Report on Nepali Computational Grammar

Prajwal Rupakheti, Laxmi Prasad Khatiwada
Bal Krishna Bal

Madan Puraskar Pustakalaya
Lalitpur, PatanDhoka,Nepal
{ prajwalrupakheti@gmail.com, lkhatiwada@gmail.com, bal@mpp.org.np }

Abstract
This document reports the research and development of a Nepali Computational Grammar (NCG) that essentially involves the development of the intermediate modules like the Parts-of-Speech(POS) Tagger, chunker and the parser. Besides, discussing on the architecture of the system, we also report the general work performance and coverage of the individual modules and the overall NCG system as a whole.

Introduction

The NCG work is an attempt to develop a basic computational framework for analyzing the correctness of a given input sentence in the Nepali language. While the primary objective remains in building such a framework, the secondary objective remains in developing intermediate standalone Natural Language Processing (NLP) modules like the tokenizer, morphological analyzer, stemmer, POS Tagger, chunker and the parser. These standalone modules may be used for any other NLP applications besides the NCG. Talking about the individual modules, we have used the TnT[1], a very efficient and the state-of-the-art statistical POS tagger and trained it with around 82000 Nepali words. The chunker module involves a hand-crafted linguistic chunk rules and a simple algorithm to process these rules. As far as the parser module is concerned, we have implemented a constraint-based parser following the dependency grammar formalism [3] and in particular the Paninian Grammar framework [1, 2, 4, 5]. Our parser module uses the linguistic resource in a form of a karaka frame consisting of about 900 Nepali verbs. The TnT POS Tagger currently tags known words and unknown words with accuracy rates of 97% and 56% respectively. The chunker module has the coverage of 50-60%. With some additional chunk rules and some refinements in the existing rules, the coverage is expected to grow much higher. The parser module provides a correct parse and a corresponding analysis given that the chunker module provides a correct chunk to the input sentence. A detailed discussion on the technical aspects of each module would follow in the next sections.

System architecture

The system architecture of the NCG is presented in Fig.1 below. As can be seen from the diagram, the NCG presents itself as a pipeline architecture whereby the output of a particular module serves as the input for the other. We discuss about the implementation aspects of each module below.

[1] http://www.coli.uni-saarland.de/~thorsten/tnt/

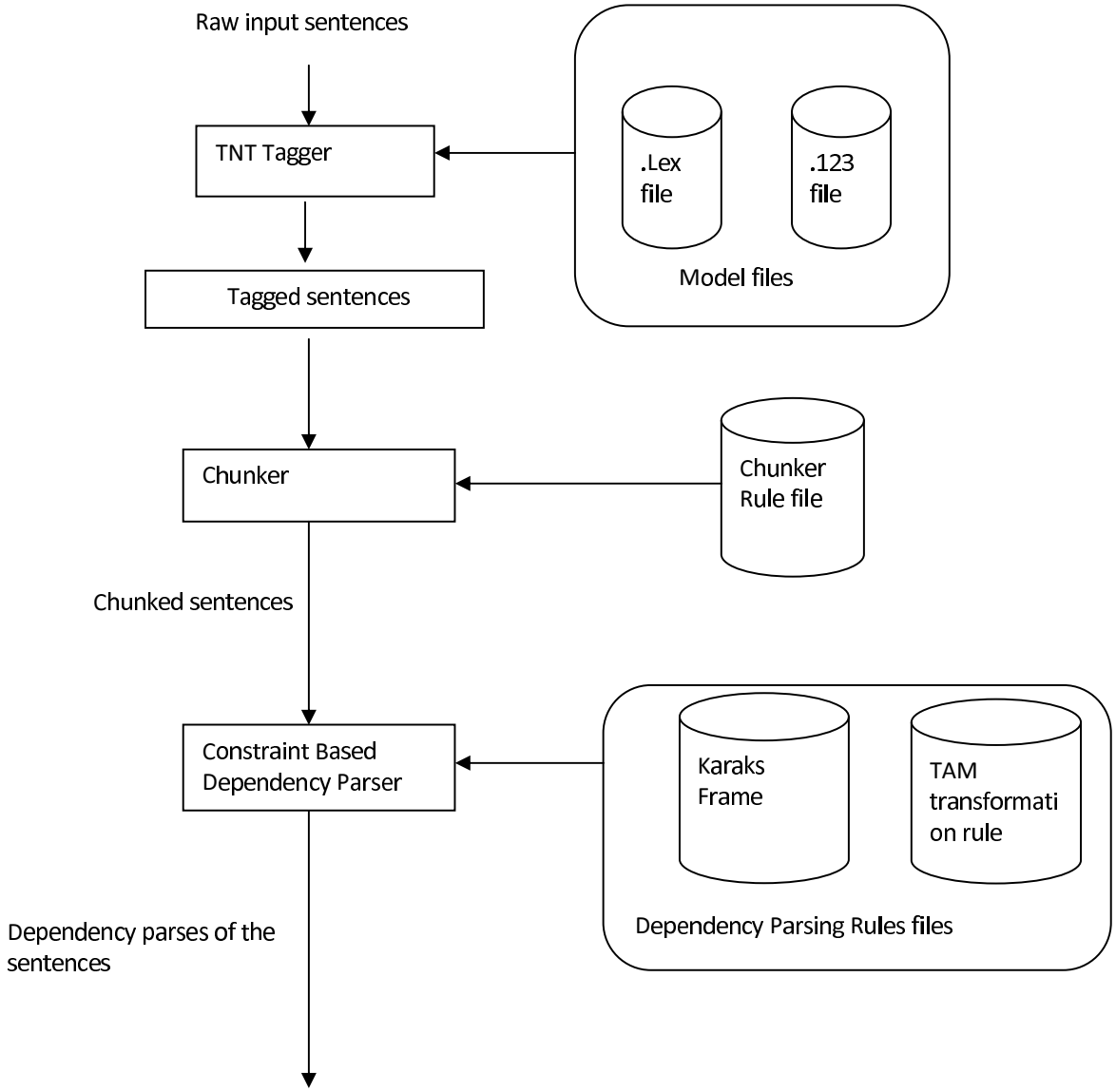


Fig. 1: Architecture of the Nepali Computational Grammar

Parts of Speech Tagging and the TNT POS Tagger

Part of Speech Tagging is the process of assigning a part-of-speech or lexical class marker to each word in a corpus. Most recent researches with trainable Part of Speech Taggers have

explored the Hidden Markov Model[2] (HMM) based stochastic tagging. HMM based stochastic tagging involves choosing the tag sequence which maximizes the product of word likelihood and tag sequence probability. In this case, we have used the trigram approach for tagging the input words. For our research purpose, we have compiled a corpus containing approximately 88000 Nepali words. The corpus has been compiled from various sources like newspapers, books etc.

