Sanskrit Compound Processor

Anil Kumar[1], Vipul Mittal[2] and Amba Kulkarni[1]

[1] Department of Sanskrit Studies, University of Hyderabad, India
anil.lalit22@gmail.com, apksh@uohyd.ernet.in
[2] Language Technologies Research Centre, IIIT, Hyderabad, India
vipulmittal@research.iiit.ac.in

Abstract. Sanskrit is very rich in compound formation. Typically a compound does not code the relation between its components explicitly. To understand the meaning of a compound, it is necessary to identify its components, discover the relations between them and finally generate a paraphrase of the compound. In this paper, we discuss the automatic segmentation and type identification of a compound using simple statistics that results from the manually annotated data.

Key words: Sanskrit Compound Splitter, Sanskrit Compound Type Identifier, Sanskrit Compounds, Optimality Theory.

1 Introduction

In recent years Sanskrit Computational Linguistics has gained momentum. There have been several efforts towards developing computational tools for accessing Sanskrit texts ([12], [14], [23], [16], [7], [1]). Most of these tools handle morphological analysis and sandhi splitting. Some of them ([10], [13], [8]) also do the sentential parsing. However, there have been almost negligible efforts in handling Sanskrit compounds, beyond segmentation.

Sanskrit is very rich in compound formation. The compound formation being productive, forms an open-set and as such it is also not possible to list all the compounds in a dictionary. The compound formation involves a mandatory *sandhi*[3]. But mere *sandhi* splitting does not help a reader in identifying the meaning of a compound. Typically a compound does not code the relation between its components explicitly. To understand the meaning of a compound, thus, it is necessary to identify its components, discover the relations between them, and finally produce a *vigrahavākya*[4] of the compound. Gillon [6] suggests tagging of compounds by enriching the context free rules. He does so by specifying the vibhakti, marking the head and also specifying the enriched category

[3] *Sandhi* means euphony transformation of words when they are consecutively pronounced. Typically when a word w_1 is followed by a word w_2, some terminal segment of w_1 merges with some initial segment of w_2 to be replaced by a "smoothed" phonetic interpolation, corresponding to minimising the energy necessary to reconfigure the vocal organs at the juncture between the words. [11]

[4] An expression providing the meaning of a compound is called a *vigrahavākya*.

of the components. He also points out how certain components such as *'na'* provide a clue for deciding the type of a compound. *Pāṇini* captures special cases and formulates rules to handle them. Implementing these rules automatically is still far from reality on account of the semantic information needed for the implementation. In the absence of such semantic information, statistical methods have proved to be a boon for the language engineers. These statistical tools use manually annotated data for automatic learning, and in turn develop a language model, which is then used for analysis. In what follows we discuss the automatic segmentation and type identification of compounds using simple statistics that result from the manually annotated data.

2 Sanskrit Compounds

The Sanskrit word *samāsaḥ* for a compound means *samasanam* which means "combination of more than one word into one word which conveys the same meaning as that of the collection of the component words together". While combining the components together, a compound undergoes certain operations such as loss of case suffixes, loss of accent, etc.. A Sanskrit compound thus has one or more of the following features ([17], [6]):

1. It is a single word (*ekapadam*).[5]
2. It has a single case suffix (*ekavibhaktikam*) with an exception of *aluk* compounds such as *yudhiṣṭiraḥ*, where there is no deletion of case suffix of the first component.
3. It has a single accent(*ekasvaraḥ*).[6]
4. The order of components in a compound is fixed.
5. No words can be inserted in between the compounds.
6. The compound formation is binary with an exception of *dvandva* and *bahupada bahuvrīhi*.
7. Euphonic change (*sandhi*) is a must in a compound formation.
8. Constituents of a compound may require special gender or number different from their default gender and number. e.g. *pāṇipādam, pācikābhāryaḥ*, etc.

