Sanskrit and Computational Linguistics

Selected papers presented at the 16th World Sanskrit Conference
28 June – 2 July 2015, Silpakorn University, Bangkok

Editors
Vineet Chaitanya
Amba Kulkarni

February 17, 2015

Prakriyāpradarśinī - an open source *subanta* generator

Dhaval Patel and Shivakumari Katuri

Abstract: Prakriyāpradarśinī is an attempt to imitate subanta derivation process by prakriyā method given in Siddhāntakaumudī (SK) of Bhaṭṭojī Dīkṣita (1910) using an open source PHP code. Our goal is to imitate SK regarding applicability of rules and give the user step by step derivation. The machine handles strīpratyayaprakaraṇa also.

In theory, there is no fixed order of rules for derivation process in sapādasaptādhyāyī of Aṣṭādhyāyī, but if we analyze SK for practical application of rules, rules are applied in some kind of order. The authors have tried to find out the optimum order of application of rules from Sanskrit NLP perspective and are proposing an 'NLP order model' and 'NLP order hypothesis' for coding subantaprakaraṇa of SK. This is extremely beneficial from coding perspective, because it would decrease the iterations compared to the prevalent 'conflict resolution model' e.g. for a 10 step process, in the 'conflict resolution model' computer will check whether any of 4000 odd sūtras are applicable or not for 10 times and resolve the conflict i.e. >40000 event checking, whereas in 'NLP order model' it would check the criteria for application of sūtras chronologically i.e. only with marginally above 4000 event checking.

The present paper tries to analyze the necessity of user input in subantaprakaraṇa of SK for proper declention. The paper also discusses some of the issues in rule ordering and conflict resolution for Sanskrit NLP from grammatical perspective.

Keywords: Aṣṭādhyāyī, Computational Linguistics, Conflict Resolution, Natural Language Programming, NLP Order Model, NLP Order Hypothesis, Pāṇini, Prakriyā, Prakriyāpradarśinī, Siddhāntakaumudī

1 Introduction

Prakriyāpradarśinī is an attempt to imitate the derivation process given in Siddhāntakaumudī (SK) of Bhaṭṭojī Dīkṣita using an open source PHP code. Our goal is to imitate Siddhāntakumudī in terms of applicability of rules. The main difference between the present approach and earlier approaches for derivation is in the methodology.

The present approach uses 'NLP order model' in contrast to 'Conflict resolution model' employed earlier. The details about this model will be discussed in section 4. This machine also handles *strīpratyaya*s. The other difference is regarding the licence of the code. This code is an open source code which anyone can use and modify according to his specific need. We have spent enough time reinventing the wheel in Sanskrit NLP world. It is high time to move on to an open source world.

2 Review of literature

In all available literature regarding Sanskrit NLP, Goyal et al. (2009) has been found the most relevant to the pursuit at hand, therefore it has been commented upon.

Introduction of that paper mentions that one has to be precise in what one wants to simulate. We have taken SK as base. *Sūtra*s and *vārtika*s which are accepted in SK have been incorporated in the code. Section 2 of the same paper raises an issue that Scharf's (2008) method closely follows SK and not that of Aṣṭādhyāyī (AS). Our present endeavour also is a simulation of SK, but there is kind of a reconciliation of AS method and SK method. The only place where we have some liberty in order is *sapādasaptādhyāyī*[1], whereas order of *tripādī*[2] is more or less unchangeable for any researcher. We have deployed one do-while loop for most of the rules of *sapādasaptādhyāyī*, which continues till the input and output are same i.e. till there is no rule in *sapādasaptādhyāyī* which can apply now. So, it works like AS method, but if the rules are in random order, the rules will loop over several times (at least 5-6 times as per our estimate). This is a heavy cost on the code and server. It is better if we arrange the rules in the method specified in 'NLP order model' which closely follows a *prakriyā* method to economize

[1] AS chapter 1-1 to 8-1
[2] AS chapter 8-2 to 8-4

on time and space. The words 'NLP order model' refer to an alternate order of rules where rules are organized in an order which is more suited for Computational Linguistics related coding as compared to AS order of rules.

While the observation 'AS models generation' is true, we would like to draw attention that sometimes the declension varies according to speaker's intention e.g. whether 'priyatri' would mean *priyāḥ trayaḥ yasya saḥ* or *priyāḥ tisraḥ yasya saḥ* depends on speaker's intention (SK on 6.4.4). The derivation also varies according to it. Therefore, it is mandatory that we take user input on places which we find ambiguous or not amenable to coding, especially in machines which do single word derivation with no context whatsoever.

According to section 4 of Goyal et al. (2009), simulation of Sanskrit grammar involves the following factors. (1) Interpretation of sūtras using the metalanguage described by Pāṇini in the AS, (2) faithful representation of sūtras, (3) automatic triggering of rules and (4) automatic conflict resolution. On these parameters, our machine works as mentioned below.

1. For **interpretation**, we have used explanation offered by SK.

2. *Sūtra*s are **represented** as faithfully as possible. As and when any wrong output is encountered, the code is re-inspected and necessary corrections are made.

3. **Automatic triggering** of rules is done as and when the necessary conditions are satisfied.

4. For **conflict resolution**, *apavāda, parasūtra, aṅgakārya* and alteration in *pratyayas* are given priority, which takes care of majority of conflicts. In case any conflict remains, it also is resolved by altering the order of the rule application. Unlike Pāṇini's structure, computer language codes are almost always executed in a linear sequential manner. Therefore, perhaps the most favoured mode to stop execution of code is to place the code in 'if blocks'. Considering the number of *sūtra*s, the places of conflict are relatively very few, and most of the time, there is ample grammatical literature to resolve the conflicts. So, in our opinion, finding the correct order for computer execution is a possibility within human reach. If, after all possible re-ordering, the conflict still remains, we can add a patch for that particular word. This is the fundamental theory behind restructuring the order of rules in 'NLP order model'.

Section 4 of that paper mentions usage of regular expressions to represent patterns and positions to represent right and left context and alter string by them. Slight modification in the present system is the use of array rather than string. Its advantage is that the array can store multiple strings e.g. it can store both *vāk*, *vāg* by application of *vā'vasāne* (8.4.56) for future manipulation. If we store the output in a string rather than an array and modify that string by rules of grammar, it becomes difficult to handle optional forms. For rule triggering, we have also used regular expressions and two custom made functions 'sub' and 'arr'.

Section 6 of the same paper highlights that rule *sasajuṣo ruḥ* (8.2.66) is an exception to *pūrvatrāsiddham* (8.2.1) metarule and its implication in coding. SK specifically mentions that *rutva* is not *asiddha* to some rules which require *rutva* as triggering event[3]. Therefore, we have placed *sasajuṣo ruḥ* (8.2.66) at two places. First place is before the application of these rules. Second is its usual *tripādī* place. To prevent re-application of the *sūtra*, we have remembered that once the rule has been applied. We check while applying for second time whether the rule has been applied already earlier. If yes, we do not apply it again, otherwise the rule in *tripādī* is applied.

