF  <fs af='kara,v,m,sg,any,,,yA' head="kiyA">  
6.1  kiyA  VM  <fs af='kara,v,m,sg,any,,,yA' name="kiyA">  
6.2  jA  VAUX  <fs af='jA,v,m,,,,,yA'>  
6.3  hE  VAUX  <fs af='hE,v,,sg,any,,,hE'>  
))
</Sentence>
3.5.18  Word Generation

In order to generate a word form for a specific grammatical category the corresponding suffixes have to be combined with the root word as per morphology of the language. The morphological generator takes its input from the transfer grammar component. The input would be the root word along with its grammatical features. The generator then inflects the root word according to the morphology of the language and outputs the target language word form. The words thus generated are produced one after another to form the complete target language sentence.

Example
Input Output Specifications
Input : other_ -_-->{root,lcat,lgen,num,per,tam}
Output : tkrn_
Input:
<Sentence id="1">
1 (( NP
1.1 piCalle NN <fs af='piCallA,unk,,,,,'/>
1.2 sAla NN <fs af='sAla,n,m,sg,,o,,' conj="blank" spec="blank" dubi="blank">
 ))
2 (( JJP
2.1 pawA JJ <fs af='pawA,n,m,sg,,o,,' conj="blank" spec="blank" dubi="blank">
 ))
3 (( VGF
3.1 nahlM NEG <fs af='nahlM,adv,,,,,'/>
 ))
4 (( NP
4.1 kOna WQ <fs af='kOna,pn,any,sg,1,d,,'/>
4.2 sI RP <fs af='sI,v,any,any,any,,0,0'>
 ))
5 (( NP
5.1 blmArI PRP <fs af='blmArI,n,f,sg,,o,,' conj="blank" spec="blank" dubi="blank">
 ))
5.2 laga VAUX <fs af='laga,v,f,sg,any,,yA,yA'>
 ))
6 (( CCP
6.1 Ora CC <fs af='Ora,avy,,,,,'/>
 ))
7 (( NP
7.1 vaha PRP <fs af='vaha,pn,any,sg,1,d,,'/>
 ))
8 (( VGF
8.1 cala VM <fs af='cala,v,any,any,any,,0,0'>
 ))
9 (( NP
9.1 basa NN <fs af='basa,v,f,sg,any,,yA,yA'>
9.2 . SYM <fs af='.,punc,,,,,'/>
 ))
</Sentence>

Output:
<Sentence id="1">
1 (( NP
1.1 piCalle NN <fs af='piCallA,unk,,,,,'/>
1.2 sAla NN <fs af='sAla,n,m,sg,,o,,' conj="blank" spec="blank" dubi="blank">
 ))
2 (( JJP
2.1 pawA JJ <fs af='pawA,n,m,sg,,o,,' conj="blank" spec="blank" dubi="blank">
 ))
3 (( VGF
3.6 Evaluation

Evaluation mechanism would be setup for end users for individual sentences as well as text. This should allow subjective evaluation, for comprehensibility and quality of the translation. In the second stage, evaluation would be done on speed & usability of the system.

Specification for the Alpha Version

A. Performance

- Word coverage ~95% (not unique)
  a) 75% WSD
- All functional words should be covered
- Syntactic Structures
  i) All single clauses.
  ii) Handle upto 2 dependent clauses. For example, relative clauses such as “The cat that ate the mouse that drank the milk
  iii) All IPs (modals and Inf)
(iv) All CPs

- Comprehensible (may not be grammatical) – Criteria for comprehensibility
  - 85% of the sentences should be at the level of 3, 4 and 5 on a five point scale (1-5)
  - 15% on level 1, 2

**B. Regression Testing**

- **Comprehensive output**

The sample of sentences will be 1000 from tourism domain with following distribution

- 200 Common
- 700 'Normal' (Neither common nor uncommon)
- 100 Unusual
4

System Integration and Testing

The modules described earlier will be integrated and tested at a single location using tools built for the purpose such as dashboard.