POS Tagset

The POS tag set that we have developed has 42 POS tags. The development of the tag set has been largely influenced by the PENN Treebank tag set. Below in Table 1, we present the list of POS tags for Nepali.

Table 1. List of POS Tags for Nepali

Category	**Remarks**	**POS Tag ID No.**	**POS Name**	**POS Tag**
Noun		1	Common Noun	NN
		2	Proper Noun	NNP
Pronoun		3	Personal Pronoun	PP
		4	Possessive Pronoun	PP$
		5	Reflexive Pronoun	PPR
		6	Marked Demonstrative	DM
		7	Unmarked Demonstrative	DUM
Verb		8	Finite verb	VBF
		9	Auxiliary verb	VBX
		10	Verb infinitive	VBI
		11	Prospective participle	VBNE
		12	Aspectual participle	VBKO
		13	Other participle verb	VBO
Adjective		14	Normal/unmarked	JJ
		15	Marked Adjective	JJM
		16	Degree Adjective	JJD
Adverb		17	Manner Adverb	RBM
		18	Other Adverb	RBO
Intensifier		19	Intensifier	INTF
Postpositions		20	Le-Postposition	PLE
		21	Lai- Postposition	PLAI

[2] http://en.wikipedia.org/wiki/Hidden_Markov_model

		22	Ko-Postposition	PKO
		23	Other Postpositions	POP
Conjunction		24	Coordinating	CC
		25	Subordinating conjunction	CS
Interjection		26	Interjection	UH
Number		27	Cardinal Number	CD
		28	Ordinal Number	OD
Plural marker		29	Plural marker हरू	HRU
Question word		30	Question word	QW
Classifier		31	Classifier	CL
Particle		32	Particle	RP
Determiner		33	Determiner	DT
Unknown word		34	Unknown word	UNW
Foreign word		35	Foreign word	FW
Punctuation		36	sentence Final	YF
		37	sentence Medieval	YM
		38	Quotation	YQ
		39	Brackets	YB
Abbreviation		40	Abbreviation	FB
Header List		41	Header List	ALPH
Symbol		42	Symbol	SYM
Null		43	Null	<Null>

TnT POS Tagger

TnT (Trigrams'n'Tags) is a very efficient statistical part-of-speech tagger that is trainable on different languages and virtually any tag set. The component for parameter generation gets trained on the POS tagged corpora. The system incorporates several methods of smoothing and of handling unknown words. TnT is not incorporated for a particular language. Instead, it is optimized for training on a large variety of corpora. This tool is suitable for tagging any language which uses white spaces to separate words, like Nepali, Hindi, English, and French. In Nepali language too, words are separated by white spaces, which makes TnT the best tool for the tagging of Nepali Language. Besides, the permission to use, copy and modify this software and its documentation is granted to non-commercial entities for free. This is also an important reason behind choosing this tool.

TnT Tagging architecture

TnT uses the second order Markov models for part-of speech tagging. The states of the model represent tags, outputs represent the words. Transition probabilities depend on the states, thus pairs of tags. Output probabilities only depend on the most recent category. To be explicit, we calculate

$$i=1n\text{argmax}Ptiti-1,ti-2P(wi|ti)$$

……..(i)

Here, we try to maximize the product of the probability of a tag pattern (trigrams in this case) and the probability of a word getting a particular tag.

TnT and the Nepali corpus

We have adapted TNT tagger for our tagging purpose. As the accuracy of TNT is 96% for languages whose words are separated by white spaces, all improvement that we need to do in our case is to simply increase the size of our corpus. So far we have a corpus of about 80,000 manually tagged words.

We have checked the accuracy of TNT with raw Nepali sentences from various domains. The accuracy so far is 56 % for unknown words and 97 % for known words. We provide below a sample of the POS Tagged Nepali text by the TnT Tagger.

राम/NNP ले/PLE हार/NN लाई/PLAI दियो/VBF

Chunking

A chunk can be defined as a collection of contiguous words such that the words inside a chunk have dependency relations among them, but only the head of the chunk has the dependency relation with the outside chunk. Hence a chunker is a tool to identify such chunks in the given sentences. For the chunker, we have defined a set of linguistic rules for chunking Nepali phrases. Besides, we have devised a simple algorithm to find the chunk in the Nepali sentences on the basis of the chunk rules. The algorithm has been already implemented in Java and provided below.

Chunking algorithm – A rule based algorithm

The following algorithm is applied for identifying chunks in a sentence.

Let us say we have following *n* rules for defining *n* chunks:

Pattern 1-> chunk 1

Pattern 2- >chunk 2

…

…

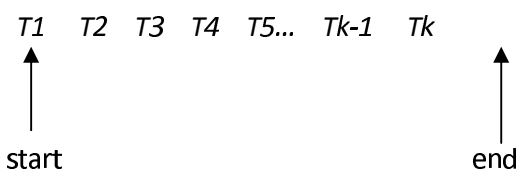
…

Pattern n ->chunk n

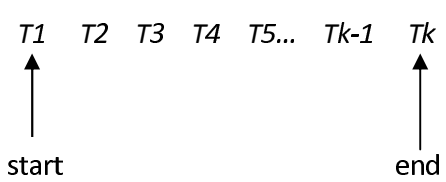
Now if a given sentences has a following pattern of POS tags of its respective lexemes:

T1 T2 T3 T4 T5…Tk

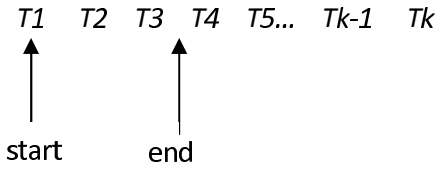
Then we proceed the chunking as follow:



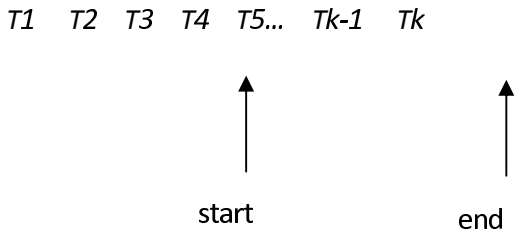
Initially we mark the first token with the *start pointer* and the last token as the *end pointer*. Now if the pattern in between the *start* and *end* inclusive (i.e. *T1 T2 T3 T4 T5…Tk)* is there in the chunk rule than we will tag the whole pattern as a single chunk. If it is not there then we will decrease the *end pointer* to one token left and continue doing this until and unless the pattern between *start* and *end* is there in the chunker rule.



Similarly by continuing the same process if the *end pointer* reached the T3 token



and the pattern between *start* and *end* (i.e. T1 T2 T3) is there in the chunker rule then we chunk the tokens between *start* and *end* as a single chunk and shift the *start pointer* to one token right of *end pointer* and *end pointer* to the rightmost end token as follow



We will continue doing this until the *end pointer* reaches the last token i.e. *Tk*.

Chunk rules and chunked output

Below, we provide a sample of the Nepali chunking rules. Currently, we have 13 chunk rules as listed below and the coverage of the rules in terms of chunking is around 50-60%. With the addition of more rules and the refinement of the existing ones, we expect the coverage to grow higher. Below, we provide a sample of the chunk rules:

```
NCH:(DUM)(PLAI)
NCH:(NN)(PLAI)
NCH:(NN)
NCH:(NNP)
NCH:(DUM)(PLE)?
NCH:(NN)(POP)
NCH: ((NN)|(NNP))((PLE)|(PLAI)|(PKO)|(POP))?
```

```
NCH:((PPR)|(PP$))((NN)|(NNP))(PKO)?
NCH:((NNP)|(NN)((PKO)|(PLE))?((NN)|(NNP))((POP)?))
NCH:(DET)?((NUM)?(CL)?)*(INT)?(ADJ)*((NN)|(NP)|(PP))((PLE)|(PLAI)|(VNE)|(VKO)
)((NN)|(NP)|(POP))
VCH:(VBF)|(VOP)|(VKO)|((VI)(VF))|(VAUX)
FVCH:(VOP)|(VKO)|((VI)(VF))|(VAUX)
NVCH:(VOP)|((VOP)?)|((VOP)(VKO))
GVCH:(VI)|(VNE)
AJCH:(INT)?(ADJ)
AVCH:(ADV)?(INT)?((ADV)|((VOP)?))
AVMCH:(NN)?((POP)|((VI)(POP))|((VKO)(POP)))
CNCH:(CC)
CLCH:(CL)(POP)?
PNCH:(PUN)
NCH:((INT)*(ADJ)?(NN)((POP)(INT|ADJ)*(NN))?)(((YM)(INT|ADJ)*(NN)((POP)(INT|AD
J)*(NN))?)*(CC)(INT|ADJ)*(NN)((POP)(INT|ADJ)*(NN))?)*(PKO)?
NCH:((CD)(NN)(PLE)|(JJ)(NN)(CC)(NN))|((JJ)(NN)(NNP){2}+(PLE))
NCH:((JJ)(NN)((NNP)(YM))*(NN))|((NN)(CD)(POP))|((NN)(NNP))|((NN)(YM)(JJ)(CC)(
JJ)(JJ)(NN))|((NN)(YM)(NN)(CC)(NN))
NCH:((NNP)(NN){2}+)|((PPR)(JJ)(NN)(PLAI))|((RBM)(NN)(POP))
NCH:((DUM)*((DM)|(CC)|(DM))*(DUM)(NN))|((DUM)|(DN)(NN)|(NNP))
NCH:((DM)*((DUM)(JJ))*((JJD)|(NN))*(NNP))
NCH:((DM)(POP))
NCH:((PP)|(PR$)|(DUM)|(DM))
NCH:(((JJ)+|(NNP))*(NN)*(NNP)*((PLE)|(PKO))*)
NCH:(((PPR)|(PP$))((NN)|(NNP))|((PKO)(PLAI)(PLE)(POP))?)
```

The extended meaning of the abbreviated chunk notations are given below:

NCH- Noun chunk

VCH – Verb chunk

FVCH – Finite verb chunk

NVCH – Non-finite verb chunk

GVCH – Gerund verb chunk

AJCH – Adjective chunk

AVCH – Adverb chunk

ADVCH – Adverbial modifier chunk

CNCH – Conjunction chunk

CLCH – Classifier chunk

PNCH – Punctuation chunk

Parsing

As noted earlier in the document, the Nepali parsing module follows the dependency grammar formalism. In the dependency based parsing, treat a sentence as set of modifier-modified

relations. A sentence has a primary modifier or the root (which is generally a verb). Dependency parser gives us the frame work to identify these relations. Relations between noun constituent and verb are called *karakas. Karakas* are syntactico-semantic in nature. Syntactic cues help us in identifying *karakas.*

Basic karaka relations

The Paninian grammar framework defines six types of karaka relations as listed below:

- Karta – agent/doer/force (k1)
- Karma – object/patient(k2)
- Karana instrument(k3)
- Sampradaan-beneficiary(k4)
- Apaadan-sources(k5)
- Adhikarana-location in place/time/other (kx)

Karaka frame

It specifies what karakas are mandatory or optional for the verb and what vibhaktis (postpositions) they take respectively. Each verb belongs to a specific verb class and each class has a basic karaka frame. Each Tense, Aspect and Modality (TAM) of a verb specifies a transformation rule.

Demand frame for a verb

A demand frame or karaka frame for a verb indicates the demands that a verb makes. It depends on the verb and its TAM label. A mapping is specified between karaka relations and vibhaktis (post-positions, suffix).

Transformation

Based on the TAM of the verb, a transformation is made on the verb frame taking reference of TAM frame.