Section 7 of the same paper points out a question regarding repeat application of *yāḍāpaḥ* (7.3.113) cyclically in *ramā* + *ṅe* because of *ṅittva* of *pratyaya*. To circumvent this cyclical application of rules (when the rule should apply only once), in the do-while loop we increase $start by one, every time the code makes a loop. In the condition for the rule triggering, we ask whether variable $start is equal to 1 or not. If it is 1, then only the rule applies. Otherwise, it does not apply.

Scharf (2009) has evaluated conflict resolution in AS with four different principles and concluded that 'determining which rule has precedence in the shared domains is not reducible to a single principle'. Therefore, we have adopted the conflict resolution explanations given in SK and coded according to it. Cardona (2009) has analysed the principle of *pūrvatrāsiddham* and its allowable exceptions in grammar. It is mainly in tandem with what traditional grammar texts offer. We have coded properly for it as far as *subanta* generation is concerned.

With this background in mind, let us proceed with the paper.

[3] See SK on ato roraplutādaplute (6.1.113), SK part 1 page 99.

3 Overview of the project

The project aims at creating an open source PHP code[4] which would derive noun forms of a given word step by step according to SK. At places, we have also adopted the explanation offered by SK and displayed it to the user to make the derivation easier to understand. But, in some cases, SK explains why a particular rule is not applicable. If it places some constraint on code, we have left out that explanation in display.

The process within the code is in SLP1 transliteration for ease of coding because it assigns a single letter to all *Devanāgarī* characters as has been mentioned by Hyman (2009). This minimizes ambiguity. For example, **प्रउग** and **प्रौढ** both would have 'prau' in their IAST / HK transliteration, whereas SLP1 for them would be 'prauga' and 'prOQa'. This also eases out transliteration to *Devanāgarī* and other encodings.

User can enter the words in IAST, SLP1 or *Devanāgarī* transliteration. A peculiar problem arises when the user enters a *halanta* word in *Devanāgarī*. Because of variety of input methods, there are sometimes associated white spaces with *halanta* marker. So, before processing, one has to remove the white spaces first with help of code. The *sūtra* display to user is bilingual in Howard Kyoto protocol and *Devanāgarī*. The word under derivation is shown in *Devanāgarī*. At later stage, if need be, this can also be shown in SLP1, IAST or HK or any other transliteration method.

If the machine needs to know some additional parameters based on the word given, ajax.php is called and it gives additional fields in the front end for user to fill. Then ultimately subanta.php is fired and the output is displayed to user.

We have followed the following style pattern, so that the display conveys additional information regarding the *sūtra*: Red colour for headings and error messages, gray colour for *vidhi sūtra*s or *apavāda sūtra*s, green colour for *paribhāṣā*s and *sañjñāvidhāyaka sūtra*s and yellow colour for explanatory notes. Thus, the user can get the information regarding *sūtra* type also from the display without much difficulty.

In Sanskrit grammar, there are certain words which are *nityadvivacana* or *nityabahuvacana* or whose derivation is same in all three *liṅga*s. We have listed some of such words as and when they occur in SK and display the user the information if some *pratyaya* can not apply to this word.

[4] `https://github.com/drdhaval2785/SanskritSubanta`

Certain fonts do not display *upadhmānīya* or *jihvāmūlīya* properly. Therefore, for proper display we have chosen Siddhanta[5] font as our default font. We have used '!' to denote anusvāra. A brief note regarding some special characters used in code can be seen here[6].

Sometimes, we had to do things not mandated by SK to accommodate user tendency. For example, we have observed that the users usually enter *visarga* instead of *sakāra* at the end. We have accepted that user behavior and modified the code to give back the *sakāra* in *prakriyā*.

Discussion in this paper would not be according to the order of *sūtra*s in SK or AS, but according to the sequence of code subanta.php[7] in which *sūtras* are applied in this machine.

4 NLP order model and NLP order hypothesis

Though Pāṇini's rule order is treated as very strict, in our opinion there is a possibility of re-ordering them for ease of computational linguistics. We propose the 'NLP order model' and suggest an alternative rule order for *subantaprakaraṇa*[8].

As the *sūtra*s have a kind of grouping based on similarity or conflict, there is some free space in which the *sūtra* order can be moved up / down in machine. Based on our experience, we put forward this '**NLP order hypothesis**' for deciding rule ordering as per '**NLP order model**' for Computational linguistics.

Let us suppose that *sūtra*s are in the order

A1, A2, ..., A(k), .., A(n)

The **Range of freedom** which the *sūtra* A(k) enjoys in term of moving it upwards or downwards in an algorithm for application to input string is equal to the **range (A(min), A(max))**, where **A(min)** is the last previous *sūtra* which can alter the input string for A(k) and make A(k) inapplicable for any possible word in Pāṇini's grammar. **A(max)** is the first next *sūtra* until which no other *sūtra* can alter the output string of A(k) for

[5] http://www.svayambhava.org/ www.svayambhava.org/

[6] https://github.com/drdhaval2785/SanskritSubanta/blob/master/encoding_notes.txt/

[7] https://github.com/drdhaval2785/SanskritSubanta/blob/master/subanta.php/

[8] https://github.com/drdhaval2785/SanskritSubanta/blob/master/rule_order.txt/

any possible word of Pāṇini's grammar. So, it will be possible to decide the proper position of most of *sūtra*s for computational linguistics with this range in future. If there are still some places where such rule ordering gives wrong results, patches may have to be applied.

Let us clarify how we came to such a conclusion with a working example.

If we have a look at the derivation of *rāma* word in SK, the following *sūtra*s are important.

rāma + ṅe -

svaujasamauṭchaṣṭābhyāmbhisṅebhyāmbhyasṅasibhyāmbhyasṅasosāmṅyossup (15), *ṅeryaḥ* (240) and *supi ca* (291). The numbers in bracket indicate their position in our code based on 'NLP order model'.

Upto this point, we have taught the machine that the order is *svaujasamauṭchaṣṭābhyāmbhisṅebhyāmbhyasṅasibhyāmbhyasṅasosāmṅyossup* (15), *ṅeryaḥ* (240) and then *supi ca* (291). At this point the range of *sūtra ṅeryaḥ* is (15,291).

When we move forward in the declension and reach *rāma + bhyas*, there is a possibility of application of *bahuvacane jhalyet* (290). Now as a coder, we will have to decide where we should put this rule. At this juncture, we can see that *supi ca* (7.3.102) is the rule which can alter the output string (*rāma+bhyas* -> *rāmā+bhyas*) and render *bahuvacane jhalyet* inapplicable (no *akāra* at end). Therefore, we can not place *bahuvacane jhalyet* after *supi ca*, otherwise, *bahuvacane jhalyet* will see an altered string (*rāmā+bhyas*) and it can not apply. Therefore, we have to put it just before *supi ca* i.e. at place 290. Thus, our code sequence will be *svaujasamauṭchaṣṭābhyāmbhisṅebhyāmbhyasṅasibhyāmbhyasṅasosāmṅyossup* (15), *ṅeryaḥ* (240), *bahuvacane jhalyet* (290) and *supi ca* (291). The lower limit upto which this code can be shifted is *svaujasamauṭchaṣṭābhyāmbhisṅebhyāmbhyasṅasibhyāmbhyasṅasosāmṅyossup* (15). Thus we have a range for this *sūtra ṅeryaḥ* as (15,290). As the code progresses, the interval gets shortened.