Though compounds of 2 or 3 words are more frequent, compounds involving more than 3 constituent words with some compounds even running through pages is not rare in Sanskrit literature. Here are some examples of Sanskrit compounds involving more than 3 words.

1. वेदवेदाङ्गतत्त्वज्ञः[7]=वेद - वेदाङ्ग - तत्त्व-ज्ञः
2. प्रवरमुकुटमणिमरीचिमञ्जरीचयचर्चितचरणयुगल[8]=प्रवर - मुकुट - मणि - मरीचि - मञ्जरी - चय - चर्चित - चरण - युगल

[5] aikapadyam aikasvaryam ca samāsatvāt bhavati [Kāśikā 2.1.46]
[6] aikapadyam aikasvaryam ca samāsatvāt bhavati [Kāśikā 2.1.46]
[7] *Rāmāyaṇam 1-1-14*
[8] *Pañcatantram in kathāmukham*

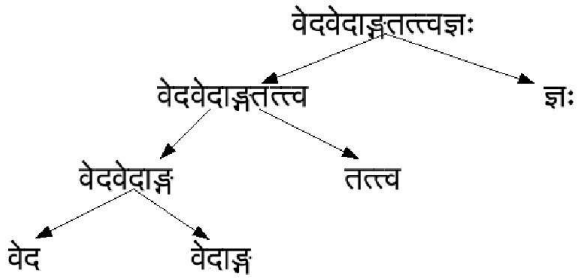

Fig. 1. Constituency representation of वेदवेदाङ्गतत्त्वज्ञः

3. जलादिव्यापकपृथिवीत्वाभावप्रतियोगिपृथिवीत्ववती[9]=जल - आदि - व्यापक - पृथिवीत्व - अभाव - प्रतियोगि - पृथिवीत्ववती

The compounds are formed with two words at a time and hence they can be represented faithfully as a binary tree, as shown in figure 1.

Semantically *Pāṇini* classifies the Sanskrit compounds into four major types:

- Tatpuruṣaḥ: (Endocentric with head typically to the right),
- Bahuvrīhiḥ: (Exocentric),
- Dvandvaḥ: (Copulative), and
- Avyayībhāvaḥ: (Endocentric with head typically to the left and behaves as an indeclinable).

This classification is not sufficient for generating the paraphrase. For example, the paraphrase of a compound *vṛkṣamūlam* is *vṛkṣasya mūlam* and *gṛhagataḥ* is *gṛham gataḥ*, though both of them belong to the same class of *tatpuruṣaḥ*. Based on their paraphrase, these compounds are further sub-classified into 55 sub-types. Appendix-A provides the classification and Appendix-B describes the paraphrase associated with each tag.

3 Compound Processor

Understanding a compound involves (ref Fig 2)

1. Segmentation (समासपदच्छेदः),
2. Constituency Parsing (समासपदान्वयः),
3. Compound Type Identification (समस्तपदपरिचायकः), and
4. Paraphrasing (विग्रह-वाक्यम्).

These four tasks form the natural modules of a compound processor. The output of one task serves as an input for the next task until the final paraphrase is generated.

[9] *Kevalavyatireki-prakaraṇam : Maṇikaṇa* [22]

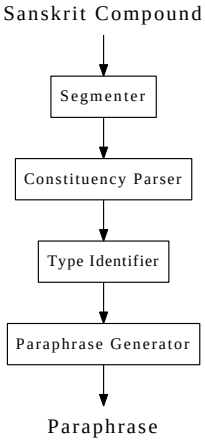

Fig. 2. Compound Analyser

3.1 Segmenter

The task of this module is to split a compound into its constituents. For instance, the compound

sumitrānandavardhanaḥ
is segmented as
sumitrā-ānanda-vardhanaḥ

Each of the constituent component except the last one is typically a compounding form (a bound morpheme)[10].