4.1 Dashboard

The Dashboard is a configuration tool for setting up a system based on blackboard architecture. It sets up the control flow of a blackboard based system besides providing many other facilities.

It assumes that the system which it has to configure, consists of modules which operate on an in-memory data structure. The data structure is usually so designed that the modules do processing based on the information present in it, and after that leave their output in the same data structure. The data structure is based on tree structure, and associated feature structures. One of the advantages of such an approach is that the system can be made robust which continues to function even after an individual module has failed.

The in-memory structure should preferably have a text notation for representing it unambiguously. This will be called as the in-stream version or in-text version. The notation helps in readability, and as we will see, making the in-memory structure usable across different programming languages, and in sending it across different machines, and processes. Together with the notation, one would need a "reader" and a "printer" which converts from text notation to in-memory data structure, and vice versa.

All the above conditions are satisfied for the implementation of ILMT system. SSF is the in-stream notation, as well as in-memory structure.

For the dashboard to setup the system, a specification file is needed, which specifies the dependencies between modules. If a module y needs the resulting data from module x (usually, immediately prior to its execution), then the module y depends on x. When such a dependency is specified, the dashboard would set the system so that y is called after x.

In one of the simplest possible ways, if several modules function in the pipe-line fashion, they can be listed one after another in the specification file, showing sequential dependencies. If the pipeline splits in two, and remerges later, it can also be handled. No loops are, however, permitted.

If a module takes more than one input, one coming from one of the module and the others as files, it can be so specified. The dashboard configures the system appropriately.

In case, a module x takes its inputs from more than one module as well as files, the output of the relevant modules are saved in files (in textual notation). The dashboard sets up these files as arguments of module x,
appropriately.

**Functionality**
The dashboard can be setup to show/store the output of each module. This helps in debugging the modules. (Debugging difficulty goes up when the system is designed in a robust way, because the system does not stay silent even in the face of failure, it simply operates in a degraded mode. This causes many errors to be missed unless proper care is taken in testing.)
The dashboard can run programs in different programming languages as part of the same larger program. This is made possible by the in-stream capability. Modules written in different programming languages can operate on their in-memory structures created using the APIs written in their respective languages, however whenever the information is to be used by a module in another programming language, the printer-reader combination can convert and transfer it from one in-memory structure to another, using in-stream conversion.
The dashboard also does validation using reference data provided by the developers of the system. It can either run the entire system, or run individual modules in stand-alone mode, depending on the availability of reference data. Thus, it becomes an invaluable aid in debugging a robust system.
It can also be setup to do profiling, and finding the time taken by each of the modules or selected modules.
In addition to functional modules, the system must have system administration modules to control and manage user access and administration of the ILMT system, and also its configuration, and system operations and maintenance tasks. Dashboard provides many facilities to support these tasks.

4.2 **Graphical User Interface (GUI)**
The GUI of ILMT will provide visualization aid at multiple levels of abstractions.
  - At module interaction level
  - At process interaction level (as a module can have multiple processes)
  - At temporal level showing time consumed by various modules in the system.
  - At post-processing level with visualization of data exchange among processes

In addition, the GUI will also be equipped with provisions for setting up the test environment and control the running of the system under observation. And hence, it must provide following primitives as well:
  - Set the system configuration
  - Set / change module configuration
  - Set the test environment (Source of Input and Recording of Output)
  - Setting of breakpoints
  - Visualize the current system state
  - Visualize the data exchange across modules
  - Stop / restart of certain module / process
  - Shutdown the system
References:

   {http://ltrc.iiit.net/downloads/nlpbook/nlp-panini.pdf}


   {http://192.168.36.181/ILMT/tasks/h7a/cdacn%23Nov-2006-29-11-48-01%23corporastd_ver2.pdf}


   {http://192.168.36.181/ILMT/M4/summary.doc}

   {http://192.168.36.181/ILMT/???

