vyakarana Documentation

Release 0.1

Arun Prasad

June 01, 2014
# Contents

1 Background  
1.1 Introduction  ......................................................... 3  
1.2 Rule Types .............................................................. 4  
1.3 Terms and Data .......................................................... 6  
1.4 Sounds ................................................................. 8  
1.5 *asiddha* and *asiddhavat* .......................................... 9  
1.6 Glossary ............................................................... 10  

2 Architecture  
2.1 Design Overview ...................................................... 13  
2.2 Inputs and Outputs ..................................................... 14  
2.3 Modeling Rules .......................................................... 14  
2.4 Selecting Rules .......................................................... 17  
2.5 Defining Rules ........................................................... 17  

3 API Reference  
3.1 API ........................................................................ 19  

Python Module Index ......................................................... 29
This is the documentation for Vyakarana, a program that derives Sanskrit words. To get the most out of the documentation, you should have a working knowledge of Sanskrit.

**Important:** All data handled by the system is represented in SLP1. SLP1 also uses the following symbols:

- `'\'` to indicate *anudatta*
- `'^'` to indicate *svarita*
- `'~'` to indicate a nasal sound

Unmarked vowels are *udita.*
CHAPTER 1

Background

This is a high-level overview of the Ashtadhyayi and how it works.

1.1 Introduction

This program has two goals:

1. To generate the entire set of forms allowed by the Ashtadhyayi without over- or under-generating.

2. To do so while staying true to the spirit of the Ashtadhyayi.

Goal 1 is straightforward, but the “under-generating” is subtle. For some inputs, the Ashtadhyayi can yield multiple results; ideally, we should be able to generate all of them.

Goal 2 is more vague. I want to create a program that defines and chooses its rules using the same mechanisms used by the Ashtadhyayi.

In other words, I want to create a full simulation of the Ashtadhyayi.

1.1.1 The Ashtadhyayi

The Ashtadhyayi (Adhyyi) is a list of about 4000 rules. It has ordinary rules, which take some input and yield some output(s), and metarules, which describe how to interpret other rules. If Sanskrit grammar is a factory, then its ordinary rules are the machines inside and its metarules are the instructions used to build the machines.

Given some input, the Ashtadhyayi applies a rule that changes the input in some way. The output of the rule is then sent to another rule, just as items on the assembly line move from one machine to the other. This continues until there’s no way to change the result any further. When this occurs, the process is complete. The result is a correct Sanskrit expression.

This documentation makes reference to various rules from the Ashtadhyayi. All rules are numbered x.y.z, where:

- x is the book that contains the rule. There are 8 books in total.
- y is the chapter that contains the rule. Each book has 4 chapters.
- z is the rule’s position within the chapter.

For example, 1.1.1 is the first rule of the text, and 8.4.68 is the last.
1.1.2 The Dhatupatha

If the Ashtadhyayi is the stuff inside the factory, then the Dhatupatha (Dhtupha) is the raw material that enters the factory. It is a list of about 2000 verb roots, each stated with a basic meaning:

1.1 bh sattrym
   bh in the sense of existence (satt)

Modern editions of the Dhatupatha are numbered x.y, where:

- x is the root’s verb class (gaa). There are 10 classes in total.
- y is the root’s position within the gaa.

Thus bh is entry 1 in gaa 1; it’s the first root in the list.

There is no single version of the Dhtupha. I used a version I found on Sanskrit Documents (specifically, this file) and made some small corrections. So far, it’s been totally competent for the task.

1.2 Rule Types

The Ashtadhyayi has ordinary rules, which take some input and yield some output(s), and metarules, which describe how to interpret other rules.

Note: The types loosely correspond to the traditional classification, but there is no 1:1 mapping.

1.2.1 Ordinary rules

Ordinary rules, or just “rules” for short, are the bulk of the Ashtadhyayi. These rules accept a list of terms as input, where a term is some group of sounds. For example, the input to a rule might be something like ca + k + a. Outputs have the same form.

There are various kinds of ordinary rules;

- rules that substitute
- rules that designate
- rules that insert
- rules that block

These are described below.

Substituting

Most rules substitute one term for another. They look something like this:

C is replaced by X (when L comes before C) (when C comes before R).

Here, L, C, R, and X are terms:

- L is the left context and appears immediately before C. Not all rules use it.
- R is the right context and appears immediately after C. Not all rules use it.
- C is the center context. It defines where the substitution occurs.
- X is the replacement. It defines the new value for C.
For each input, we look for a place where we have \( L \), \( C \), and \( R \) in order. Then we replace \( C \) with \( X \).

For example, rule 6.1.77 of the Ashtadhyayi states that simple vowels (or \( ik \), if we use a pratyhra) are replaced by semivowels (\( ya \)) when followed by other vowels (\( ac \)). Given this input:

\[
ca + k + a
\]

we have a match when \( C = \) and \( R = a \). (\( L \) is unspecified, so we ignore it.) We replace with \( X = r \) to get our output:

\[
ca + k + a \rightarrow ca + kr + a
\]

**Designating**

Some rules designate a term by assigning some name to it. They look something like this:

\[
C \text{ is called } X \text{ (when } L \text{ comes before } C) \text{ (when } C \text{ comes before } R).
\]

where \( X \) is the name given to the center context \( C \).

For example, rule 1.3.1 states that items in the Dhatupatha are called \( dhtu \) (“root”) Given this input:

\[
bh
\]

we have a match where \( C = bh \), with \( L \) and \( R \) unspecified. We then give \( bh \) the name “dhtu.” In other words, \( bh \) is a \( dhtu \).