3.2 Constituency Parser

This module parses the segmented compound syntactically by pairing up the constituents in a certain order two at a time. For instance,

sumitrā-ānanda-vardhanaḥ
is parsed as
<sumitrā-<ānanda-vardhanaḥ>>

3.3 Type Identifier

This module determines the type on the basis of the components involved. For instance,

<sumitrā-<ānanda-vardhanaḥ>>
is tagged as

[10] with an exception of components of an *'aluk'* compound.

$$<\textbf{\textit{sumitrā-}}<\textbf{\textit{ānanda-vardhanaḥ}}>\textbf{\textit{T6}}>\textbf{\textit{T6}} \quad .$$

where T6 stands for compound of type *ṣaṣṭī-tatpuruṣa*. This module needs an access to the semantic content of its constituents, and possibly even to the wider context.

3.4 Paraphrase Generator

Finally after the tag has been assigned, the paraphrase generator [16] generates a paraphrase for the compound. For the above example, the paraphrase is generated as:
ānandasya vardhanaḥ = *ānandavardhanaḥ*,
sumitrāyāḥ ānandavardhanaḥ = *sumitrānandavardhanaḥ*.

4 Compound Segmenter

The task of a segmenter is to split a given sequence of phonemes into a sequence of morphologically valid segments [18]. The compound formation involves a mandatory sandhi. Each sandhi rule is a triple (x, y, z) where y is the last letter of the first primitive, z is the first letter of the second primitive, and x is the letter sequence resulting from the euphonic combination. For analysis, we reverse these rules of sandhi and produce $y + z$ corresponding to a x. Only the sequences that are morphologically valid are selected. The segmentation being non-deterministic, segmenter produces multiple splits. To ensure that the correct output is not deeply buried down the pile of incorrect answers, it is natural to prioritize solutions based on some scores. We follow GENerate-CONstrain-EVALuate paradigm attributed to the Optimality Theory [21] for segmentation. As is true of any linguistic theory, the Optimality Theory basically addresses the issue of generation. Nevertheless, there have been successful attempts ([4], [3]) to reverse the process of generation. It will be really challenging to implement the sandhi rules from *Aṣṭādhyāyī* as CONstraints and then reverse them for splitting. In this attempt however, we simply use the 'cooked' sandhi rules in the form of triplets. We describe below the GENerate-CONstrain-EVALuate cycle of the segmenter and the scoring matrix used for prioritizing the solutions.

4.1 Scoring Matrix

A parallel corpus of Sanskrit text in sandhied and unsandhied form is being developed as a part of the Sanskrit Consortium project in India. The corpus contains texts from various fields ranging from children stories, dramas, purāṇas to Ayurveda texts. From around 100K words of such a parallel corpus, 25K parallel instances of sandhied and unsandhied text were extracted. These examples were used to get the frequency of occurence of various sandhi rules. If no instance

of a sandhi rule is found in the corpus, for smoothing, we assign the frequency of 1 to this sandhi rule.

We define the estimated probability of the occurrence of a sandhi rule as follows: Let R_i denote the i^{th} rule with f_{R_i} as the frequency of occurrence in the manually split parallel text. The probability of rule R_i is:

$$P_{R_i} = \frac{f_{R_i}}{\sum_{j=1}^{n} f_{R_j}}$$

where n denotes the total number of sandhi rules found in the corpus.

Let a word be split into a candidate S_j with k constituents as $< c_1, c_2, ..., c_k >$ by applying $k-1$ sandhi rules $< R_1, R_2, ..., R_{k-1} >$ in between the constituents. It should be noted here that the rules $R_1, ..., R_{k-1}$ and the constituents $c_1, ..., c_k$ are interdependent since a different rule sequence will result in a different constituents sequence. The sequence of constituents are constrained by a language model whereas the rules provide a model for splitting. We define two measures each corresponding to the constituents and the rules to assign weights to the possible splits.

Language Model Let the unigram probability of the sequence $< c_1, c_2, ..., c_k >$ be PL_{S_j} defined as:

$$PL_{S_j} = \prod_{x=1}^{k} (P_{c_x})$$

where P_{c_x} is the probability of occurrence of a word c_x in the corpus.