Let us take the derivation of '*tad*' *pulliṅga*, to see how the range gets minimized as the code progresses.

In this code the relevant rules for our discussion are *tyadādīnāmaḥ* (7.2.102), *ato guṇe* (6.1.97) and *bahuvacane jhalyet* (7.3.103). When we want to place *tyadādīnāmaḥ* at its proper place, it has to be before *ato guṇe*, which also should be before *bahuvacane jhalyet.* Therefore their *inter se* applicability would be *tyadādīnāmaḥ* -> *ato guṇe* -> *bahuvacane jhalyet.* Out of which *tyadādīnāmaḥ* and *ato guṇe* are treated in a single piece of code. Therefore they are applied at specified place let's say 191th place looking at other exigencies. As is obvious, they should be placed before *bahuvacane jhalyet*, because otherwise the word *tad* + *bhyas* woould not have conditions suitable for *bahuvacane jhalyet* to apply (it would have *tad* + *bhyas* instead of expected *ta* + *bhyas*). So, the range of *sūtra ṅeryaḥ* has become (191,290). Similarly the range *bahuvacane jhalyet* has become (191,291) instead of earlier (15,291). In this way, the location of *sūtra* becomes more and more restricted as the code advances. This way we can keep on playing with the location of *sūtra*s as SK advances. It is beyond our mathematical capacities, but we suggest that if the sequences of application of rules in *prakriyā grantha*s like SK are evaluated mathematically, near perfect rule order with least possible iteration loops may be derived mathematically.

If at any given point, there is a difficulty in identifying proper location of a *sūtra*, and alteration in position gives erroneous forms, we can create a patch (some code to sort the issue out) for the same and overcome it. But in most of the *sūtra*s in *subantaprakaraṇa*, it was possible to find their place properly[9].

5 Overview of the code

There are mainly 6 files in our code – ajax requirements.docx, ajax.php, function.php, mystyle.css, script.js, subanta.html and subanta.php. We shall now examine the salient features of the code used in this machine.

5.1 subanta.html, ajax requirements.docx, ajax.php and script.js

Because we are treating only the *subantaprakaraṇa* of SK (SK pp. 110-326), it is not possible that machine knows all other *sūtra*s of AS. Even otherwise, there are cases when the word declension depends on user's intention (*vivakṣā*). Therefore, it is not possible that a machine alone can give us desired declension without user input. We have tried to make machine responsive to the word entered by user. It shows appropriate radio buttons to gather

[9] Till the paper for *subanta* generation was redrafted, the code has progressed beyond *strīpratyayaprakaraṇa* and 80% of *tiṅantaprakaraṇa*. This 'NLP order model' still held good. So, the suggestion made earlier has been substantiated as the code advanced..

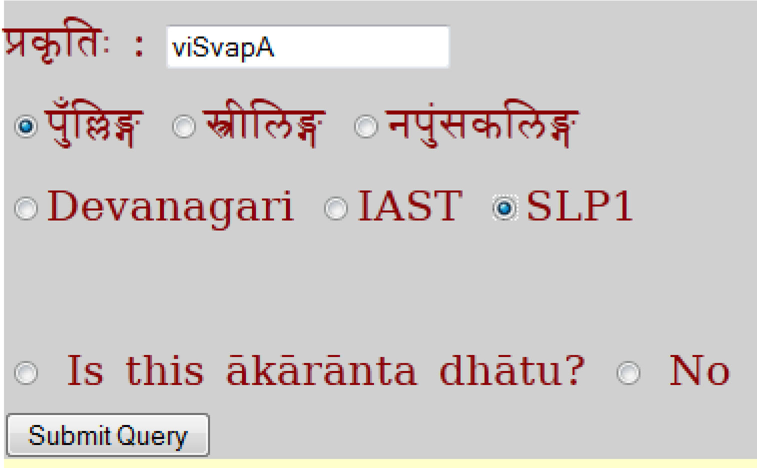

Figure 1
User input window

additional information from user, if needed for our purpose. ajax requirements.docx, ajax.php and script.js are the scripts responsible for seeking the input of user.

Let us clarify this with an example. If a user enters a word with '*ā*' at the end and wants to decline it in masculine gender, we need to get the information whether this is *ākārānta dhātu* or not. To get this information, we take user input via ajax when a word ending in '*ā*' is entered and masculine gender is selected as shown in Figure 1. For description of methodology, please see section 11. The detailed list of cases where we take user input are available on our website[10].

At this point let us clarify about two types of user input which have been deployed. The first type is where the input is taken, because the declension depends on the intention of user. An example of this case would be whether '*sarva*' is used as *sañjñā* or not. This is non-negotiable type of input, in the sense that even in future we would not be able to do away with them. We

[10] `https://github.com/drdhaval2785/SanskritSubanta/blob/master/ajax\%20requirement.docx/`

have written 'no' in the list in this file[11] for this type of feedback. The second type is where we do not know (as of now) how to decide some parameter e.g. right now we are asking the user whether the word is *ābanta* or not. In future, when the machine learns how to differentiate a word having *ābanta*, we will no longer need this input. In that sense, this type is negotiable. As the machine progresses, these input points can be removed. We have noted down the second type of input with a 'yes' and a note on how we can remove them in future. Veracity or otherwise of this list is open to suggestion.

5.2 function.php

Pāṇinian grammar works on many *sūtra*s which are called for execution as and when a condition is satisfied. We have devised some functions based on those *sūtra*s for repetitive work e.g. functions 'prat' (*pratyāhāra*), savarna, vriddhi, guna, dirgha, ṭi, mit etc. This file[12] also holds the data sets e.g. vowels, consonants etc in addition to the functions.

5.3 slp-dev.php and dev-slp.php

They are transliterator codes which convert SLP1 transliteration to *Devanāgarī* and vice versa. These codes are borrowed and modified from Dicrunch code of Ananda Loponen[13].

5.4 subanta.php

This is the code which actually processes the entered word and shows the result back to the user. The most intricate part about automatic declension machine have always been rule triggering, conflict resolution and ordering of *sūtra*s. The first two issues will be dealt with at their respective places. The third is a bit lengthy, so its concept is discussed in the section of rule ordering. Details of rule order are available on our website[14].