Developing a verb and a TAM frame

In developing the verb frame for various verbs and their associated derivatives, we have considered present tense, first person and singular to be the primary frame. In this regard, we will have the frame for Nepali verbs like पढ्छ, खान्छ, दिन्छ, भन्छ and so on. Similarly we will have the frame (modifier rule) for Verb modifier like यो, छ, एला,नेछ, and so on.

In tables 2,3 and 4, we present sample karaka frames for दिन्छfियो and karaka frame transformation from दिन्छ to दियो .

Table 2. Sample karaka frame दिन्छ

Arc label	Necessity	Vibhaktis	Lextype	Arc pos	Arc dir
K1	M	Null	n	l	c
K2	M	Null/को	n	l	c
K3	D	द्वारा	n	l	c
K4	D	लाई	n	l	c

Table 3. Sample Transformation Rule यो

Arc label	Necessity	Vibhaktis	Lextype	Arc pos	Arc dir
K1	M	ले	-	-	-

Table 4. Sample Transformation frame (यो transforming दिन्छ to make दियो frame)

Arc label	Necessity	Vibhaktis	Lextype	Arc pos	Arc dir
K1	M	ले (transformed)	n	l	c
K2	M	Null/को	n	l	c
K3	D	द्वारा	n	l	c
K4	D	लाई	n	l	c

Steps of parsing

There are altogether four steps in the parsing process as outlined below:

- **Finding the verb candidate**

First of all, the verbs (candidate) in the sentence should be identified. From the identified verb, its seed lexicon and TAM should be identified, i.e. if the verb is पढेको,

Seed lexicon is पढ् and TAM is एको.

- **Identifying the verb frame and making the necessary transformation**
  On the basis of this verb seed lexicon, the verb frame to be loaded is identified. Similarly, on the basis of TAM, the TAM frame to be loaded is identified.
- **Labeling of the arc**
  With the help of transformed verb frame, we will start labeling the arc.
- **Imposing the constraints**
  Once the arcs are labeled, we will start filtering the unnecessary arcs on the basis of the following constraints.
  - C1: For each of the mandatory demands in a demand frame for each demand group, there should be exactly one outgoing edge labeled by the demand from the demand group.
  - C2: For each of the optional demands frame for each demand group, there should be at most one outgoing edge labeled by the demand from the demand group
  - C3: There should be exactly one incoming arc into each source group.

Integer Programming Constraints (Constraints Equations)

Let X_{ijk} represents a possible arc from word group *i* to *j* with karaka label *k*. It takes value *1* if the solution has that arc and *0* otherwise. It cannot take any other values. The constraints rules are formulated into constraints equations.

- C1: For each demand group *i*, for each of its mandatory demands *k*, the following equations must hold.
  $M_{ik} : \sum_j X_{ikj} = 1$
- C2: For each demand group *i*, for each of its optional or desirable demands k, the following inequalities must hold.
  $O_{ik} : \sum_j X_{ikj} <= 1$

- C3: For each of the source groups *j*, the following equalities must hold

  $S_j : \sum_{jk} X_{ikj}=1$

Parsing example

Sentence: रामले खाना खाएर मोहनलाई पुस्तक दियो।

Step1: We will find the verb candidates for the sentence i.e. दियो and खाएर. We will make a word break analysis for दियो whose seed lexicon is दि. In this regard, दिन्छ verb frame (Table 2) is loaded. Since the postposition for दियो is यो, the corresponding frame for यो is also loaded (Table 3). Now the frame for दिन्छ is transformed by यो to get a transformed frame for दियो (Table 4). On the basis of this frameflwe will make the graph as shown in the figure.

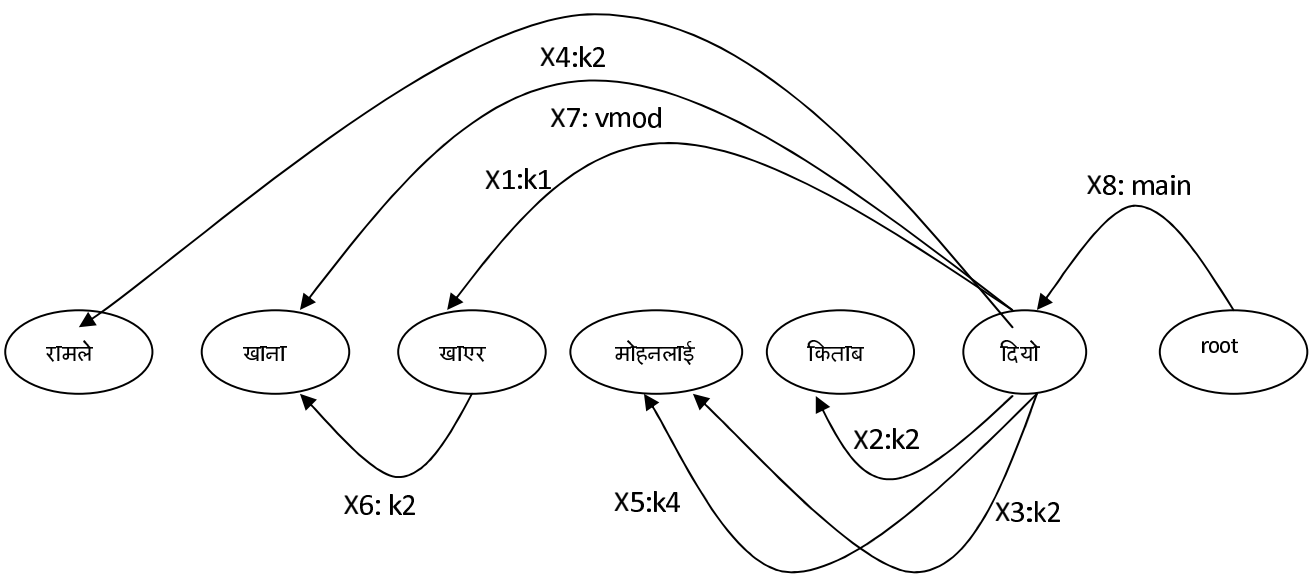


Fig. 2: Graph formed from the transformed frame

Step 2: On the basis of constraints equation, we will filter various arcs and from the above and get the final parse.