Split Model Let the splitting model PS_{S_j} for the sandhi rules sequence $< R_1, R_2, ..., R_{k-1} >$ be defined as:

$$PS_{S_j} = \prod_{x=1}^{k-1} (P_{R_x})$$

where P_{R_x} is the probability of occurrence of a rule R_x in the corpus.

Therefore, the weight of the split S_j is defined as the product of the language and the split model as:

$$W_{S_j} = \frac{PL_{S_j} * PS_{S_j}}{k}$$

where the factor of k is introduced to give more preference to the split with less number of segments than the one with more segments.

4.2 Segmentation Algorithm

The approach followed is GENerate-CONstrain-EVALuate. In this approach, all the possible splits of a given string are first generated and the splits that are not validated by the morphological analyser are subsequently pruned out.

Currently we apply only two constraints viz.

- C1 : All the constituents of a split must be valid morphs.
- C2 : All the segments except the last one should be valid compounding forms.

The system flow is presented in Figure 3.

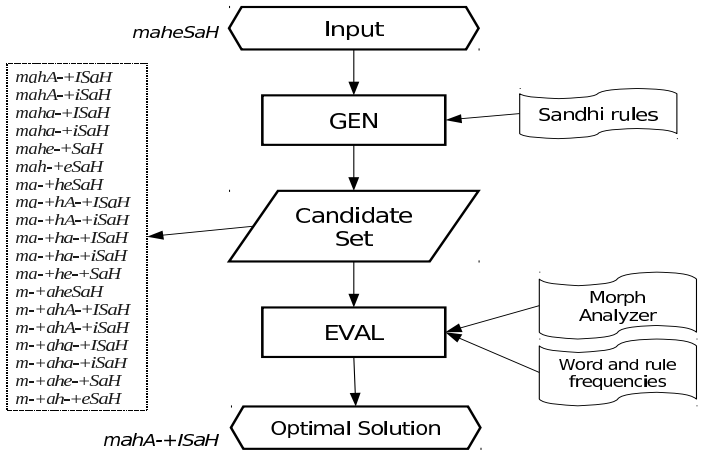

Fig. 3. Compound Splitter: System Data Flow.

The basic outline of the algorithm is:

1. Recursively break a word at every possible position applying a sandhi rule and generate all possible candidates for the input.
2. Pass the constituents of all the candidates through the morph analyser.
3. Declare the candidate as a valid candidate, if all its constituents are recognised by the morphological analyser, and all except the last segment are compounding forms.
4. Assign weights to the accepted candidates and sort them based on the weights as defined in the previous subsection.
5. The optimal solution will be the one with the highest weight.

Results The current morphological analyser[11] can recognise around 140 million words. Using 2,650 rules and a test data of around 8,260[12] words parallel corpus for testing, we obtained the following results:

- Almost 92.5% of the times, the first segmentation is correct. And in almost 99.1% of the cases, the correct split was among the top 3 possible splits.
- The precision was about 92.46% (measured in terms of the number of words for which first answer is correct w.r.t. the total words for which correct segmentation was obtained).
- The system consumes around 0.04 seconds per string of 15 letters on an average.[13]

The complete rank wise distribution is given in Table 1.

Rank	**% of words**
1	92.4635
2	5.0492
3	1.6235
4	0.2979
5	0.1936
>5	0.3723

Table 1. Rank-wise Distribution.

5 Constituency Parser

Segmenter takes a compound as an input and produces one or more possible segmentations conditioned by the morphological analyser and the sandhi rules. Constituency parser takes an output of the segmenter and produces a binary tree showing the syntactic composition of the compound corresponding to each of the possible segmentations. Each of these compositions show the possible ways various segments can be grouped. To illustrate various possible parses that result from a single segmentation, consider the segmentation a-b-c of a compound. Since a compound is binary, the three components a-b-c may be grouped in two ways as $< a < bc >>$ or $<< ab > c >$. Only one of the ways of grouping may be correct in a given context as illustrated by the following two examples.