---

[11] `https://github.com/drdhaval2785/SanskritSubanta/blob/master/user_input.pdf`

[12] `https://github.com/drdhaval2785/SanskritSubanta/blob/master/function.php/`

[13] `https://github.com/drdhaval2785/sanskrit/tree/master/diCrunch/`

[14] `https://github.com/drdhaval2785/SanskritSubanta/blob/master/rule_order.txt/`

This code was first developed as a *sandhi* generator, and later on merged with *subanta* generator. Therefore, coding for all *sūtra*s mentioned in *sandhiprakaraṇa* is kept intact within this *subanta* generation machine. Most of them are even used for derivation processes also. User may note some *pragṛhya* related portion which is not relevant to *subanta* generation directly in the code, but we have retained it from the legacy of *sandhi* generator.

For most of the code we have retained a '+' sign between *prakṛti* and *pratyaya*. But when it comes to *dvitvaprakaraṇa* in *tripādī*, '+' sign creates some disturbance in the function because of a coding issue which we could not overcome. So, for now we have removed + sign before *dvitvaprakaraṇa*. There are not many *sūtra*s after *dvitvaprakaraṇa*, which need the identification of *pada* and *pratyaya*. Therefore, there is not much information which is lost by removing this '+' sign.

khari ca (8.4.55) is a cyclic process till all possible combinations are over. So, we have kept a while loop till there is no member satisfying the condition. Let us clarify it with an example.

e.g. *suhṛd* + *sup* => *suhṛdd* + *su* (by *anaci ca* (8.4.47))=> *suhṛdt* + *su* (by *khari ca* (8.4.55)).

At this stage, there still is a '*d*' preceding '*t*'. Therefore, *khari ca* (8.4.55) finds its application once again. So, we have kept a while loop till there is no member satisfying the condition. By doing this, we could ensure that '*d*'-> '*t*' transition can still take place, and gives *suhṛttsu*.

There is some gray area regarding application of *paribhāṣā parjanyavallakṣaṇapravṛttiḥ* (*pa* 119)[15]. If there is a combination of *car* + *khar* letter, should *khari ca* (8.4.55) apply, because '*car*' is itself a subset of '*jhal*'? Though there is no difference in the form, the rule must apply because of *parjanyavallakṣaṇapravṛttiḥ* (*pa* 119) *paribhāṣā*, because the conditions for application of rule are satisfied. Should we display such cases or not is yet to be determined, but, anyway, the code is mature enough to handle both the choices.

6 Variables

AS uses variables very effectively in its structure. Several *sañjñā*s are assigned to the word and they are made use of at a later stage e.g. '*sarva*' gets

[15] The paribhāṣā numbers refer to paribhāṣenduśekhara (1913)

sarvanāma-sañjñā by *sarvādīni sarvanāmāni* (1.1.27) and AS uses them at places like *sarvanāmnaḥ smai* (7.1.14). In coding parlance, its close approximation is something like this:

if (input word = *sarva*) { $sarvanama=1; }

This assigns $sarvanama value of 1 like Pāṇini assigns it *sarvanāma sañjñā*.

if ($sarvanama===1) { Do *sarvanāmnaḥ smai* (7.1.14) }.

This checks whether $sarvanama is equal to 1 or not, and executes the code. It is similar to application of *sarvanāmnaḥ smai* (7.1.14) in case the word has *sarvanāma sañjñā* in Pāṇinian system.

Thus, variables play very crucial role in the simulation of Pāṇini's grammar. We have enumerated some of the variables which we have used in our code so that the reader may have a bird's view about what is happening in the code.

Examples of variables used in the code are: *sup*, *pada*, *bha*, input word, gender, transliteration, *nadī*, *ṅī*, *ābanta*, *taddhita*, *dhātu* etc. All the variables can be seen in function.php and subanta.php. Their explanation and importance are given in the code itself as and when they are applied for the first time. Unless specified otherwise, the meaning of different values are: 0 - no application, 1 - mandatory application and 2 - optional application.

Variables are used for two purpose in our code: (1) to remember that some rule **has been applied** e.g. $Ap=1 would mean that the word is derived from some *āp pratyaya* (that rule has already applied) and (2) to remember that some rule **has to be applied** e.g. $sarvafinal=1 would mean that all rules specific to *sarvanāma*s have to be applied. We have chosen the variable names close to the corresponding grammatical notation so that the it is easy to understand the code.

7 Rule Triggering

There are specific *prakriyā*s to be followed in grammar when specific conditions are satisfied. Therefore, appropriate rule triggering is of utmost importance for success of the code. We have used inbuilt functions of PHP, syntax of PHP, operators and some user defined functions to check whether the conditions for application of a rule are met or not.

8 Sample Code

We will show a sample code here along with its explanation so that the user may get the feel of what is happening in the background for fetching the required output.

8.1 Rule triggering code

```
/* ṅeryaḥ (7.1.13) */
    if (arr($text,'/[a][+][ṅ][e]/') && $pada=== "pratyaya" && $so ===
"ṅe" )
    {
    $text = one(array("a+ṅe"),array("a+ya"),0);
    echo "<p class = sa >By ṅeryaḥ (7.1.13) :</p>";
    echo "<p class = sa >     ( . .  ) :</p>";
    display(3);
    $ṅe=1; // 0 - This sūtra has not applied. 1 - This sūtra has been applied.
    }
```

8.2 Rule triggering explanation

1. if (arr($text,'/[a][+][ṅ][e]/') && $pada=== "*pratyaya*" && $so === "*ṅe*")

In this section we check the following conditions – the suffix ($so) is '*ṅe*', *akāra* is followed by *ṅe* and *ṅe* is a *pratyaya*. When these conditions are satisfied, the rest of the code is executed.

2. { }

The bracketed area is code which is to be executed.

3. $text = one(array("*a+ṅe*"),array("*a+ya*"),0);

In this section, we convert '*a+ṅe*' to '*a+ya*' i.e. we apply *ṅeryaḥ sūtra.*

4. echo "<p class = sa >By *ṅeryaḥ* (7.1.13) :</p>";

echo "<p class = sa > (. .) :</p>";

We display the *sūtra* which has been applied in this case.

5. display(3);

We display the word to user. (In *Devanāgarī*)

6. $ṅe =1;

We remember that the *ṅeryaḥ* has been applied to this word, for future use in code.

9 Conflict Resolution

We have used usual Pāṇinian methods like *apavāda*, *parasūtra* etc. for conflict resolutions as discussed in SK. Grammar books and their commentaries provide plenty of literature on conflict resolution. There are many *paribhāṣā*s also. The most important *paribhāṣā* dealing with conflict resolution is *paranityāntaraṅgāpavādānāṃ uttarottaraṃ balīyaḥ* (*pa* 38).

This has been taken care of implicitly in rule ordering itself. We have tried to place *parasūtra*, *nitya prakriyā*s, *antaraṅga prakriyā*s and *apavādasūtra*s before *pūrvasūtra*, *anitya prakriyā*s, *bahiraṅga prakriyā*s and *utsargasūtra*s respectively. If there is conflict among *para*, *nitya*, *antaraṅga*, *apavāda*, the later wins. Such encounter has not happened in *subanta* generation stage. Sometimes *paribhāṣā*s are *nitya* / *anitya*. Sometimes *pūrvavipratiṣedha*[16] applies e.g. *numaciratṛjvadbhāvebhyo nuṭ pūṛvavipratiṣedhena* (*vā* 4374). We have used *pūrvavipratiṣedha* whenever it is explicitly mentioned in the text of SK. Thus, generic application of metarules is not possible. So they will be applied in specific cases only. In addition, there are places where grammarians have difference of opinion. In such cases we have taken SK as authority[17]. Whatever has been accepted in SK is imitated in the code. If SK is silent on some topic, other available commentaries are explored for solution.