**Inserting**

Of the rules left, most insert:

\[
X \text{ is inserted after } L \text{ (when } L \text{ comes before } R).
\]

For example, rule 3.1.68 states that \( a \) is inserted after a verb root when the root is followed by a certain kind of verb ending. Given this input:

\[
car + ti
\]

we have a match where \( L = car \) and \( R = ti \). So, we insert \( X = a \) to get our output:

\[
car + ti \rightarrow car + a + ti
\]

**Blocking**

Some rules are used to block other rules from occurring:

\[
C \text{ does not accept rule } X \text{ (when } L \text{ comes before } C) \text{ (when } C \text{ comes before } R).
\]

For example, rule 1.1.5 blocks \( gua \) substitution if the right context has a certain property.

**Other rules**

A few rules are combinations of the ones above. For example, rule 3.1.80 inserts one term then performs a substitution on another.
1.2.2 Metarules

Metarules define the metalanguage used by the Ashtadhyayi. Since we’re using our own metalanguage (Python), many of these metarules are modeled implicitly.

There are basically two kinds of metarules:

• rules that help us interpret other rules
• rules that provide useful context for other rules

These are described below.

Interpreting

Most metarules are intended to help us understand what rules in the Ashtadhyayi mean. Such rules are called paribh. Some examples:

Terms in case 6 define the center context. (1.1.49)
Terms in case 7 (tasmin) define the right context. (1.1.66)
Terms in case 5 (tasmt) define the left context. (1.1.67)
If X is just a single letter, then only the last letter of C is replaced. (1.1.52)

Contextualizing

All other metarules provide some extra context for other rules. Such rules are called adhikra. Some examples:

In the rules below, all inserted terms are called pratyaya. (3.1.1)
In the rules below, L and R together are replaced by X. (6.1.84)

1.3 Terms and Data

The rules of the Ashtadhyayi accept a list of terms as input and produce a new list of terms as output. Let’s start by discussing what terms are and what information they contain.

Throughout this section, our working example will be ca + k + a, a sequence of three terms. Depending on the data attached to these terms, this sequence can yield a variety of outputs:

• cakra ("he/I did", perfect tense)
• cakara ("I did", perfect tense)
• cakra ("he did", perfect tense)

1.3.1 Sounds

Our example has three terms, each of which represents a piece of sound. These “pieces of sound” usually represent morphemes, but that’s not always the case.

We’ll have more to say about these sounds later, but for now they’re pretty straightforward.
### 1.3.2 Sajñ

Each term has a variety of designations (sajñ) associated with it. These sajñ, which are assigned by the Ashtadhyayi itself, enable some rules and block others. By assigning names to different terms and changing which rules can be used, the system can guide the original input toward the desired output.

Our example uses the following sajñ:

<table>
<thead>
<tr>
<th>ca</th>
<th>k</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>abhyasa</td>
<td>dhtu</td>
<td>prayaya</td>
</tr>
<tr>
<td>_</td>
<td>_</td>
<td>vibhakti</td>
</tr>
<tr>
<td>_</td>
<td>_</td>
<td>ti</td>
</tr>
<tr>
<td>_</td>
<td>_</td>
<td>rdhadhuka</td>
</tr>
</tbody>
</table>

In addition, $ca + k$ together are called both *abhya* and *aga*.

Some examples of what these sajñ do:

- *dhtu* allows the rule that creates the *abhya*.
- *abhya* allows a rule that changes *ka* to *ca*.
- *rdhadhuka* allows a rule that strengthens the vowel of the term before it.

### 1.3.3 it tags

Terms also use a second set of designations, which we can call *it* tags. Just a shirt might have a label that tells us how to wash it, a term might have an *it* that tells us how it behaves in certain contexts.

For example, $k$ has two *it* tags. The first is *u*, and it allows $k$ to take a certain suffix. The second is *ñ*, and it allows $k$ to use both *parasmaipada* and *manepada* endings in its verbs. *It* tags are attached directly to the term of interest, like so: $ukñ$

We can remove *it* tags by applying some metarules. For some term T, the following are *it* tags:

- nasal vowels (1.3.2)
- at the end of T:
  - consonants (1.3.3)
    - but not \{t, th, d, dh, n, s, m\} when T is a *vibhakti* (1.3.4)
- at the beginning of T:
  - *ñi*, *u*, and *u* (1.3.5)
- at the beginning of T, if T is a *pratyaya*:
  - (1.3.6)
  - *c, ch, j, jh, n, h, h* (1.3.7)
  - *l, ś, k, kh, g, gh*, if not a *taddhita* suffix

*It* tags are not letters in any meaningful sense, and they have no meaning outside of the metalanguage of the Ashtadhyayi. In other words, all they do is describe certain properties; they have no deeper linguistic meaning and are not a fundamental part of Sanskrit. So if you see a term like $ukñ$, you should read it as:

$k$ with the *it* tags $u$ and $ñ$.

The *it* tags are often stated with the word *it* after them. Thus *vit* and *ñit*. A term stated with its *it* letters is called the *upadeśa* of the term. Thus $ukñ$ is the *upadeśa* of the root $k$. 

---

**1.3. Terms and Data**  

7
Usage

*it* tags are basically just *sajñ* that are expressed more tersely.

To illustrate how alike these two are, let’s return to our *ca* + *k* + *a* example. We saw above that this sequence can yield three different results. But the result depends on the *sajñ* and *it* tags applied to the suffix *a*. As you read on, note how the different *sajñ* and *it* tags interact.

- If the *upadeśa* is just *a*, then rule 1.2.5 tags the suffix with *kit*. This prevents *gua*. After a few more rules, we get *cakra* for our result.
- If the *upadeśa* is *al*, the suffix has *it*, which causes *vddhi*. After a few more rules, we get *cakra* for our result.
- If the *upadeśa* is *al*, the suffix has *it*. But if the suffix has *uttama* as a *sajñ* – that is, if it is in the first person – then *it* is used only optionally. If we reject *it*, then the *rdhadhtuka-sajñ* causes *gua*. After a few more rules, we get *cakara* for our result.

The glossary describes the most common *it* tags and some of the roles they perform. Many *it* tags are overloaded to provide a variety of different functions.