1. < *baddha-* < *jāmbūnada- srajaḥ* >>
2. << *tapas-svādhyāya* >- *niratam* >

[11] available at http://sanskrit.uohyd.ernet.in/ anusaaraka/sanskrit/samsaadhanii/morph/index.html.
[12] The test data is extracted from manually split data of *Mahābhāratam*.
[13] Tested on a system with 2.93GHz Core 2 Duo processor and 2GB RAM.

With 3 components, only these two parses are possible. But as the number of constituents increase, the number of possible ways the constituents can be grouped grows very fast. The constituency parsing is similar to the problem of completely parenthesizing *n+1* factors in all possible ways. Thus the total possible ways of parsing a compound with $n+1$ constituents is equal to a *Catalan number,* C_n [13] where for $n \geq 0$,

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

The task of the constituency parser is then to choose the correct way of grouping the components together, in a given context. To the best of our knowledge no work has been initiated yet that produces the best constituency parse for a given segmentation, in a given context.

6 Type Identifier

After getting a constituency parse of a compound, the next task in the compound analysis is to assign an appropriate operator (tag) to each non-leaf node. This operator will then operate on the leaf nodes to produce the associated meaning.

We use manually tagged corpus as a model for predicting the tags, given a pair of constituents of a compound. Manually tagged corpus consists of approximately 150K words which contain 12,796 compounds[14]. These texts were tagged using the tagset given in appendix-I. All these compounds are thus tagged 'in context' and contain only one tag. This corpus formed the training data. Another small corpus with 400 tagged compounds from totally different texts, was kept aside for testing.

Some features of the manually tagged data

1. Around 10% of the compounds were repeated.
2. The 12,796 tokens of compounds contain 2,630 types of left word and 7,106 types of right word.
3. The frequency distribution of highly frequent tags is shown in Table 2. To study the effect of fine-grained-ness we also merged the sub-types. Table 3 gives the frequency distribution of major tags, after merging the sub-types.

We define,

$$P(T/W_1 - W_2) = \text{probability that a compound } W_1 - W_2 \text{ has tag T.}$$

Assuming that occurrence of W_1 and W_2 are independent,

[14] Only compounds with 2 components were selected

Tag	% of words
T6	28.35
Bs6	12.45
K1	9.63
Tn	8.56
Di	7.23
U	5.73

Table 2. Distribution of Fine-grain-Tags

Tag	% of words
T	52.43
B	18.96
K	12.04
D	8.84
U	5.73

Table 3. Distribution of Coarse-Tags

$$P(T/W_1 - W_2) = P(T/W_1) * P(T/W_2)$$

where $P(T/W_i) = \frac{P(T*W_i)}{P(W_i)}$, i= 1,2.

Since the data is sparse, to account for smoothing, we define, for unseen instances,

$P(T.W_i) = \frac{1}{F}$,
$P(W_i) = \frac{1}{F}$,

where F is the total number of manually tagged compounds.

6.1 Performance Evaluation

The test data of 400 words are tagged 'in context'. While our compound tagger does not see the context, and thus suggests more than one possible tag, and ranks them. Normally a tool is evaluated for its coverage and precision. In our case, the tool always produces tags with weights associated with them. The coverage and precision therefore are evaluated based on the ranks of the correct tags. Table4 shows results of 400 words with a coarse as well as fine grained tagset.

	with 55 tags		with 8 tags	
Rank	no of words	% of words	no of words	% of words
1	252	63.0	291	72.7
2	44	10.9	53	13.2
3	29	7.2	38	9.5

Table 4. Precision of Type Identifier.

Thus, if we consider only the 1^{st} rank, the precision with 8 tags is 72.7% and with 55 tags, it is 63.0%. The precision increases substantially to 95.4% and 81.1% respectively if we take 1^{st} three ranks into account.