- Verb दिन्छ
  - Mandatory Demands(C1)

        - K1x1=1
        - K2x2+x3+x4=1
    - Optional demands(c2)
        - K4x5=1

- Verb खान्छ
    - Mandatory Demands(C1)
        - K2x6=1
        - vmodx7=1
- _ROOT_
    - C1
        - Mainx8=1

- Incoming arcs source(C3)
    - राम
        - X1-1
    - खाना
        - X4+x6=1
    - खाएर
        - X7=1
    - मोहन
        - X3+x5=1
    - किताब
        - X2=1
    - दियो
        - X8=1

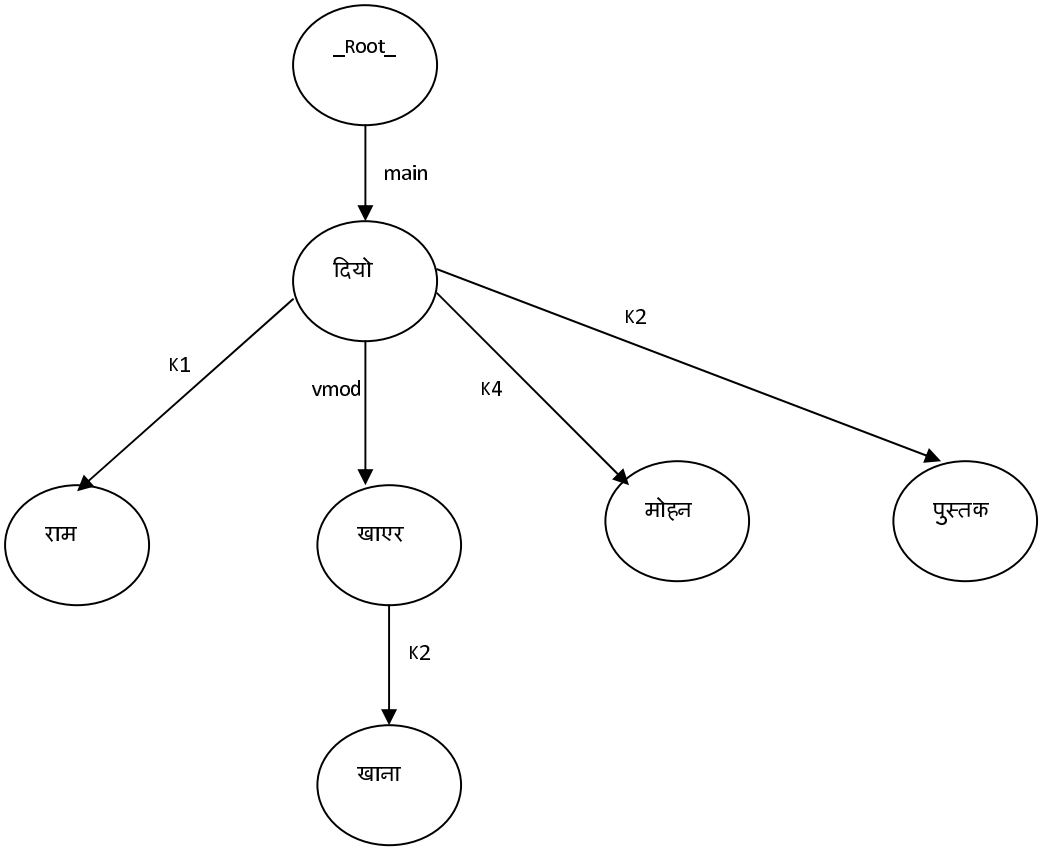


Fig. 3: Solution parse

Implementation of the Dependency Parser

For the implementation of the dependency parser several auxiliary modules were developed. These are the modules such as Word Breaker, Frame Parser, Frame Transformer and Equation Solver. All these modules are briefly discussed below.

Word Breaker Module

If we give any raw word to the Word Breaker module, it involves in breaking the word in two parts viz. Seed Lexicon and Suffix (Post Position). Basically this module performs the following three functions:

1. **Finding the verb candidate**

   Firstly, the verbs candidates in the sentence are identified, after tokenizing the words from the sentences in the text file.

2. **Separation of the verb into seed and suffix**

After identifying the verb, its seed lexicon and post position are identified.

For example, if the verb is “दियो´ , Seed lexicon is “दि´˜and the post position is “यो´ .

Matching the suffixes in input word

Once the input word that is to be fragmented to its corresponding Seed Lexicon and Post Position is encountered, all the data from the suffix file is loaded. The loaded suffixes data will then be traversed linearly to check its availability as a suffix in the input word. Once it is found, the suffix from the input word is truncated and the potential seed remains. The seed data from the seed file is then loaded. With the potential seed remaining from above its presence in the seed data is checked. If it is not there a single character from its left will be deleted and rechecking will be done similarly. Again if this doesn’t work then we add a special character " ्˜" (हलन्त) in the potential seed and check whether it is present in the provided Seed’s database. In this way the seed and suffix are identified.

Example 1: If the match is found for the seed lexicon word “दि´ , then its frame is loaded.

Example 2: If the verb is “हिँड्दै´. And if the match is not found for the seed lexicon word “हिँड्दै´, then a character is removed from the root lexicon (“ँ´˜is removed). If the match is found for the seed lexicon word “हिड्´˜after removing a character, then its respective frame is loaded.

Word Breaker Simplified Architecture

The individual files of seed lexicon and post position database are loaded by the Load ( ) operation. The seed will be loaded into a new Hash table, whereas the post position will be loaded into a list. The break ( ) operation breaks the given word into two parts. One is its Seed Lexicon and another is the Post Position. The GetSeed ( ) operation will traverse the Seed Lexicon file and matches the corresponding Seed of the given word and the GetPostPosn ( ) operation will match the corresponding post position word.