9.1 Methods to avoid application of a rule

1. Ordering *apavāda*, *parasūtra*, *antaraṅga*, *nitya*, *aṅgavidhi*, *pratyaya* alteration rules before the *utsarga*, *pūrvasūtra*, *bahiraṅga*, *anitya* and other rules. This way, the later group sees a modified string which does not satisfy criteria for their application.

2. Remembering that a rule has to be applied in future e.g. variable $purvapara=1 means that the rule *pūrvaparāvaradakṣiṇottarāparādharāṇi vyavasthāyāmasaṃjñāyām* (1.1.33) will apply later on. When the turn of this rule comes we check whether variable $purvapara is equal to 1 or not.

[16] *pūrvavipratiṣedha* means processes where the preceding *sūtra* is given priority over subsequent *sūtra* , violating the general rule.

[17] See SK part 1 page 161 under the rule *trestrayaḥ* 7.1.53

3. Remembering that a rule is not to be applied in future e.g. if *ato'm* (7.1.24) has been applied, we store the value of variable $atom as 1 and when conditions of application of *svamornapuṃsakāt* (7.1.23) are tested, we tell it not to apply the *sūtra* to words where *ato'm* (7.1.24) has been applied.

9.2 Notes on issues in conflict resolution

With this background, let us examine some of the issues which cropped up during process of simulating *subantaprakaraṇa*:

1. *ārambhasāmarthya.*

   supi ca (7.3.102) is *parasūtra*compared to *nāmi* (6.4.3). Even then, it does not apply in case of *nāmi* (6.4.3) even though it is *parasūtra* because of *ārambhasāmarthya.* So, due care needs to be taken in coding such cases according to the explanation given in grammar texts.

2. *anityatva* of *paribhāṣā*s.

   For example, *sannipātaparibhāṣā* does not apply in case of *ṅe pratyaya. supi ca* (7.3.102) applies in that case. *kaṣṭāya kramaṇe* (3.1.14) is an example of *anityatva*[18] of this *paribhāṣā.* If we code for such *paribhāṣā*s to apply in every case, this form will get distorted. The better alternative is to apply such *paribhāṣā*s only in cases where SK has validated its applicability.

3. *sasajuṣo ruḥ* and treatment of *sakāra.*

   Following code blocks should be placed before *sasajuṣo ruḥ* (8.2.66) for execution of code, otherwise their ultimate *sakāra* may take *sasajuṣo ruḥ* (8.2.66) to give undesired forms. - 1. *etattadoḥ sulopo'ko'nañsamāse hali* (6.1.132), 2. *so'ci lope cetpādapūraṇam* (6.1.134), 3. *aniditāṃ hala upadhāyāḥ kṅiti* (6.4.24) for *sraṃs, dvaṃs* etc 4. *vasusraṃsudhvaṃsvanaḍuhāṃ daḥ* (8.2.72). These code blocks have to be kept before actual execution of *sasajuṣo ruḥ* (8.2.66).

4. *ṇatva.*

[18] This means the same rule applies with some condition at some place, but does not apply at some other place even if the same condition is satisfied.

It is difficult to identify where *pada* ends and another starts, especially in *samāsa*s. Therefore, the conflict resolution among *sūtra*s *ekājuttarapade ṇaḥ* (8.4.12) / *raṣābhyāṃ no ṇaḥ samānapade* (8.4.1) / *aṭkupvāṅnumvyavāye'pi* (8.4.2)is extremely difficult until *samāsa* is parsed effectively.

10 Notes on some grammar issues and their coding implications.

This section is devoted to brief mention of issues we encountered during development of this software, which may serve as a reference point to the future researchers in case they face the same difficulty in implementation.

1. Do-while loop for *sapādasaptādhyāyī*.

   pūrvatrāsiddham (8.2.1) *sūtra* creates two separate data spaces for *sūtra*s of AS namely (1) *sapādasaptādhyāyī* and (2) *tripādī*. There are two other *sūtra*s which are also used to create separation of data spaces like *asiddhavad atrābhāt* (6.4.22) and *ṣatvatukorasiddhaḥ* (6.1.86)[19]. We have not treated these two *sūtra*s here, because they are not explicitly treated in *subantaprakaraṇa* of SK. So we will keep our discussion focused on *pūrvatrāsiddham* (8.2.1) only[20]. To put it it very basic terms, *pūrvatrāsiddham* (8.2.1) makes provision that the rules in *sapādasaptādhyāyī* have no fixed order of application and those of *tripādī* have to be applied after all possible *sapādasaptādhyāyī* rules have applied to the word. Even inside *tripādī*, the rules are to be applied sequentially. For *sapādasaptādhyāyī*, we have created a do-while loop which checks whether the word entering and word coming out of this loop is the same or not. It continues till both are the same (i.e. until there is no rule in *sapādasaptādhyāyī* which can apply and alter the word). In most of the cases, no looping of *sapādasaptādhyāyī* is needed. In other cases, only one looping of *sapādasaptādhyāyī* is needed. This is quite low burden on code as compared to checking 4000 rules every time. After that, it moves on to *tripādī*, where the

[19] See Goyal et al. (2009) section 3.6

[20] 'na lopaH supsvarasajhjjhātugvidhiṣu' is dealt with in the code at places where it is applicable.

rules are usually written in sequence of their appearance in AS, so that the rules apply sequentially.

In this approach, there is a limitation. There are certain rules in *sapādasaptādhyāyī*, which can apply only once. For such rules, we have applied them before do-while loop or we have specified in do-while loop that this rule is to be applied when the code is being executed for the first time only by specifying $start=1. When the code comes for execution for second time, we check whether $start===1. In the second loop $start has value of 2 and therefore, this rule is not applied for second time. This is, in a nutshell, what we do to ensure secrecy of data space of *sapādasaptādhyāyī* and *tripādī.*

2. *asiddhatva* of *ṣakāra* for applicability of *sasajuṣo ruḥ* (8.2.66).

   There are two possible cases (1) *dhātu*s - *pipaṭhiṣ, āśiṣ.* (2) not *dhātu*s - *doṣ, dhanuṣ.* The usual user tendency is to enter *ṣakāra* at the end of such words. If we proceed with this word, there is no *sakāra* at the end of the word, and therefore *sasajuṣo ruḥ* (8.2.66) can not apply. To circumvent this, *ṣakāra* has to be converted back to *sakāra* for *sasajuṣo ruḥ* (8.2.66)'s application. Similar situation appears in case of *vivakṣ, didhakṣ, pipakṣ* etc. *skoḥ saṃyogādyorante ca* (8.2.29) does not apply because *kutva* is *asiddha* to *kalopa.* Similarly, in case of *cikīrṣ* - *ṣatva* is *asiddha* to *rātsasya* (8.2.24). Therefore, *rātsasya* (8.2.24) sees *cikīrs* only and not *cikīrṣ.* Thus, it causes elision of last *ṣakāra.* For such typical cases, patches have to be applied in the code.