1.4 Sounds

Sandhi is an important part of Sanskrit. Thus sandhi is an important part of the Ashtadhyayi. The metalanguage of the Ashtadhyayi gives us a few ways to describe different groups of sounds as tersely as possible.

1.4.1 Savara sets

First, a way to describe related sounds:

\[
\text{Vowels and semivowels, as well as consonants with } \underline{u} \text{ as an } \underline{it} \text{ letter, refer to all } \text{savara} \text{ (“homogeneous”) terms. (1.1.69)}
\]

*Savara* has a precise definition, but generally it refers to sounds that are similar in some way. Anyway, some examples:

- *a* refers to *a*
- *i* refers to *i*
- *ku* refers to all sounds in *kavarga*
- *cu* refers to all sounds in *cavarga*

*a* and *i* also refer to the corresponding nasal vowels, but generally we can ignore the nasal sounds entirely. (The rule mentions semivowels because some semivowels can be nasal, too.)

1.4.2 Single vowels

In the grammar, *a* always refers to both *a* and . To refer to just the sound *a*, we use the following rule:

\[
\text{A vowel stated with } \underline{t} \text{ refers to just that vowel. (1.1.70)}
\]

Some examples:

- *at* refers to just *a*
- *t* refers to just

These terms refer to nasal sounds too, but generally we can ignore the nasal sounds entirely.
1.4.3 Pratyhra

Finally, a way to refer to other groups of interest. Consider the following list:

1. a i u
2. k
3. e o
4. ai au c
5. ha ya va ra
6. la
7. ŋa ma a a na m
8. jha bha ŋ
9. gha ha dha
10. ja ba ga a da š
11. kha pha cha ha tha ca a ta v
12. ka pa y
13. śa a sa r
14. ha l

These rows are usually called the Shiva Sutras. They were arranged deliberately so that similar sounds would appear next to each other.

Here’s how we use the list. Each row has a list of sounds that ends with an it tag. We take advantage of the following metarule:

In lists like the one above, an item stated with an it refers to all the items between them, too. (1.1.71)

and use it to produce concise terms for various Sanskrit sounds.

For example, the ha on row 5, when used with it letter l on row 14, creates the term hal. And this hal refers to all sounds between ha and that it letter l. That is, it refers to the set of Sanskrit consonants.

Such groups are called pratyhra. Other examples:

- ac refers to all vowels. By rule 1.1.69, a refers to , and so on for the other vowels.
- khar refers to all unvoiced consonants.
- ya refers to all semivowels.
- al refers to all sounds.

Certain sounds and it letters are used in the list twice, but context is enough to tell us how to interpret a given pratyhra.

1.5 asiddha and asiddhavat

When a rule applies to some input to yield some output, the input is discarded and all future applications act on the output. But sometimes the original input preserves some information that we want to keep.
1.5.1 asiddha

TODO

1.5.2 asiddhavat

Consider the following input:

\[
\text{\textit{\textit{\textit{\textit{\textit{\textit{\textit{\textit{s}s + hi}}}}}}}}
\]

By 6.4.35, \textit{s}s becomes \textit{s} when followed by \textit{hi}. By 6.4.101, \textit{hi} becomes \textit{dhi} when preceded by a consonant. If one applies, the other is blocked. But to get the correct form \textit{\textit{\textit{\textit{\textit{\textit{\textit{s}dhi}}}}}}}, we have to apply both rules together.

The Ashtadhyayi solves this problem by placing both rules in a section called \textit{asiddhavat}. For any two rules A and B within this section, the results of A are invisible to B (or “as if not completed”, i.e. \textit{a-siddha-vat}). This allows each rule to act without being blocked by the other.

In practical terms, this means that each term has at least two values simultaneously: one accessible only to the non-\textit{asiddhavat} world (e.g. \textit{s}) and one accessible only to the \textit{asiddhavat} world (\textit{s}s).

To see how the program handles these problems, see the data spaces stuff in Inputs and Outputs.

Note: Issues of \textit{asiddha} and \textit{asiddhavat} are subtle and outside the scope of this documentation. Those interested might see rule 6.4.22 of the Ashtadhyayi or section 3.5 of Goyal et al.

1.6 Glossary

1.6.1 Sanskrit

Generally, these are used to describe concepts from the grammatical tradition.

\textit{aga} _

\textit{anubandha} See \textit{it}.

\textit{abhysa} If a term is doubled, \textit{abhysa} refers to the first part.

\textit{abhyasta} If a term is doubled, \textit{abhyasta} refers to the two parts together.

\textit{tmanepada} The last 9 ti suffixes.

\textit{rdhadhtuka} Refers to certain kinds of verb suffixes.

\textit{Adhyy, Ashtadhyayi} A list of rules. It takes some input and produces one or more valid Sanskrit expressions.

\textit{it} An indicatory letter.

\textit{upadeśa} A term stated with its indicatory letters (\textit{it}).

\textit{gua} An operation that strengthens a vowel to the “medium” level (\textit{a}, \textit{e}, \textit{o}, but \textit{and} become \textit{ar}). Also refers to the result of this operation.

\textit{vddhi} An operation that strengthens a vowel to the “strong” level (\textit{ai}, \textit{au}, but \textit{and} become \textit{r}). Also refers to the result of this operation.

\textit{ti} Refers to one of the 18 basic verb suffixes: 9 in \textit{parasmaipada} and 9 in \textit{tmanepada}.

\textit{dhtu} A verb root.

\textit{Dhtupha, Dhatupatha} A list of verb roots. These roots are used as input to the Ashtadhyayi.
parasmaipada  The first 9 ti suffixes.
pratyaya  A suffix.
vibhakti  A triplet of noun/verb endings. Also, an ending within that triplet.
sajñi  A technical name that is assigned to a group of terms. For example, pratyaya is a sajñi for the set of all suffixes.
sravadhtuka  Refers to certain kinds of verb suffixes. Generally, ti and sit suffixes receive this sajñi.
sthn  In a substitution, the term where the substitution occurs.