WordBreaker Class Diagram

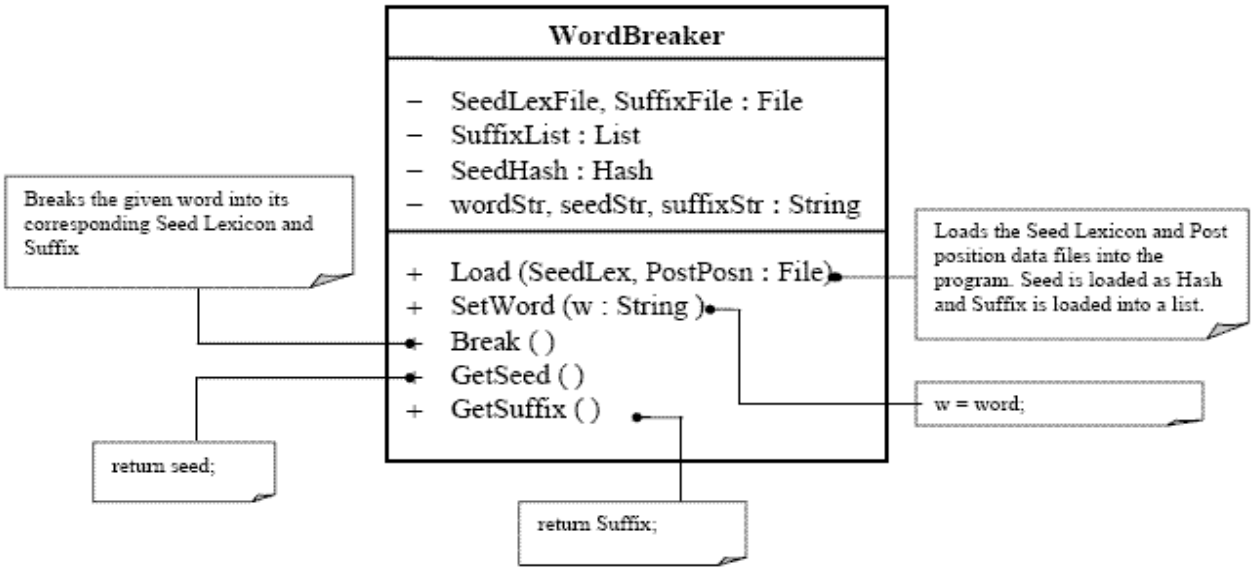


Fig. WordBreaker Class Diagram

Usage

Following is the method of initialization of the WordBreaker object.

WordBreaker wb = new WordBreaker (seedFile, postPosFile);

…

…

…

wb.SetWord (" any string here.");

seedStr = wb.GetSeed( );

postStr = wb.GetPost( );

Frame Parser Module

In our WordBreaker module, we have already accomplished the job of breaking a given word (verb) into its corresponding seed and post position. FrameParser module takes the input word from the WordBreaker. Depending upon the type of the verb, FrameParser loads the frame of the respective verb. A frame is simply a xml document whose item node contains entry for the corresponding verb and its necessary constituents.

The Frame Parser module performs the following functions:

- **Identifying the corresponding item node**

  According to the group identified by the WordBreaker the corresponding frame file is loaded as Type 'A' or Type 'B'. The post position of each item node of the xml document is traversed and searched linearly. This will give the address of the item node which contains the desired postposition candidate. On the basis of the address of this item we continue further processing.

- **Loading the frames of the item node identified**

  After identifying the item node which contains the desired postposition, all the frames that are child nodes of that node are loaded. The item node contains one or many "*kframe*" element. This "*kframe*" contains children nodes as *karaka*, *lextype*, *vibakti1*, *mandatory* and *position* respectively. Depending upon the text contents of the children nodes, FrameParser module loads them creating an internal storage buffer.