3. All *pratyaya* alterations have to be completed before applying *sūtra*s which remove *it* marker, otherwise it would not be possible to check the exact *pratyaya* e.g. *rāma*+*ṅe* -> *rāma*+ya by *ṅeryaḥ* (7.1.13) *sūtra.* If we had removed the *it* marker 'ṅ' before application of *ṅeryaḥ*, the situation would have been *rāma*+e and we would have tough time finding out whether this is a *ṅit pratyaya* or not.

4. attributes of stem and suffix.

   It is of vital importance to remember the attributes of stem and suffix e.g. *sarvanāma, sarvanāmasthāna, it* markers, previous application of some *sūtra, bhāṣitapuṃskatva, tṛjvadbhāva,* original input word, *sambuddhi, ṣaṭ, nadī sañjñā* etc. Some of them are discussed here in brief. These attributes may be absent, present or optionally present in grammar. Because of these three types, it is not advisable to use boolean variables for attributes, as they cater to only presence and absence.

Status of *sarvanāma* - It is important to know whether the word is a *sarvanāma*, not a *sarvanāma* or optionally a *sarvanāma*. With the help of *sarvādi gaṇa* and some user input, we decide whether $sarvafinal = 0 (not *sarvanāma*), 1 (*sarvanāma*) or 2 (optionally *sarvanāma*). Sometimes, we need to add a member at a later stage in a *gaṇa*. e.g. *'sva'* is pronoun only if it is used not in the meaning of *jñāti* or *dhana*. If we add *'sva'* directly into *sarvanāma* set, it will give erroneous results if it is used in sense of *jñāti* or *dhana*.

it markers play an important role in derivation process e.g. *aco ñṇiti* (7.2.115) will apply only when the *pratyaya* has '*ñ*' or '*ṇ*' as *it* marker. Similarly there are many rules where we have to know about *it* marker.

It is wise to remember the first input word. This may be needed after some time e.g. *hali lopaḥ* (7.2.113) mandates elision of '*id*' of '*idam*' when certain conditions are met. Till this stage, '*idam*' is already converted to '*ida*' by *tyadādīnāmaḥ* (7.2.102) and *ato guṇe* (6.1.96). Therefore we need to check whether this '*ida*' was derived from '*idam*' or not.

sambuddhi forms are different than regular forms. Therefore, it is mandatory to remember whether the pratyaya is *sambuddhi* or not. e.g. *eṅhrasvātsambuddheḥ* (6.1.69) will only apply in case of *sambuddhi*.

5. *stoḥ ścunā ścuḥ* (8.4.40) and *śāt* (8.4.44).

   This case is different form of expression than regular ABC-> ADC context based transformation. Therefore, special treatment is needed. In this case, no specific order of letters is mandated. This rule applies in case of juxtaposition rather than order. Therefore, a separate code is needed to handle this rule. Similarly *ṣṭunā ṣṭuḥ* (8.4.41) and *na padāntāṭṭoraṇām* (8.4.42), *anāmnavatinagarīṇāmiti vācyam* (*vā* 5016) and *toḥ ṣiḥ* (8.4.43) need specific treatment.

6. *na lumatāṅgasya* (1.1.63).

This rule prevents conversion of *padānta kim* to *ka*. Explanation of SK is given a place in derivation scheme. *na lumatāṅgasya* (1.1.63) also bars application of *pratyayalope pratyayalakṣaṇam* (1.1.62) - so we have to remember whether luk has happened or not. We should also remember that *na lumatāṅgasya* (1.1.63) is *anitya paribhāṣā*. It does not apply in case of *tricaturoḥ striyāṃ tisṛcatasṛ* (7.2.99)[21].

7. *sthānivadbhāva.*

   There is a great deal of literature on what is *sthānivadbhāva* and what is not in SK. Therefore, we have not treated *sthānivadbhāva* generically. We have coded according to it only when the derivation demands such intervention to be made.

8. *nimittāpāye naimittikasyāpāyaḥ.*

   When doing elision by *saṃyogāntasya lopaḥ* (8.2.23), we have to be ready for application of *nimittāpāye naimittikasyāpāyaḥ*.

9. Aberrant behaviour of rules.

   vṛddhyauttvatṛjvadbhāvaguṇebhyo num pūrvavipratiṣedhena (*vā* 4373) mandates that the *numāgama* gets precedence over rules mentioned here by *pūrvavipratiṣedha*. Similarly, *numaciratṛjvadbhāvebhyo nuṭ pūṛvavipratiṣedhena* (*vā* 4374) mandates that *nuḍāgama* gets precedence over rules mentioned here by *pūrvavipratiṣedha*. These rules are exception to the general precedence of *parasūtra*.

   Sometimes grammarians try to justify the derivation of a *śiṣṭa* word by '*akṛtavyūhāḥ pāṇinīyāḥ*'.. An example of it can be seen in application of *acaḥ* (6.4.138).

10. Difference of opinion amongst grammarians. There is a difference of opinion among grammarians regarding *prarīṇām*[22] and *varṣābhū*. We have accepted SK's position.

11. Difficulty in coding.

    rutva can happen any time in between code. Therefore, *upadeśe'janunāsika it* (1.3.2) which elides *ukāra* of '*su!*' might not

[21] See pp. 196 of SK part 1.

[22] See SK part 1 p. 220

work properly. Therefore, a special patch is made for *rutva* to convert it to *repha*.

ḍhralope pūrvasya dīrgho'ṇaḥ (6.3.111). It is mandatory to remember the *repha* / *ḍhakāra* where this is to be applied. Otherwise, all the *hrasva* + *r/ḍh* combinations in the word would be converted to *dīrgha* + *r/ḍh*. For this purpose, we have added an artificial sign # before *repha* or *ḍhakāra* where this rule has to be applied.

bhāṣitapuṃskatva is very difficult to know from words, and many *prakriyā*s depend on whether the word is *bhāṣitapuṃska* or not. This calls for user input.

In case of verbs, users may adopt different conventions for writing a verb. For example, user may insert *ancu*, *añcu, anc, añc, ancu!, añcu!* anything. Such behavior is seen more frequently in case of verbs with anubandhas and especially with verbs having a nasal letter in it. So, the code has to be resilient enough to account for such aberrant behavior of users.

Keeping *ato guṇe* (6.1.97) applicable to each *sūtra* creates many issues like interfering with *akaḥ savarṇe dīrghaḥ* (6.1.101). Right now, *ato guṇe* (6.1.97) is used in close conjunction with the rules where grammar textbook mandates it and not as a separate code block.