1.6.2 English

Generally, these are used to describe concepts in the program.

base filter  A filter defined in an inherit() decorator. It is “and”-ed with all of the rule tuples created by the decorated function.
center context  The term that undergoes substitution. In a sajñi rule: the term that receives the sajñi.
fILTER  A callable object that is used to test for a certain context. For details, see the Filter class.
left context  The term(s) that appear immediately before the center context. If no center context is defined: the term(s) after which something is inserted.
metarule  A rule that defines part of the metalanguage of the Ashtadhyayi. Some are explicitly stated, but many are implicit.
operator  A callable object that is used to apply an operation to a state. For details, see the Operator class.
ordinary rule  A rule that takes some input and produces some output(s). In this documentation, such rules are usually just called “rules.”
right context  The term(s) that appear immediately after the center context. If no center context is defined: the term(s) before which something is inserted.
rule tuple  A special shorthand for specifying rules of the Ashtadhyayi. This must be expanded into a full Rule definition before it can be used.

1.6.3 it tags

kit  Prevents gua and vddhi. If a replacement is marked with k, it is added to the end of the sthn.
it  Prevents gua and vddhi. If a replacement is marked with , it replaces the last letter of the sthn.
ñit  Causes vddhi for certain vowels.
it  If a replacement is marked with , it is added to the beginning of the sthn. If a lakra is marked with , then it undergoes some basic rules, e.g. replacement of ths with se.
it  Causes vddhi for certain vowels.
pit  Causes anudtta accent on a pratyaya. A sravadhtuka suffix not marked by p is treated as it.
mit  If a replacement is marked with m, it is inserted after the last vowel of the sthn.
sit  If a replacement is marked with ś, it replaces the entire sthn. Generally, a pratyaya marked with ś can be called sravadhtuka.
This describes the overall architecture of the system.

2.1 Design Overview

2.1.1 Philosophy

As much as possible, the program follows the principles of the Ashtadhyayi. It makes use of almost all of its technical devices, and many of its methods and classes have 1:1 correspondence to particular concepts from the grammatical tradition. This is the case for a few reasons:

- We can model a system that’s well-known and (fairly) easy to understand.
- We can take advantage of the tradition’s prior work.
- We can make it easier to prove certain properties of the system.

The program’s performance is currently just OK, but only a few parts of it use any kind of optimization. With more aggressive caching it can probably run respectably, but if it stays bad (and if those problems are due to language features), I will probably port it to Scala or some other statically-typed functional language.

2.1.2 How the program works

We pass a single input to `ashtadhyayi.Ashtadhyayi.derive()`, the most interesting method in the `Ashtadhyayi` class. This input is stored on an internal stack. As long as the stack is non-empty, we:

1. Pop an input off of the stack.
2. Find all rules such that that:
   - the rule has space to apply to the input
   - if applied, the rule would yield at least one new result.

   Instead of applying these rules simultaneously, we apply just one then repeat the loop.
3. Pick the rule from (2) with highest rank. If no rules were found in (2), send the input to the `asiddha` module and yield the results.

   **Note:** The `asiddha` module is basically legacy code. Currently it’s too complicated to model easily, but in the future it will be modeled like the rest of the system.
4. Apply the rule and push the results back onto the stack.

In other words, the main function of interest is a generator that loops over a stack and yields finished sequences.

The following pages explore elements of this process in detail. In particular:

• what inputs and outputs look like (Inputs and Outputs)
• determining whether a rule has “space to apply” (Modeling Rules)
• ranking rules (Selecting Rules)
• defining rules tersely (Defining Rules)

2.2 Inputs and Outputs

With rare exception, all data handled by the system is processed functionally. That is, every operation applied to an input must create a new input, without exception. The program follows this principle for two reasons:

• branching. Since one input can produce multiple outputs, it’s easier to just create new outputs and ensure that no implicit information can be propagated.
• basic sanity. This makes the system easier to model mentally.

2.2.1 Terms

A rule accepts a list of terms as input and returns the same as output. A term is an arbitrary piece of sound and usually represents a morphere, but that’s not always the case.

In the Ashtadhyayi, these terms are usually called upadeśa, since the grammar is taught (upadiṣyate) by means of these terms. And in the program, these terms are usually represented by instances of the Upadesha class. These classes provide some nice methods for accessing and modifying various parts of the term. For details, see the documentation on the Upadesha class.

Data spaces

As mentioned earlier, terms in the Ashtadhyayi often contain multiple values at once. Within the program, these are modeled by data spaces, which make it easier to access and manipulate these values. These data spaces are basically just tuples; instead of containing a single data value, each term contains a variety of values that are valid simultaneously.

TODO

2.2.2 States

A State is a list of terms. Like the other inputs used by the grammar, states are modified functionally. For details, see the documentation on the State class.

2.3 Modeling Rules

As a reminder, this is how ordinary rules are usually structured:

• C is replaced by X (when L comes before C) (when C comes before R).
• C is called X (when L comes before C) (when C comes before R).
• X is inserted after L (when L comes before R).
• C does not accept rule Y (when L comes before C) (when C comes before X).

We can rewrite these templates into a more general form:

When we see some context window $W$, perform some operation $O$.

where $W$ is an arbitrary set of contexts and $O$ is an abstraction for some arbitrary change, such as:

• replacing $C$ with $X$
• calling $C$ by the name of $X$
• inserting $X$ after $L$
• blocking rule $Y$ on $C$

With this general form in mind, we can decompose a rule model into two parts:

• matching a context. To do so, we use filters.
• applying an operation. To do so, we use operators.

Or in other words: filters test and operators transform.

2.3.1 Filters

A Filter is a callable object that accepts a state and index, performs some test on state[index], and returns True or False as appropriate. For example, the samjna filter returns whether or not state[index] has some particular samjna.