Frame Parser Class Diagram

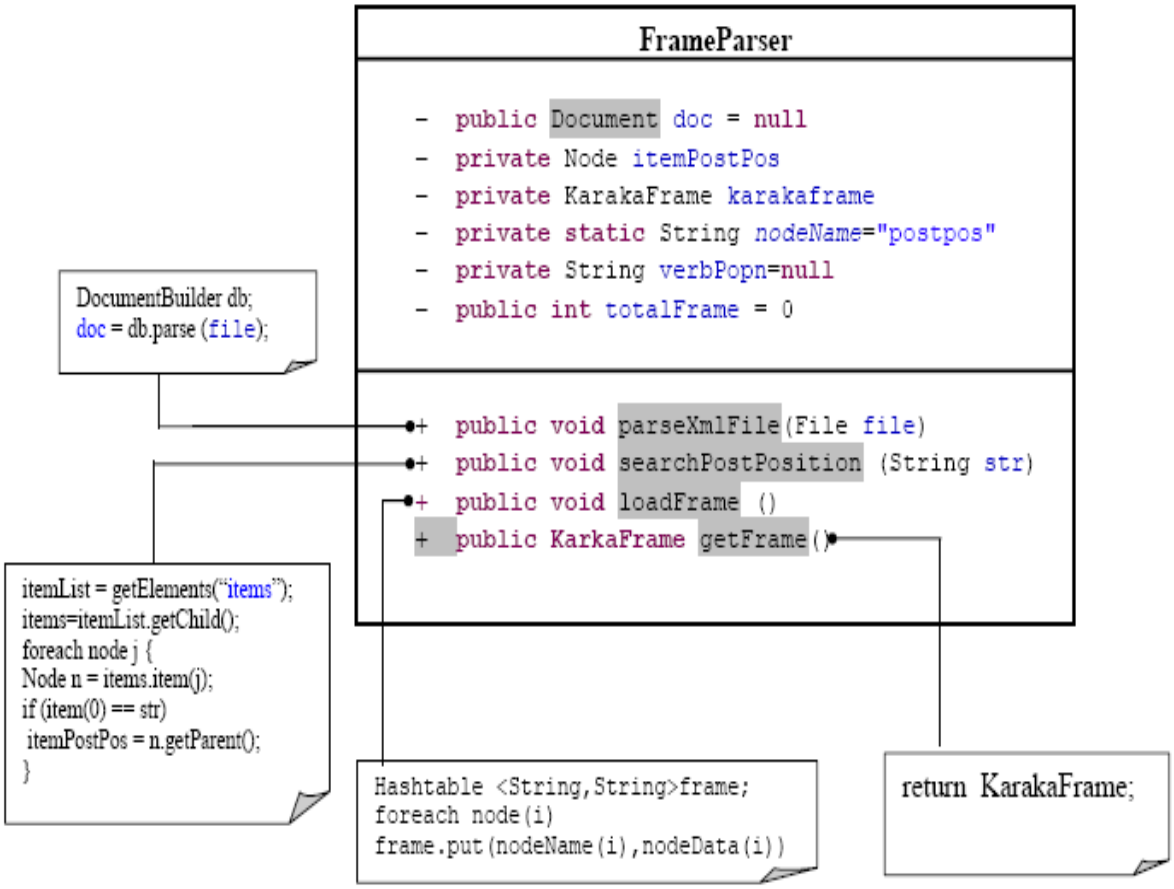


Fig. FrameParser Class Diagram

Frame Transformer Module

For frame transformation, frames are divided into two categories as root frame and transformer frame. "*Karaka*" element of the transformer frame is obtained by traversing the transformer frame. Next, root frame is traversed and its respective "*Karaka*" element is extracted. Both "*Karaka*" elements obtained from transformer frame and root frame are checked whether they

are equal. If they are equal, then the "*Vibakti1*" element of the root frame is transformed by the "*Vabakti1*" element from the transformer frame.

Based on the TAM (**T**ense **A**spect **M**odality) of the verb, a transformation is made on the verb frame on the basis of TAM frame. In making verb frame for various verbs and its associated derivatives, we consider present tense, first person and singular to be the primary frame. In this regard, we will have the frame for Nepali verbs like ***पढ्छ***, ***खान्छ***, ***दिन्छ***,***भन्छ***~and so on. Similarly we will have the frame (modifier rule) for Verb modifier like ***यो***, ***छ***, ***ैला***, ***नेछ***, and so on.

The Equation Generator Module

The equation generator module represent equation in terms of string that contains only 0 and 1. So if there are total of n arc variable in total than the length of string is n and 1 in the string represents that coefficient of that variable is 1. For example, if we have the following equations:

X1+X2=1

X2+X3=1

Then the equations will be represented as 110 and 011 respectively.

Equation Solver Module (Solving the Constraints Satisfaction Problem)

We can obtain the raw graph from the previous module which contains multiple arcs. Our aim is to find a mechanism to satisfy the given constraints in order to produce a final solution graph. For obtaining this solution, the idea is to compute all the possible combinations of each variable. Then all the possible values obtained are substituted in the equation created following the algorithm.

On basis of equations formed, let us say we have total of n unsolved variables $X_1 \ldots X_n$. Since each of the variables can either take a discrete value of either 0 or 1, the total possible combinations solutions are 2^n . Out of these 2^n possible combinations only few will satisfy our constraints that will ultimately be our solutions. If in case we get more than one solutions we will have multiple parse of our solutions.

Once we get our solution, we remove all those arcs whose value have zeros.

For example, in the following initial graph:

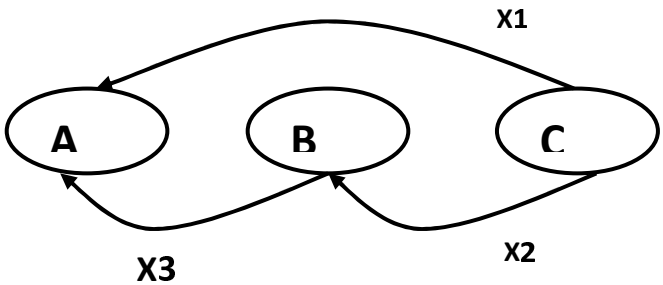


Fig: Example initial Graph

If the following is equations is formed from the constraints:

$X_1 + X_2 = 1$

$X_2 + X_3 = 1$

Than we will have following sets of combinations as shown in Table 5:

Table 5. Possible combinations on the values of the variables

Combination No.	**X_1**	**X_2**	**X_3**
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Out of which only combination No 3 (0 1 0) and combination No 6 (1 0 1) satisfy the constraints. Hence two possible parses follow from the above.

In this way once we get the solution, we will remove those arcs whose corresponding arc variables have value 0.