The performance of the type identifier can be further improved by using semantically tagged lexicon. There are around 200 rules in the *Aṣṭādhyāyī* which

provide semantic contexts for various compounds. One such example is the *Pāṇini*'s sūtra *annena vyañjanam* (2.1.34). This sūtra gives a condition for forming *tṛtīyā tatpuruṣaḥ* compound. Thus what is required is a list of all words that denote eatables. A lexicon rich with such semantic information would enhance the performance of the tagger further. The compound processor with all the functionality described above is available online at

http://sanskrit.uohyd.ernet.in/samAsa/frame.html

References

1. Bharati, A., Kulkarni, A.P., and Sheeba, V.: *Building a wide coverage Sanskrit Morphological Analyser: A practical approach*. In: The First National Symposium on Modelling and Shallow Parsing of Indian Languages, IIT-Bombay. (2006)
2. Bhat, G.M.: *Samāsaḥ*. Samskrita Bharati, Bangalore, Karnataka, 2006.
3. Fortes, F.C.L., and Roxas, R.E.O.: *Optimality Theory in Morphological Analysis* In: National Natural Language Processing Research Symposium, January 2004.
4. Fosler, J.E.: *On Reversing the Generation Process in Optimality Theory*. In : Proceedings of the Association for Computational Linguistics. (1996)
5. Gillon, B.S.: *Exocentric Compounds in Classical Sanskrit*. In: Proceeding of the First International Symposium on Sanskrit Computational Linguistics(SCLS-2007), Paris, France, 2007.
6. Gillon, B.S.: *Tagging Classical Sanskrit Compounds*. In: Sanskrit Computational Linguistics 3, pages 98-105, Springer-Verlag LNAI 5406. (2009)
7. Hellwig, O.: *Sanskrit Tagger: A Stochastic Lexical and POS Tagger for Sanskrit*. In: Sanskrit Computational Linguistics 1 & 2, pages 266-277, Springer-Verlag LNAI 5402. (2007)
8. Hellwig, O.: *Extracting Dependency Trees from Sanskrit Texts*. In: Sanskrit Computational Linguistics 3, pages 106-115, Springer-Verlag LNAI 5406. (2009)
9. Hoeks, J.C.J., and Hendriks, P.: *Optimality Theory and Human Sentence Processing: The Case of Coordination*. In: Proceedings of the 27th Annual Meeting of the Cognitive Science Society, Erlbaum, Mahwah, NJ, pp. 959–964. (2005)
10. Huet, G.: *Shallow syntax analysis in Sanskrit guided by semantic nets constraints*. In: Proceedings of International Workshop on Research Issues in Digital Libraries, Kolkata. (2006)
11. Huet, G.: Lexicon-directed Segmentation and Tagging of Sanskrit. XIIth World Sanskrit Conference, Helsinki, Finland, Aug. 2003. In Themes and Tasks in Old and Middle Indo-Aryan Linguistics, Eds. Bertil Tikkanen and Heinrich Hettrich. Motilal Banarsidass, Delhi, 2006, pp. 307-325.
12. Huet, G.: *Formal structure of Sanskrit text: Requirements analysis for a Mechanical Sanskrit Processor*. In: Sanskrit Computational Linguistics 1 & 2, pages 162-199, Springer-Verlag LNAI 5402. (2007)
13. Huet, G.: *Sanskrit Segmentation*. In: South Asian Languages Analysis Roundtable XXVIII, Denton, Ohio. (October 2009)
14. Jha, G.N. and Mishra, S.K.: *Semantic Processing in Pāṇini's Kāraka System*. In: Sanskrit Computational Linguistics 1 & 2, pages 239-252, Springer-Verlag LNAI 5402. (2007)
15. Kulkarni, A.P., Shukla, D.: *Sanskrit Morphological Analyser: Some Issues*. In: To appear in Bh.K Festschrift volume by LSI. (2009)