If all of a rule’s filters return True, then the rule has scope to apply.

In older version of the code base, filters were functions that accepted an Upadesha and returned True or False. This approach changed for two reasons:

• A few filters require global access to the state. If they accept just a single term, there’s no way to get information on the rest of the state. So filters were changed to accept state-index pairs.

• Usually, a rule’s filter is a combination of two other filters. One nice way to do this is to use Python’s unary operators (e.g. &, |). But custom operators are supported only for class instances. So filters were changed to class instances.

Parameterized filters

Parameterized filters group filters into families and make it easier to create a lot of related filters. Specifically, they are classes that can be instantiated (parameterized) by passing arguments.

For example, the al class tests whether a term has a particular final letter:

```
ac = al('ac')
ak = al('ak')
hal = al('hal')
```

Note: Parameterized filters have lowercase names for historical reasons. Also, they better match the names for unparameterized filters, e.g. al('i') & ~samyogapurva.
Combining filters

We can create new filters by using Python’s unary operators.

We can invert a filter (“not”):

```python
# ekac: having one vowel
anekac = ~ekac
```

take the intersection of two filters (“and”):

```python
# samyoga: ending in a conjunct consonant
# samjna(‘dhatu’): having ‘dhatu’ samjna
samyoga_dhatu = samyoga & samjna(‘dhatu’)
```

and take the union of two filters (“or”):

```python
# raw(‘Snu’): raw value is the ‘nu’ of e.g. ‘sunute’, ‘Apnuvanti’
# samjna(‘dhatu’): having ‘dhatu’ samjna
# raw(‘BrU’): raw value is ‘BrU’
snu_dhatu_bhru = raw(‘Snu’) | samjna(‘dhatu’) | raw(‘BrU’)
```

2.3.2 Operators

An Operator is a callable object that accepts a state and index, performs some operation, and returns the result. For example, the guna operator applies guna to state[index] and returns a new state.

Parameterized operators

*Parameterized operators* group operators into families and make it easier to create a lot of related operators. Specifically, they are classes that can be instantiated (parameterized) by passing arguments.

For example, the al_tasya class does arbitrary letter substitution:

```python
# ku h: k, kh, g, gh, , h
# cu: c, ch, j, jh, ñ
kuhos_cu = al_tasya(‘ku h’, ‘cu’)
```

```python
# f: ,
# at: a
ur_at = al_tasya(‘f’, ‘at’)
```

Note: Parameterized operators have lowercase names for historical reasons. Also, they better match the names for unparameterized operators.
2.4 Selecting Rules

2.4.1 Rank

2.4.2 Conflict resolution

2.5 Defining Rules

The machinery behind a given rule is often complex and complicated. But by abstracting away the right things, we can greatly reduce the code required per rule, often to just one line in length.

2.5.1 Rule tuples

A rule tuple is a 5-tuple containing the following elements:

1. the rule name, e.g. ‘6.4.77’
2. the left context
3. the center context
4. the right context
5. the operator to apply

These tuples contain the essential information needed to create a full rule, but they are often underspecified in various ways. Some examples:

- A context can take the value True, which means that the rule should use the context defined for the previous rule.
- A context can take the value None, which means that it uses the base filter (see below).
- A context can be an arbitrary string. All contexts are post-processed with auto(), which converts them into actual Filter objects.
- An operator can be an arbitrary object, usually a string. The program usually does a good job of transforming these “operator strings” into actual Operator objects. For example, if the operator is just ‘Nit’, the program recognizes that this is an it and that the rule is assigning a saññ.

Rule tuples are usually contained in RuleTuple objects, but most rules are just stated as tuples.

Some example rule tuples, from throughout the program:

# Analogous extension of it
('1.2.4', None, f('sarvadhatuka') & ~f('pit'), None, 'Nit'),

# Adding vikaraa "sap"
('3.1.77', F.gana('tu\da~^'), None, None, k('Sa')),

# Performing dvirvacaana
# do_dvirvacaana is an unparameterized operator defined separately.
('6.1.8', None, ~f('abhyasta'), 'li~w', do_dvirvacaana),

# Vowel substitution
# _6_4_77 is an unparameterized operator defined separately.
('6.4.77', None, snu_dhatu_yvor, None, _6_4_77),
# Replacing 'jh' with 'a'

('7.1.3', None, None, None, O.replace('J', 'ant'))

Those familiar with these rules will wonder why so much crucial information is missing (e.g. that the center context in 7.1.3 should be a pratyaya). This information is supplied in a special decorator, which we discuss now.

## 2.5.2 @inherit

When an Ashtadhyayi object is created, the system searches through all modules for functions decorated with the inherit() decorator. These functions create and return a list of rule tuples. An example:

```python
@inherit(None, F.raw('Sap'), None)
def sap_lopa():
    return [
        ('2.4.71', F.gana('a\da~'), None, None, 'lu~k'),
        ('2.4.74', F.gana('hu\'), None, None, 'Slu~')
    ]
```

inherit() takes at least 3 arguments, which correspond to the three contexts (left, center, and right). These arguments define base filters that are "and"-ed with all of the returned tuples. If the context in some rule tuple is None, the system uses just the base filter. That is, the rules above will take the following form:

```python
('2.4.71', F.gana('a\da~'), F.raw('Sap'), None, 'lu~k'),
('2.4.74', F.gana('hu\'), F.raw('Sap'), None, 'Slu~')
```

## 2.5.3 Rule conditions

The majority of the Ashtadhyayi’s rules consists of some context window and an operator. But many rules are modified by some other term, such as na (blocking) or vibh (optional). These terms are defined as subclasses of RuleTuple:

```python
# 'i' augment denied
Na('7.2.8', None, None, f('krt') & F.adi('vaS'), U('iw')),

#: Denied in another context
Ca('7.2.9', None, f('krt') & titutra, None, True),
```

## 2.5.4 Converting tuples to rules

To interpret a rule tuple, we need:

- the tuple itself
- the previous tuple
- any base filters defined in the inherit() function.