8. POS tag Standards: Available at ILMT site  
   {link at: Tasks & Downloads, H1a. POS Tagger & Chunking engine,  
   On Site: http://ltrc.iiit.ac.in/ILMT/login.php}

9. Dictionary Standards: Available at ILMT site  
   {Tasks & Downloads, H6a. Dictionary Standards, On Site:  
   http://ltrc.iiit.ac.in/ILMT/login.php}
10. Sinha Mukul, Software Engg process of ILMT 2007,
{ At link :ILMT Portal--> Project Documents --> A Software Integration
Appendix A: XML File Structure for Corpora Creation (.cml files)

<document> - Root Element
Attributes: docid, docnumber

<author>
Attributes: lastname, firstname, middlename

<availability>
Attribute: format  Electronic/Physical

<bibl>

<bytecount>
Size of file in KB

<domain>
Health/Tourism/General

<creation>
Attributes: date, institutename, creatorname: lastname, middlename, firstname

<distributor>

<edition>
Number

<encodingdesc>
Attributes: originalencoding, newencoding

<sentencemarker>

<language>
Attributes: Name kn/ma/mr/hi/bn/pa/te/ta/ur, writingsystem (LTR,RTL), script

<normalization>
Attributes: normalized Yes/No, utilityname:

<projectdesc>
IL-IL consortia

<pubaddress>
Web Address/Physical Address

<pubdate>
Date of Publication

<publicationstmt>
copyrighted/non copyrighted

<publisher>
Name/URL

<pubplace>

<translation>
Attributes: availability(Yes/No), translations (Name of xml file which contains the translation), translator
(Translator Name(Fname, lname, mname),Institute name,language,writing system,script

<wordcount>
(using wc of Linux/Unix)

<body>

<p> Paragraph
Attributes like “paranumber”

<segment> Sentences in the paragraph
Attributes like “segmentnum”

<sentence> Attribute: Sentencenumber

<foreign>
Attributes: Language, writingsystem

<Note>
Appendix B: Shakti Standard Format: BNF with Brief Description

A. SSF-BNF

Sentence Level SSF

\[
\text{sen_ssf} ::= \text{snt_hdr} (\text{row})* \text{snt_footer}
\]

\[
\text{snt_hdr} ::= \langle\text{Sentence id} = \text{\[0-9]+ }\rangle\text{ >}
\]

\[
\text{snt_footer} ::= \langle/\text{Sentence}\rangle\text{ >}
\]

I) First way to define $row$

\[
\text{row} ::= ^\text{$addr_ (t) (stk_ (t) (cat_ (t (alt_fs)? } \n\| (t ') )\rangle'}
\]

\[
\text{addr_} ::= [0-9]+ ([0-9]+)*
\]

\[
\text{stk_} ::= \text{$ANSU \| 'l'}
\]

\[
\text{cat_} ::= \text{$AN}
\]

\[
\text{alt_fs} ::= \text{$fs ( \| $fs )*}
\]

\[
\text{fs} ::= \langle\text{fs} (\text{key = \text{\[alt_val\]+ }\rangle\text{ >}
\]

\[
\text{key} ::= \text{$AN}
\]

\[
\text{alt_val} ::= \text{$val ( \| $val )*}
\]

Constraint:
1) if $\text{key} = 'af'$
   then $\text{VAL}$ is $\text{ANDU_Com}$

2) if $\text{Key} = 'drel'$
   then $\text{Val}$ is $\text{ANDU_COLN}$

3) else $\text{VAL}$ is $\text{ANDU}$
Some definition:

$\text{VAL} ::= \text{ANDU} \mid \text{ANDU\_Com} \mid \text{ANDU\_COLN} \mid \text{SAN}

$\text{SAN} ::= [a-zA-Z 0-9]+ \quad //\text{Alpha Numeric}

$\text{ANSU} ::= \text{SAN\ (SAN)}^* \quad //\text{Alpha Numeric Single Underscore}

$\text{ANDU} ::= \text{ANSU\ (ANSU)}^* \quad //\text{Alpha Numeric Double Underscore}

$\text{ANDU\_Com} ::= \text{ANDU\ (',', \text{ANDU})}\{7\} //\text{Alpha Numeric Underscore Comma}

$\text{ANDU\_COLN} ::= \text{ANDU\ :-\ ANSU} \quad //\text{Alpha Numeric Underscore colon}

B. Description of the SSF

B1: The SSF representation consists of a sequence of trees. Each tree is made up of nodes.