16. Kulkarni, A.P., Kumar, A., Sheeba, V.: *Sanskrit compound paraphrase generator.* In: Proceedings of ICON-2009: 7th International Conference on Natural Language Processing, Macmillan Publishers, India.
17. Mahavira: *Pāṇini as Grammarian (With special reference to compound formation).* Bharatiya Vidya Prakashan [Delhi - Varanasi], India, June 1978.
18. Mittal, V.: *Automatic Sanskrit Segmentizer using Finite State Transducers.* In: Proceeding of Association for Computational Linguistics - Student Research Workshop. (2010)
19. Murty, M.S.: *Sanskrit Compounds-A Philosophical Study.* Chowkhamba Sanskrit Series Office, Varanasi(India), 1974.
20. Pandit Ishvarachandra: *Aṣṭādhyāyī.* Chaukhamba Sanskrit Pratisthan, Delhi, 2004.
21. Prince, A., Smolensky, P.: *Optimality Theory: Constraint Interaction in Generative Grammar.* In: RuCCS Technical Report 2 at Center for Cognitive Science, Rutgers University, Piscataway. (1993)
22. Sarma, E.R.S.: *Maṇikaṇa : A Navya-Nyāya Manual.* The Adyar Library and research Centre, Madras. (1960)
23. Scharf, P.M.: *Levels in Pāṇini's Aṣṭādhyāyī.* In: Sanskrit Computational Linguistics 3, pages 66-77, Springer-Verlag LNAI 5406. (2009)
24. Yuret, D., Biçici, E.: *Modeling Morphologically Rich Languages Using Split Words and Unstructured Dependencies.* In: ACL-IJCNLP, Singapore. (2009)

Appendix-A : Sanskrit Compound Type and sub-type Classification

अव्ययीभावः		कर्मधारयः	
Compound sub-types	Tags	Compound sub-types	Tags
अव्यय-पूर्वपदः	A1	विशेषण-पूर्वपद-कर्मधारयः	K1
अव्यय-उत्तरपदः	A2	विशेषण-उत्तरपद-कर्मधारयः	K2
तिष्टद्गुप्रभृति	A3	विशेषण-उभयपद-कर्मधारयः	K3
संख्यापूर्वपद-नद्युत्तरपद	A4	उपमान-पूर्व-पद-कर्मधारयः	K4
नद्युत्तरपद-अन्यपदार्थे संज्ञायाम्	A5	उपमान-उत्तर-पद-कर्मधारयः	K5
संख्यापूर्वपद-वंश्योत्तरपदः	A6	अवधारणापूर्वपदः-कर्मधारयः	K6
पारे-मध्ये-पूर्वपद षष्ठ्युत्तरपद	A7	सम्भावनापूर्वपद-कर्मधारयः	K7
		मध्यमपदलोपी-कर्मधारयः	Km
तत्पुरुषः		बहुव्रीहिः	
Compound sub-types	Tags	Compound sub-types	Tags
प्रथमातत्पुरुषः	T1		
द्वितीयातत्पुरुषः	T2	द्वितीयार्थ-बहुव्रीहिः(समानाधिकरणः)	Bs2
तृतीयातत्पुरुषः	T3	तृतीयार्थ-बहुव्रीहिः(समानाधिकरणः)	Bs3
चतुर्थीतत्पुरुषः	T4	चतुर्थ्यर्थ-बहुव्रीहिः(समानाधिकरणः)	Bs4
पञ्चमीतत्पुरुषः	T5	पञ्चम्यर्थ-बहुव्रीहिः(समानाधिकरणः)	Bs5
षष्ठीतत्पुरुषः	T6	षष्ठ्यर्थ-बहुव्रीहिः(समानाधिकरणः)	Bs6
सप्तमीतत्पुरुषः	T7	सप्तम्यर्थ-बहुव्रीहिः(समानाधिकरणः)	Bs7
नञ्तत्पुरुषः	Tn	दिग्वाचक-बहुव्रीहिः(समानाधिकरणः)	Bsd
समाहार-द्विगुः	Tds	प्रहरणविषयक-बहुव्रीहिः(समानाधिकरणः)	Bsp
तद्धितार्थद्विगुः	Tdt	ग्रहणविषयक-बहुव्रीहिः(समानाधिकरणः)	Bsg
उत्तरपदद्विगुः	Tdu	अस्त्यर्थ-मध्यमपदलोपी(नञ्)-बहुव्रीहिः	Bsmn
गतिसमासः	Tg	प्रादि-बहुव्रीहिः	Bvp
कुसमासः	Tk	संख्योभयपद-बहुव्रीहिः(समानाधिकरणः)	Bss
प्रादिसमासः	Tp	उपमानपूर्वपद-बहुव्रीहिः(समानाधिकरणः)	Bsu
मयूरव्यंसकादिः	Tm	व्यधिकरण-बहुव्रीहिः	Bv
तत्पुरुषः बहुपदः	Tb	सङ्ख्योत्तरपदः व्यधिकरण-बहुव्रीहिः	Bvs
तत्पुरुषः उपपदः	U	सहपूर्वपद-व्यधिकरण-बहुव्रीहिः	BvS
		उपमानपूर्वपद-व्यधिकरण-बहुव्रीहिः	BvU
		बहुपदः-बहुव्रीहिः	Bb
द्वन्द्वः		केवल	
Compound type:	Tags	Compound type:	Tags
इतरेतरयोग-द्वन्द्वः	Di	केवल समासः	S
समाहार-द्वन्द्वः	Ds	द्विरुक्तिः	
एकशेष	E	द्विरुक्तिः	d