These are combined as described above. For details, see `vyakarana.inference.create_rules()`.
API Reference

This contains information about specific classes, functions, and methods.

3.1 API

3.1.1 Lists

vyakarana.lists

Lists of various terms, designations, and sounds. Some of these lists could probably be inferred programmatically, but for the sake of basic sanity these are encoded explicitly. Thankfully these lists are rather small.

license MIT and BSD

vyakarana.lists.DHATUKA = ['sarvadhatuka', 'ardhadhatuka']
sajñ for verb suffixes

vyakarana.lists.IT = set(['wvit', 'Git', 'adit', 'odit', 'Sit', 'anudattet', 'kit', 'Yit', 'wit', 'xdit', 'Udit', 'qit', 'pit', 'qvit', 'arit', 'xvit', 'Rit', 'svarita', 'idit', 'Kit', 'fdit', 'svaritet', 'cit', 'udit', 'mit', 'Nit'])
Technical designations (1.3.2 - 1.3.9)

vyakarana.lists.KARAKA = ['karta', 'karma', 'karana', 'adhikarana', 'sampradana', 'apadana']
sajñ for kraka relations (currently unused)

vyakarana.lists.LA = set(['la~w', 'li~N', 'lf~N', 'le~w', 'lu~N', 'lo~w', 'lu~w', 'li~w', 'la~N', 'lf~w'])
Abstract suffixes that are replaced with items from TIN. Collectively, they are called the “lakra” or just “la”.

vyakarana.lists.PADA = ['parasmaipada', 'atmanepada']
sajñ for verb ‘pada’

vyakarana.lists.PRATYAYA = set(['la~w', 'lf~N', 'Snam', 'SnA', 'Slu', 'lu~N', 'lo~w', 'li~w', 'la~N', 'la~w', 'lf~w', 'up', 'li~N'])
Various pratuyaya

vyakarana.lists.PURUSHA = ['prathama', 'madhyama', 'uttama']
sajñ for various persons

vyakarana.lists.SAMJNA = set(['pada', 'atmanepada', 'abhyasta', 'vrddhi', 'ekavacana', 'prathama', 'saptami', 'sarvadhatuka']
All sañ

A collection of various sounds, including:

• savara sets (1.1.69)

• single-item sets (1.1.70)
3.1.2 Inputs and Outputs

class vyakarana.terms.Upadesha (raw=None, **kw)
A term with indicatory letters.

data
The term’s data space. A given term is represented in a variety of ways, depending on the circumstance. For example, a rule might match based on a specific upadesa (including ‘it’ letters) in one context and might match on a term’s final sound (excluding ‘it’ letters) in another.

samjna
The set of markers that apply to this term. Although the Ashtadhyayi distinguishes between samjna and it tags, the program merges them together. Thus this set might contain both ‘kit’ and ‘pratyaya’.

lakshana
The set of values that this term used to have. Technically, only pratyaya need to have access to this information.

ops
The set of rules that have been applied to this term. This set is maintained for two reasons. First, it prevents us from redundantly applying certain rules. Second, it supports painless rule blocking in other parts of the grammar.

parts
The various augments that have been added to this term. Some examples:
• ‘aw’ (verb prefix for past forms)
• ‘iw’ (‘it’ augment on suffixes)
• ‘vu~k’ (‘v’ for ‘BU’ in certain forms)

static as_anga (*a, **kw)
Create the upadesha then mark it as an ‘anga’.

static as_dhatu (*a, **kw)
Create the upadesha then mark it as a ‘dhatu’.

adi
The term’s first sound, or None if there isn’t one.

antya
The term’s last sound, or None if there isn’t one.

asiddha
The term’s value in the asiddha space.

asiddhavat
The term’s value in the asiddhavat space.
The term’s value without svaras and anubandhas.

The term’s raw value.

The term’s penultimate sound, or None if there isn’t one.

The term’s value in the siddha space.

Parameters names – the lakshana to add

Parameters names – the ops to add

Parameters names – the parts to add

Parameters names – the samjna to add

Parameters names –

Parameters locus –

Parameters names – the samjna to remove

Parameters asiddha – the new asiddha value

Parameters asiddhavat – the new asiddhavat value

Parameters

• locus –

• value –

Parameters raw – the new raw value

Parameters value – the new value

A sequence of terms.

This represents a single step in some derivation.

class vyakarana.derivations.State(terms=None, history=None)

3.1. API
3.1.3 Filters

vyakarana.filters

Excluding paribh, all rules in the Ashtadhyayi describe a context then specify an operation to apply based on that context. Within this simulator, a rule’s context is defined using filters, which return a true or false value for a given index within some state.

This module defines a variety of parameterized and unparameterized filters, as well as as some basic operators for combining filters.

license  MIT and BSD

class  vyakarana.filters.Filter (*args, **kw)

Represents a “test” on some input.

Most of the grammar’s rules have preconditions. For example, the rule that inserts suffix `snam` applies only if the input contains a root in the `rudh` group. This class makes it easy to define these preconditions and ensure that rules apply in their proper contexts. Since these conditions filter out certain inputs, these objects are called filters.

Originally, filters were defined as ordinary functions. But classes have one big advantage: they let us define custom operators, like `&`, `|`, and `~`. These operators give us a terse way to create more complex conditions, e.g. `al('hal') & upadha('a')`.

category = None

The filter type. For example, a filter on the first letter of a term has the category `adi`.

ame = None

A unique name for the filter. This is used as a key to the filter cache. If a filter has no parameters, this is the same as `self.category`.

body = None

The function that corresponds to this filter. The input and output of the function depend on the filter class.