B2: A node has three "system" properties:
   (a) address - referred to by property name ADDR_
       (not stored explicitly, however can be computed).
   (b) token - accessed by property name TKN_
   (c) category - accessed by property name CAT_

and others which are defined by the developers:
   (d) others - accessed through their feature names

B3: A property has a prop-name and prop-val as follows:

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property value format</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR_</td>
<td>:= &lt;number&gt; (.&lt;number&gt;)^*</td>
</tr>
<tr>
<td>TKN_</td>
<td>:= '(' [\ AN PUNC*</td>
</tr>
<tr>
<td>CAT_</td>
<td>:= pcat(CAT_) -&gt; {NP, VG ...};</td>
</tr>
<tr>
<td></td>
<td>lcat(CAT_) -&gt; {NN, JJ, ...}</td>
</tr>
</tbody>
</table>

where pcat stands for all the phrasal categories, and lcat for all the lexical categories.

Example: Given below are two trees with a single node each, having their tokens respectively as 'children' and 'played', and
their categories as NN and VB:

<table>
<thead>
<tr>
<th>ADDR_</th>
<th>TKN_</th>
<th>CAT_</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>children</td>
<td>NN</td>
</tr>
<tr>
<td>2</td>
<td>played</td>
<td>VB</td>
</tr>
</tbody>
</table>

The addresses are 1 and 2.

B4: A node may have one or more features. A feature consists of attribute-value pair.

Example: Two nodes with attributes 'root':

<table>
<thead>
<tr>
<th>ADDR_</th>
<th>TKN_</th>
<th>CAT_</th>
<th>OTHER_</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>children</td>
<td>NN</td>
<td>&lt;fs root=child num=pl&gt;</td>
</tr>
<tr>
<td>2</td>
<td>played</td>
<td>VB</td>
<td>&lt;fs root=play tense=past&gt;</td>
</tr>
</tbody>
</table>

Note that the feature 'root' has values 'child' and 'play' for the respective nodes. Similarly, features 'num' (number) and 'tense' are also shown.

B5: Attributes and Values
First some definitions:
(a) Alpha-numeric $AN = [a-zA-Z0-9]+$  
   (In standard Unix notation for regular expressions, square brackets enclose any of the possible character, and the hyphen ('‐') shows the range. Thus, 'a-z' means any lower case alphabetic character, etc.)
(b) Alpha-numeric-underline $ANU = $AN[a-zA-Z0-9]$_*$  
   (Start with alphanumeric ($AN) followed by any of alpha-numeric and underline)
(c) Alpha-numeric-underline without two underlines in a sequence (start with alpha-numeric)  
   $ANSU = $AN(_$AN)*  
(d) Alpha-numeric-underline with two possible double underlines  
   $ANDU = $AN(_$AN|__$AN)*

Now,
(i) An attribute is defined by $ANSU
(ii) A value is defined by $ANDU
(iii) A simple value is defined by $ANSU

Observe from the above that:
(a) A double underline (in a sequence) should not occur in an attribute or a simple value.
(b) A double underline (in a sequence) is used to define a structured or successively refined value.

B6: A convention that is used is that a user defined attribute should not have a underline at the end. System attributes may have underlines at their end.

B7: Values are in a hierarchically of progressively more refined values, or detailed definition. For example, if a value is: vmod__varg__k1
It shows the value as 'vmod' (modifier of a verb), which is further refined as 'varg' (argument of the verb) of type 'k1'.

B8: A value B covers another value C, if C is more refined or detailed. In other words, C is of type B, and is more specified, or detailed than B.
Thus, if a constraint says that the value must be of type B, then C also satisfies the constraint.