11 User input

By the words **user input**, we mean 'getting desired input from user for correct declension of a word'. For a good code, this has to be at bare minimum to enhance user experience. So we have decided what user input fields are negotiable ones and which are not in this file[23]. For user input, we use radio buttons. In future, if there is a case where we must check multiple parameters simultaneously, multiple check boxes can also be employed. The documentation for user input is stored in ajax requirements.docx e.g. if ajax.php uses $_GET['cond1_2']===2, it means that condition 1.2.2 in the document is satisfied. The user can easily make out from code and ajax requirements.docx what we check out in user input.

[23] `https://github.com/drdhaval2785/SanskritSubanta/blob/master/user_input.pdf`

12 Limitations

After discussion on methodology followed by us, it would be of interest if we place before the researchers some problem areas which we encountered, so that they may be explored further with grammar texts and solutions may be arrived at.

1. Difficulty in identifying attributes. Correct derivation depends on correct identification of attributes, but it becomes extremely difficult to identify those attributes in some cases. In such cases, user input may be our only hope.

   nityastrīliṅgatva has to be taken as user input, because it is difficult to guess *nityastrīliṅgatva* by merely looking at the word. Some detailed analysis of feminine words may help in this regard.

   iyaṅuvaṅsthānatva[24] presupposes the knowledge whether the word meets the criteria for application of '*iyaṅ*' or '*uvaṅ*'. It is difficult to analyze this beforehand.

   There is a perpetual problem whether (1) one should retain "*ṅi*", "*ṅe*" etc till the *prakriyā*s with *ṅittva* start or (2) should we convert them to '*i*', '*e*' and remember that it has '*ṅ*' as it. Right now, the former method is used preferably. *upasarjanībhūtatva / pradhānatva* – It is very important to know these qualities in *samāsa*s. Right now, we are not able to analyse *samāsa*s, so we have taken user input in this case.

   As *kvin / kvip pratyaya*s do not leave any mark on the word, they are difficult to identify. Right now, more or less we are listing out *kvinnanta* and *kvibanta* words manually, which may not be workable in long run. We will have to think of some alternative to overcome this problem.

   It is difficult to differentiate *ābantatva* or *ākārānta dhātu* from merely looking at the word. We will have to understand *ābanta pratyaya*s as well as *dhātu prakriyā*s to code properly for them.

   abhyastatva. It is difficult to code for *abhyastatva*, till *abhyāsa prakriyā* is taught.

---

[24] AS 1.4.4

2. Problems with *dhātu*s. It is difficult to identify *dhātu*s. Even if we enlist all *dhātu*s in *dhātu*pāṭha, there are *nāmadhātu*s and *sanādi dhātu*s too, which make it difficult to identify where *kṛdatiṅ* (3.1.93) is to be applied. Right now, we are looking for '*īy*' to identify *nāmadhātu*s. e.g. *eḍakīyati*. It may need further revision when *nāmadhātu*s are taught to the system.

   One needs to find all possible *dhātu*s starting with "ṛ" to decide whether *upasargādṛti dhātau* (6.1.91) is applicable or not. Even then, special treatment for *nāmadhātu*s is needed because the rule is optional for *nāmadhātu*s.

   It is equally difficult to identify *prātipadika*s, because there are many *pratyaya*s which may derive a new noun from a *dhātu*. Some of them do not even leave a mark morphologically like *kvip* / *kvin* etc. Therefore, it is really difficult to identify *prātipadika*s and separate them from *dhātu*s.

3. Issues of morphologic similarity.

   Sometimes the *prakriyā* is specified for word ending with some word e.g. if a *prakriyā* for 'ahan' is specified and we search for string 'ahan' only, the *prakriyā* for '*sūryāhan*' may not work well. Therefore, we have to think about any morphological change which the word might undergo under the influence of rules of *sandhi* too.

   As we work with string of letters in coding, it is difficult to isolate words ending with 'han' for applicability of *inhanpūṣāryamṇāṃ śau* (6.4.12). 'han' at the end of a word can also be part of 'ahan', where this rule would be erroneously applicable. User input or exhaustive enumeration will be needed for clarity.

4. Issues of *ekādeśa*.

   ādyantavadekasmin (1.1.21) rule is difficult to code, because right now we are keeping a '+' sign in between the stem and suffix. It is difficult in current scheme of things to code properly to remember that the *ekādeśa* behaves as end of the previous one and the start of the next one. Another question which deserves attention is how should *ekādeśa* be displayed? e.g. *ādguṇaḥ* (6.1.87) mandates *ekādeśa*. What should we display in case of '*rāma+i*'? '*rām+e*' or '*rāme+*' ? Let us show our approach with example. In case of *ādguṇaḥ* (6.1.87), the term '*āt*' means that the *ādeśa* is after *akāra*. So, we have kept it 'rām+e', whereas *akaḥ savarṇe dīrghaḥ* (6.1.101) mandates replacement of 'ak'. So, we have kept 'rāmā+t'.

5. Issues in contextual derivation. As we are not working with sentences for now, it is difficult to analyse attributes which depend on sentences e.g. whether there is *anvādeśa* or not in case of derivation of *asmad* / *yuṣmad*. Currently only words are being treated and not sentences, so *pāda* related functions are not applied for now.

6. Different derivation in different meanings. e.g. the word *sudhī* can be analysed as *suṣṭhū dhīryasyāḥ*, *suṣṭhu dhyāyati*, *suṣhṭhu dhīḥ*. The derivation differs in all the situations. Therefore it is mandatory to take user input to specify which of these meanings he intends to use.

7. Listings. Various lists (over and above *gaṇa*s) are needed for proper declention of a word e.g. *ugit dhātu*s, *idit dhātu*s, *ṝkārānta* words etc. Exhaustive lists remain to be made for such words.

13 Scope for Future work

1. We have left out accents e.g. *caturanaḍuhorāmudāttaḥ* (7.1.98) – we have coded only for '*ām*' and left out '*udāttaḥ*'. We will have to treat the accent at a later stage for sure, because the *strīpratyaya*s and *taddhitapratyaya*s have very peculiar effect on accent, otherwise morphologically *ṅīp*, *ṅīṣ*, *ṅIn* give the same forms.

2. The *sūtra*s which we have not coded for are specifically mentioned in code subanta.php. The user is advised to refer to them for further details.

3. *sūtra*s which involve interpretation of *samāsa*s are right now on user input mode. Once *samāsa* interpretation is taught to the machine, they can be properly coded.

4. We have prepared a list with a hint whether the requirement for user input can be done away with or not[25]. This can serve as a guide for future researchers. The future attempts should be primarily focused on removing the unnecessary user input from the machine. Once this is achieved, researchers can take up the more challenging task of handling the non negotiable kind of user input.

14 Conclusion

AS has a complex system of interrelated rules. Various authors have tried in past to reorganize the order of AS for *prakriyāgrantha*s. Similarly there is a need to reorder the AS for easy implementation of computational simulation of AS. NLP order model and NLP order hypothesis presented in the present work is a step in that direction.