Hence, the parsing one of the complicated modules in the NCG, currently parses a given input chunked sentence with an accuracy rate of approximately 90% but the accuracy would highly depend upon the accuracy of the chunker module. The other limitation is that the parser currently parses only simple declarative sentences with just one verb candidate. Sentence with multiple verb candidates and joiners are currently not handled. This limitation is however expected to get addressed as we develop some more chunk rules. Below, we give a sample of the parsed and analyzed sentence.

```
=====================================================================
Sentence is : {राम/NNP ले/PLE }/NCH {हरि/NN लाई/PLAI }/NCH {दियो/VBF }/VCH
Candidate verb found :दियो
Seed : दि Suffix : यो Group: a

Relationship finding between VERB: दियो and CHUNK: {राम/NNP ले/PLE }/NCH
starts.**************************
{राम/NNP ले/PLE }/NCH --> vibhakti: ले lextype: NCH

Start of Frame 0 for Verb--->दियो
Vibhaktis: ले
karaka: k1
Lextype: NCH
Demand type: YES

Relation established.

Incoming arc Variable added to chunk: {राम/NNP ले/PLE }/NCH
{दियो/VBF }/VCH -->{राम/NNP ले/PLE }/NCH  by karaka:k1 Demand is :YES



OutGoing arc Variable added to chunk: {दियो/VBF }/VCH
{दियो/VBF }/VCH -->{राम/NNP ले/PLE }/NCH  by karaka:k1 Demand is :YES


End of Frame 0 for Verb--->दियो
```

```
Start of Frame 1 for Verb--->दियो
Vibhaktis: लाई,PHI
karaka: k2
Lextype: NCH
Demand type: YES
End of Frame 1 for Verb--->दियो

Start of Frame 2 for Verb--->दियो
Vibhaktis: ले,PHI
karaka: k3
Lextype: NCH
Demand type: NO

Relation established.

Incoming arc Variable added to chunk: {राम/NNP ले/PLE }/NCH
{दियो/VBF }/VCH -->{राम/NNP ले/PLE }/NCH  by karaka:k3 Demand is :NO


OutGoing arc Variable added to chunk: {दियो/VBF }/VCH
{दियो/VBF }/VCH -->{राम/NNP ले/PLE }/NCH  by karaka:k3 Demand is :NO


End of Frame 2 for Verb--->दियो

Start of Frame 3 for Verb--->दियो
Vibhaktis:
कालागि,मा,प्रति,तर्फ,बाट,देखि,सँग,तिर,साथ,माथि,सम्म,बारे,निर,द्वारा,वाहेक,भित्र,बीच,समक्ष,बाट,भरि,बाहिर
,अगाडि,अनुरूप,पट्टि,नेर,सामु,पछाडि,मध्ये,सँगै,बाहेक,अनुसार,अन्तर्गत,मार्फत,सित,बमोजिम,वापत,निम्ति,का
निम्ति,सम्बन्धि,मुनि,मध्ये,स्थित,पर्यन्त,देखि
karaka: kx
Lextype: NCH
Demand type: NO
End of Frame 3 for Verb--->दियो

Relationship finding between VERB: दियो and CHUNK: {राम/NNP ले/PLE }/NCH
ends. *************************

Relationship finding between VERB: दियो and CHUNK: {हरि/NN लाई/PLAI }/NCH
starts.*************************
{हरि/NN लाई/PLAI }/NCH --> vibhakti: लाई lextype: NCH


Start of Frame 0 for Verb--->दियो
Vibhaktis: ले
karaka: k1
Lextype: NCH
```

```
Demand type: YES
End of Frame 0 for Verb--->दियो

Start of Frame 1 for Verb--->दियो
Vibhaktis: लाई,PHI
karaka: k2
Lextype: NCH
Demand type: YES

Relation established.

Incoming arc Variable added to chunk: {हरि/NN लाई/PLAI }/NCH
{दियो/VBF }/VCH -->{हरि/NN लाई/PLAI }/NCH  by karaka:k2 Demand is :YES



OutGoing arc Variable added to chunk: {दियो/VBF }/VCH
{दियो/VBF }/VCH -->{हरि/NN लाई/PLAI }/NCH  by karaka:k2 Demand is :YES


End of Frame 1 for Verb--->दियो

Start of Frame 2 for Verb--->दियो
Vibhaktis: ले,PHI
karaka: k3
Lextype: NCH
Demand type: NO
End of Frame 2 for Verb--->दियो

Start of Frame 3 for Verb--->दियो
Vibhaktis:
कालागि,मा,प्रति,तर्फ,बाट,देखि,सँग,तिर,साथ,माथि,सम्म,बारे,निर,द्वारा,वाहेक,भित्र,बीच,समक्ष,बाट,भरि,बाहिर
,अगाडि,अनुरूप,पट्टि,नेर,सामु,पछाडि,मध्ये,सँगै,बाहेक,अनुसार,अन्तर्गत,मार्फत,सित,बमोजिम,वापत,निम्ति,का
निम्ति,सम्बन्धि,मुनि,मध्ये,स्थित,पर्यन्त,देखि
karaka: kx
Lextype: NCH
Demand type: NO
End of Frame 3 for Verb--->दियो

Relationship finding between VERB: दियो and CHUNK: {हार/NN लाई/PLAI }/NCH
ends. *************************

Incoming eq: 110 Added

Incoming eq: 001 Added

Out going mandatory eq: 100 Added
```

```
Out going mandatory eq: 001 Added

Out going optional eq: 010 Added

Equation is solved. Following are the solutoins:
101

Solution parse: 0 starts ****************
{दियो/VBF }/VCH -->k1<--{राम/NNP ले/PLE }/NCH
{दियो/VBF }/VCH -->k2<--{हरि/NN लाई/PLAI }/NCH
Solution parse: 0 ends ****************
```

Interpreting the solution parse

Between the first NCH and VCH, there is a valid K1 relation whereas between the second NCH and the VCH, we have a valid K2 relation. Hence, the sentence रामले हरिलाई दियो is a valid sentence.

==========

Conclusion

The document reported the research and implementation aspects of the different modules of the NCG. The NCG system, which represents as an integrated system of all of these modules presents as output – parsed and analyzed sentence to an input simple declarative sentence with just one verb candidate present. What remains missing or what could be added to the NGC is the "Agreement Module" that would further filter the parses returned by the parser module taking feature agreement like gender, number, person, tense, aspect and modality as governing attributes. We recommend this module to be worked on as a further research to the NCG work done so far.

Acknowledgment

The PAN L10n works have been carried out with the aid of a grant from the International Development Research Centre, Ottawa, Canada, administered through the Center for Research in Urdu Language Processing (CRLUP), National University of Computing and Emerging Sciences, Lahore, Pakistan (NUCES).

Special thanks to Mr. Throstan Brant (Saarland University, Computational Linguistics, Saarbrucken, Germany) for providing us with the TnT tagger for our research and development.

Our special thanks also go to the engineering student from Kathmandu University Mr. Niraj Pokhrel, Ms. Srijana Pokhrel and Ms. Dipti Sharma for helping us with the project as part of their internship.

References

[1] A. Bharati, V. Chaitanya, and R. Sangal, *Natural Language Processing - A Paninian Perspective*, Eastern Economy Edition ed. Kanpur: Prentice Hall, New Delhi, 1995.

[2] A. Bharati, R. Sangal, and T. P. Reddy, "A Constraint Based Parser Using Integer Programming," in *Proceedings of the ICON-2002*, Mumbai, 2002, pp. 121-127.

[3] J. Nivre. http://w3.msi.vxu.se. [Online]. HYPERLINK "http://w3.msi.vxu.se/~nivre/papers/05133.pdf" http://w3.msi.vxu.se/~nivre/papers/05133.pdf

[4] A. Bharati and R. Sangal, "Parsing free word order languages in the Paninian framework.," in *Proceedings of the 31st Annual Meeting on Association For Computational Linguistics (Columbus, Ohio, June 22 - 26, 1993). Annual Meeting of the ACL.*, Morristown, NJ, 1993, pp. 105-111.

[5] M. Pedersen, D. Eades, S. Amin, and L. Prakash, "Relative clauses in Hindi and Arabic: A paninian dependency grammar analysis," in *Proceedings of the Twentieth International Conference on Computational Linguistics*, Geneva, 2004, pp. 17-24.