For a general `Filter`, this function accepts a state and index and returns `True` or `False`.

domain = None

A collection that somehow characterizes the domain of the filter. Some examples:

- for an `al` filter, the set of matching letters
- for a `samjna` filter, the set of matching samjna
- for a `raw` filter, the set of matching raw values
- for an and/or/not filter, the original filters

classmethod  no_params (fn)

Decorator constructor for unparameterized filters.

Parameters  fn – some filter function.

supersets

Return some interesting supersets of this filter.

Consider a universal set that contains every possible element. A filter defines a subset of the universal set, i.e. the set of items for which the filter returns `True`. Thus every filter defines a set. For two filters `f1` and `f2`: 
• $f_1 \& f_2$ is like an intersection of two sets
• $f_1 \mid f_2$ is like a union of two sets
• $\neg f_1$ is like an “antiset”

Now consider a filter $f$ composed of $n$ intersecting filters:

$$f = f_1 \& f_2 \& \ldots \& f_n$$

This function returns the $n$ filters that compose $f$. Each $f_i$ is essentially a superset of $f$.

“Or” and “not” filters are tough to break up, so they’re treated as indivisible.

subset_of (other)
Return whether this filter is a subset of some other filter.

All members of some subset $S$ are in the parent set $O$. So if it is the case that:

$S$ applies -> $O$ applies

then $S$ is a subset of $P$. For the “set” interpretation of a filter, see the comments on supsets().

Parameters other – a filter

class vyakarana.filters.TermFilter (*args, **kw)
A Filter whose body takes an Upadesha as input.

Term filters give us:

• Convenience. Most filters apply to just a single term.
• Performance. Since we can guarantee that the output of a term filter will change only if its term changes, we can cache results for an unchanged term and avoid redundant calls.

class vyakarana.filters.AlFilter (*args, **kw)
A filter that tests letter properties.

class vyakarana.filters.adi (*args, **kw)
Filter on a term’s first sound.

class vyakarana.filters.al (*args, **kw)
Filter on a term’s final sound.

class vyakarana.filters.contains (*args, **kw)
Filter on whether a term has a certain sound.

class vyakarana.filters.dhatu (*args, **kw)
Filter on whether a term represents a particular dhatu.

vyakarana.filters.gana (start, end=None)
Return a filter on whether a term is in a particular dhatu set.

Parameters

• start – the raw value of the first dhatu in the list
• end – the raw value of the last dhatu in the list. If None, use all roots from start to the end of the gana.

class vyakarana.filters.lakshana (*args, **kw)
Filter on a term’s prior values.

class vyakarana.filters.part (*args, **kw)
Filter on a term’s augments.
class vyakarana.filters.raw(*args, **kw)
    Filter on a term’s raw value.

class vyakarana.filters.samjna(*args, **kw)
    Filter on a term’s designations.

class vyakarana.filters.upadha(*args, **kw)
    Filter on a term’s penultimate sound.

class vyakarana.filters.value(*args, **kw)
    Filter on a term’s current value.

tyakarana.filters.auto(*data)
    Create a new Filter using the given data.

    Most of the terms in the Ashtadhyayi have obvious interpretations that can be inferred from context. For ex-
ample, a rule that contains the word dhato clearly refers to a term with dhtu as a sajñ, as opposed to a term with


dhbu as its current value. In that example, it’s redundant to have to specify that F.samjna(’dhatu’) is a

    samjna filter.

    This function accepts a string argument and returns the appropriate filter. If multiple arguments are given, the

    function returns the “or” of the corresponding filters. If the argument is a function, it remains unprocessed.

    Parameters data – arbitrary data, usually a list of strings

3.1.4 Operators

vyakarana.operators

Excluding paribh, all rules in the Ashtadhyayi describe a context then specify an operation to apply based on that
context. Within this simulator, operations are defined using operators, which take some (state, index) pair and return
a new state.

This module defines a variety of parameterized and unparameterized operators.

    license MIT and BSD

class vyakarana.operators.Operator(*args, **kw)
    A callable class that returns states.

    category = None
        The operator type. For example, a substitution operator has category tasya.

    name = None
        A unique name for this operator. If the operator is not parameterized, then this is the same as self.category.

    body = None
        The function that corresponds to this operator. The input and output of the function depend on the operator
class. For a general Operator, this function accepts a state and index and returns a new state.

    params = None
        the operator’s parameters, if any.

    classmethod parameterized(fn)
        Decorator constructor for parameterized operators.

        Parameters fn – a function factory. It accepts parameters and returns a parameterized operator
        function.

    classmethod no_params(fn)
        Decorator constructor for unparameterized operators.
Parameters fn – some operator function

conflicts_with (other)
Return whether this operator conflicts with another.

Two operators are in conflict if any of the following hold:
• they each insert something into the state
• one prevents or nullifies the change caused by the other. By “nullify” I mean that the result is as if
  neither operator was applied.

For example, two insert operators are always in conflict. And hrasva and dirgha are in conflict, since
hrasva undoes dirgha. But hrasva andguna are not in conflict, since neither blocks or nullifies the other.

Parameters other – an operator

class vyakarana.operators.DataOperator(*args, **kw)
An operator whose body modifies a term’s data.

body accepts and returns a single string.

3.1.5 Rules and Rule Stubs

class vyakarana.rules.Rule (name, window, operator, modifier=None, category=None, locus='value',
optional=False)
A single rule from the Ashtadhyayi.

Rules are of various kinds. Currently, the system deals only with transformational rules (“vidhi”) explicitly.

VIDHI = ‘vidhi’
Denotes an ordinary rule

SAMJNA = ‘samjna’
Denotes a saji rule

ATIDESHA = ‘atidesha’
Denotes an atideśa rule

PARIBHASHA = ‘paribhasha’
Denotes a paribh rule

name = None
A unique ID for this rule, e.g. ‘6.4.1’. For most rules, this is just the rule’s position within the
Ashtadhyayi. But a few rules combine multiple rules and have hyphenated names, e.g. ‘1.1.60 –
1.1.63’.

filters = None
A list of filter functions to apply to some subsequence in a state. If the subsequence matches, then we can
apply the rule to the appropriate location in the state.

operator = None
An operator to apply to some part of a state.

locus = None

optional = None
Indicates whether or not the rule is optional

utsarga = None
A list of rules. These rules are all blocked if the current rule can apply.
**apply** *(state, index)*

Apply this rule and yield the results.