[25] https://github.com/drdhaval2785/SanskritSubanta/blob/master/user_input.pdf

References

Bhatta, Nagesha. 1913. *Vaidyanāthakritagadatikāsamvalitaḥ Paribhāṣenduśekharaḥ*. Ed. by Hari Narayan Apte. Anandāśram Press.

Cardona, George. 2009. "Pūrvatrāsiddham and āśrayāt siddham." In: *Studies in Sanskrit Grammar*. Ed. by George Cardona, Ashok Aklujkar, and Hideyo Ogawa. DK Printworld, pp. 123–62.

Dīkṣita, Bhaṭṭojī. 1910. *Siddhāntakaumudī of śri Bhaṭṭojī Dīkṣita with the commentary śri Bālamanoramā of śri Vāsudeva Dīkṣita*. S. Candrasekhara Sastrigal, Teppakulam.

Goyal, Pawan, Amba Kulkarni, and Laxmidhar Behra. 2009. "Computer Simulation of *Aṣṭadhyāyī*: Some Insights." In: *Sanskrit Computational Linguistics 1 & 2*. Ed. by Gérard Huet, Amba Kulkarni, and Peter Scharf. Springer-Verlag LNAI 5402, pp. 139–61.

Hyman, Malcolm. 2009. "From Pāṇinian Sandhi to Finite State Calculus." In: *Sanskrit Computational Linguistics 1 & 2*. Ed. by Gérard Huet, Amba Kulkarni, and Peter Scharf. Springer-Verlag LNAI 5402, pp. 253–65.

Scharf, Peter M. 2008. "Modeling Pāṇinian Grammar." In: *Sanskrit Computational Linguistics 1 & 2*. Ed. by Gérard Huet, Amba Kulkarni, and Peter Scharf. Springer-Verlag LNAI 5402, pp. 95–126.

—. 2009. "Rule Selection in Aṣṭādhyāyī' or Is Pāṇini's Grammar Mechanistic?" In: *Studies in Sanskrit Grammar*. Ed. by George Cardona, Ashok Aklujkar, and Hideyo Ogawa. DK Printworld, pp. 319–50.

Appendix : Sample derivation of ramā

Attached below is the derivation of '*ramā*' word [26]

[26] as on 27.9.2014 from `http://lanover.com/lan/sanskrit/subanta.php?first=ramA&gender=f&tran=Devanagari&cond2_1=2&step=2_1_2`

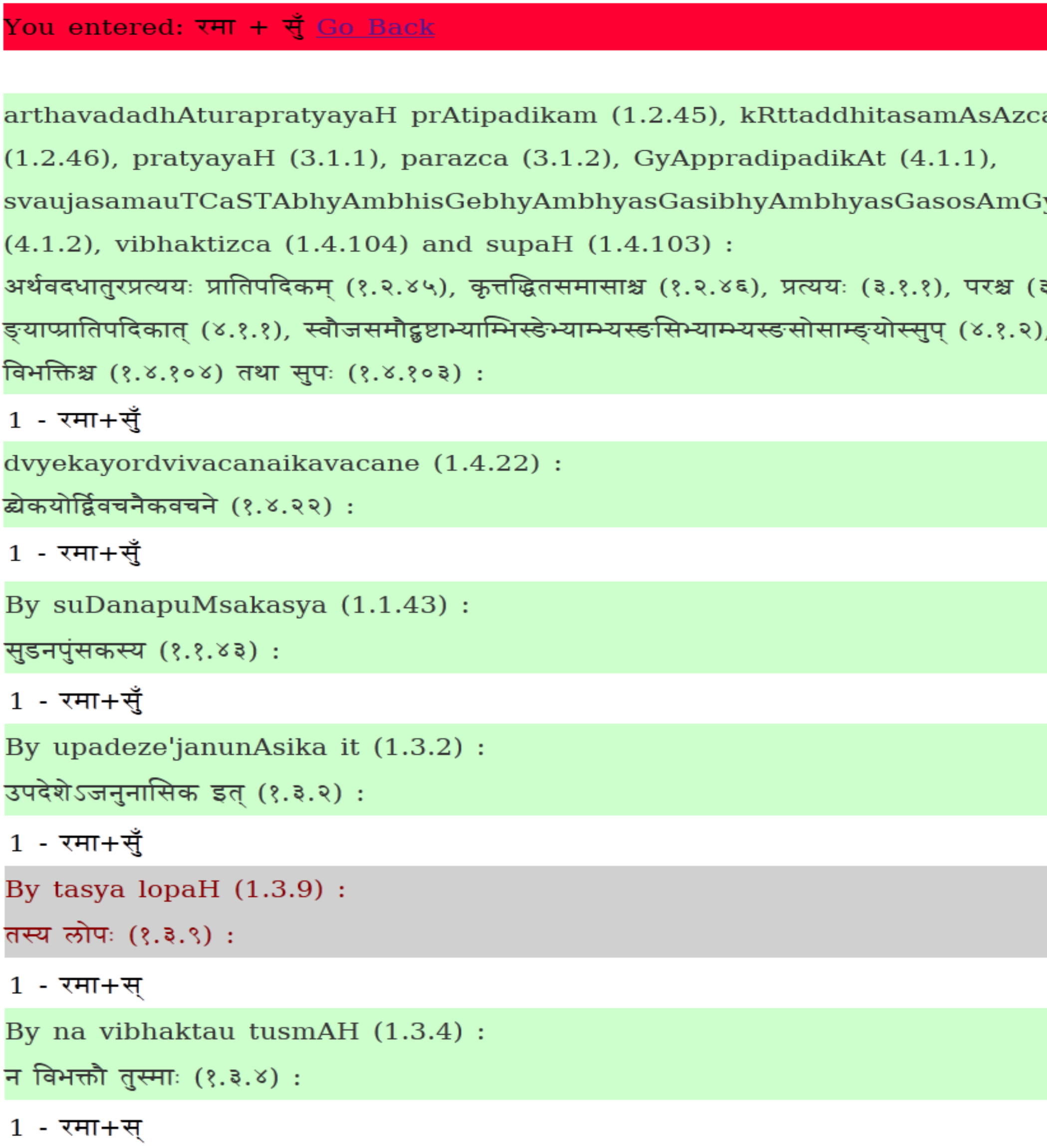

Figure 2
Sample Derivation: Part 1 of 2

By apRkta ekAlpratyayaH (1.2.41) :

अपृक्त एकाल्प्रत्ययः (१.२.४१) :

1 - रमा+स्

By halGyAbbhyo dIrghAtsutisyapRktaM hal (6.1.68) :

हल्ङ्याब्भ्यो दीर्घात्सुतिस्यपृक्तं हल् (६.१.६८) :

1 - रमा+

By aNo'pragRhyasyAnunAsikaH (8.4.57) :

अणोऽप्रगृह्यस्यानुनासिकः (८.३.५७) :

1 - रमा

2 - रमाँ

Final forms are :

आखिरी रूप हैं :

1 - रमा

2 - रमाँ

Figure 3

Sample Derivation: Part 2 of 2