Appendix-B : Rules for Paraphrase Generation.

	अव्ययीभावः
1	<x-y>A1 => y{6} f{x} where f maps x to the noun with same semantic content. A function f needs to be defined.
2	<x-y>A2 => x{3} विपरीतम् वृत्तम्
3	<x-y>A4 => x{6} y{6} समाहारः
4	<x-y>A5 => x'{1} y{1} यस्मिन् देशे x' has same gender as that of y'.
5	<x-y>A6 => x{6} y{6} समाहारः if x = द्वि, both x and y will be in द्विवचनम्
6	<x-y>A7 => x{6} y
	तत्पुरुषः
7	<x-y> T_n => x{n} y $2 \leq n \leq 7$
8	<x-y>Tn => x{n} y
9	<x-y>Tds => x{6;ba} y{6;ba} समाहारः
10	<x-y-z>Tb = x{1} y{1} z{1}
	कर्मधारयः
11	<x-y>K1 => x{1} तत् y{1} च
12	<x-y>K2 => x{1} च y{1} च
13	<x-y>K3 => x{1} च असौ y{1} च
14	<x-y>K4 => x{1} इव y{1}
15	<x-y>K5 => x{1} y{1} इव
16	<x-y>K6 => x{1} एव y{1}
17	<x-y>K7 => x{1} इति y{1}
	बहुव्रीहिः
19	<x-y>Bsd => x{6} च y{6} च यदन्तरालम्
20	<x-y>Bsp => x{3} च y{3} च प्रहृत्य इदम् युद्धम् प्रवृत्तम्
21	<x-y>Bsg => x{7}-y{7}+ गृहीत्वा इदम् युद्धम् प्रवृत्तम्
22	<x-y>Bsmn => x'-y{1} यस्य
23	<x-y>Bss = > x{1} वा y{1} यस्य
24	<x-y>Bsu => x{1} इव y{1} यस्याः
25	<x-y>Bv => x y{1} यस्य
26	<x-y>Bvs => y{6} x' ये सन्ति ते
27	<x-y>BvS => y{3} सह
28	<x-y>BvU => x{6} इव y यस्य
	द्वन्द्वः
29	<x-y>Di => x{1} च y{1} च
30	<x-y>Ds => x{1} च y{1} च
31	<x-y>S => y{1} x{1}
32	<x-y>d => x y