For example, a value B covers value C in the following:

\[ B = \text{vmod__varg} \]
\[ C = \text{vmod__varg__k1} \]

B says that something is an argument (vmod__varg), and C says that it is an argument of type k1.

This indicates that C is a further refined/detailed form of B.

If a constraint says that a value should be of type 'vmod__varg' then clearly both B and C satisfy the constraint.

A value B is covered by a value C if the following holds:

- case (i): \( B = C \)
- case (ii): B is a prefix of C, followed by two underlines and $ANDU$.

B9: Nodes might be interlinked with each other through directed edges. Usually, these edges have nothing to do with phrase structure tree, and are concerned with dependency structure, thematic structure, etc. These are specified using the feature structure syntax, however, they do not specify a value for an attribute.

For example, if a node is karta karaka of a node named 'play1' in the dependency structure, in other words, if there is a directed edge from the latter to the former, it can be shown as follows:

\[
\begin{array}{c|c|c|c}
\text{ADDR} \_ & \text{TKN} \_ & \text{CAT} \_ & \text{OTHER} \\
1 & \text{children} & \text{NN} & <\text{fs drel=k1:play1}> \\
2 & \text{played} & \text{VB} & <\text{fs name=play1}> \\
\end{array}
\]

The above says that there is an edge labelled with 'k1' from 'played' to 'children' in the 'drel' tree (dependency relation tree). The node with token 'played' is named as 'play1' using a feature (with a special attribute called 'name').

So the syntax is as follows: if you associate a feature with a node C as follows:

\[ \text{TREENAME} = \text{EDGELABEL:.NODENAME} \]

It means that there is an edge from NODENAME to C, and the edge is labeled with EDGELABEL. Name of a node may be declared with the feature 'name':

\[ \text{name} = \text{NODENAME} \]

(All the words in all capitals may be substituted with appropriate user-defined names.)
## Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Description</th>
<th>Author /Contributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/10/2008</td>
<td>V1.03</td>
<td>Approved for release</td>
<td>Dr. Rajeev Sangal</td>
</tr>
<tr>
<td>20/09/2008</td>
<td>V1.03</td>
<td>1) Revised on the basis of 19-21 August workshop in DIT</td>
<td>Pawan Kumar, Rohit Gupta</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) Redefine the following sub modules:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a)Pruning Morph Pruning Guess</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Morph Pick one Morph</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) Local Word Grouper/Splitter</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c) SL to TL Transfer as Transfer Engine Module, Lexical transfer Engine, Transliteration</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>d) Generator as Put target language features Agreement Feature, Inter-chunk Agreement, Intra-chunk Agreement, TAM Vibhakti Splitter, Agreement Distribution in split vibhakti, Assign, Default Features, Word Generation</td>
<td></td>
</tr>
<tr>
<td>20/07/2008</td>
<td>V1.022</td>
<td>Modified on feedback</td>
<td>Rohit Gupta</td>
</tr>
<tr>
<td>10/02/2008</td>
<td>V1.021</td>
<td>Modified on feedback</td>
<td>Rohit Gupta</td>
</tr>
<tr>
<td>05/01/2008</td>
<td>V1.02</td>
<td>Release to customer</td>
<td>Dr. Mukul Sinha</td>
</tr>
<tr>
<td>10/09/2007</td>
<td>V1.01</td>
<td>Suggestion by customer</td>
<td>DIT</td>
</tr>
<tr>
<td>15/02/2007</td>
<td>V1.01</td>
<td>Modified by Reviewer</td>
<td>Dr. Mukul Sinha</td>
</tr>
<tr>
<td>01/2/2007</td>
<td>V1.00</td>
<td>Approved for release</td>
<td>Dr. Rajeev Sangal</td>
</tr>
<tr>
<td>20/01/2007</td>
<td>V0.9</td>
<td>Draft Document</td>
<td>Rohit Gupta</td>
</tr>
</tbody>
</table>