**Parameters**

- **state** – a state
- **index** – the index where the first filter is applied.

**has_apavada** *(other)*

Return whether the other rule is an apavada to this one.

Rule B is an apavada to rule A if and only if:

1. \( A \neq B \)
2. If A matches some position, then B matches too.
3. A and B have the same locus
4. The operations performed by A and B are in conflict

For details on what (4) means specifically, see the comments on `operators.Operator.conflicts_with()`.

**Parameters**

- **other** – a rule

---

**vyakarana.templates**

This module contains classes and functions that let us define the Ashtadhyayi’s rules as tersely as possible.

**license**  MIT and BSD

**class** `vyakarana.templates.RuleStub` *(name, left, center, right, op, **kw)*

**Bases:** `object`

Wrapper for tuple rules.

The Ashtadhyayi uses a variety of terms to control when and how a rule applies. For example, ‘anyatarasym’

denotes that a rule specifies an optional operation that can be accepted or rejected.

In this system, these terms are marked by wrapping a rule in this class or one of its subclasses.

- **name** = None
  - The rule name
- **window** = None
  - The rule context
- **operator** = None
  - The rule operator

**class** `vyakarana.templates.Ca` *(name, left, center, right, op, **kw)*

**Bases:** `vyakarana.templates.RuleStub`

Wrapper for a rule that contains the word “ca”.

“ca” has a variety of functions, but generally it preserves parts of the previous rule in the current rule.

**class** `vyakarana.templates.Na` *(name, left, center, right, op, **kw)*

**Bases:** `vyakarana.templates.RuleStub`

Wrapper for a rule that just blocks other rules.
class vyakarana.templates.Nityam(name, left, center, right, op, **kw)
    Bases: vyakarana.templates.RuleStub
    Wrapper for a rule that cannot be rejected.
    This is used to cancel earlier conditions.

class vyakarana.templates.Option(name, left, center, right, op, **kw)
    Bases: vyakarana.templates.RuleStub
    Wrapper for a rule that can be accepted optionally.
    This is a superclass for a variety of optional conditions.

class vyakarana.templates.Anyatarasyam(name, left, center, right, op, **kw)
    Bases: vyakarana.templates.Option
    Wrapper for a rule that is indifferently accepted.
    Modern scholarship rejects the traditional definition of anyatarasym, but this system treats it as just a regular option.

class vyakarana.templates.Va(name, left, center, right, op, **kw)
    Bases: vyakarana.templates.Option
    Wrapper for a rule that is preferably accepted.
    Modern scholarship rejects the traditional definition of v, but this system treats it as just a regular option.

class vyakarana.templates.Vibhasha(name, left, center, right, op, **kw)
    Bases: vyakarana.templates.Option
    Wrapper for a rule that is preferably not accepted.
    Modern scholarship rejects the traditional definition of vibh, but this system treats it as just a regular option.

class vyakarana.templates.Artha(name, left, center, right, op, **kw)
    Bases: vyakarana.templates.Option
    Wrapper for a rule that applies only in some semantic condition.
    Since the semantic condition can be declined, this is essentially an optional provision.

class vyakarana.templates.Opinion(name, left, center, right, op, **kw)
    Bases: vyakarana.templates.Option
    Wrapper for a rule that is accepted by prior opinion.
    Since the opinion can be declined, this is essentially the same as an optional provision.

vyakarana.templates.Shesha = <object object at 0x7f303f6898d0>
Signals use of the `sea` device, which affects utsarga-apavada inference.

### 3.1.6 Texts

class vyakarana.ashtadhyayi.Ashtadhyayi(stubs=None)
    Given some input terms, yields a list of Sanskrit words.
    This is the most abstract part of the system and doesn’t expect any internal knowledge about how the system works. This is almost always the only class that client libraries should use.
    The heart of the class is `derive()`, which accepts a list of terms and yields `State` objects that represent finished words.
**derive** *(sequence)*
Yield all possible results.

**Parameters**
- **sequence** – a starting sequence

**rule_tree** = None
Indexed arrangement of rules

**classmethod with_rules_in** *(start, end, **kw)*
Constructor using only a subset of the Ashtadhyayi’s rules.

This is provided to make it easier to test certain rule groups.

**Parameters**
- **start** – name of the first rule to use, e.g. “1.1.1”
- **end** – name of the last rule to use, e.g. “1.1.73”

**class vyakarana.dhatupatha.Dhatupatha** *(filename=\None)*
A collection of all verb roots in the Sanskrit language.

This class makes it easy to select a continuous range of roots from the Dhtupha and query for other properties of interest, such as the original gaa.

All data is stored in a CSV file, which is read when the program begins.

The Dhtupha is traditionally given as a list of roots, each stated in upadeśa with a basic gloss. An example:

1.1 bh sattym

The first number indicates the root gaa, of which there are ten. This gaa determines the form that the root takes when followed by *srvadhtuka* affixes. The second number indicates the root’s relative position within the gaa.

Although few modern editions of the text have accent markings, the Sanskrit grammatical tradition has preserved the original accents all of the original items. Per the conventions of SLP1, these are written as follows:

<table>
<thead>
<tr>
<th>Accent</th>
<th>SLP1</th>
<th>Devanagari</th>
<th>IAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>udatta</td>
<td>(no mark)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>anudtta</td>
<td>\</td>
<td></td>
<td></td>
</tr>
<tr>
<td>svarita</td>
<td>^</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**all_dhatu** = None
List of all dhatu, one for each row in the original CSV file.

**dhatu_list** *(start, end=\None)*
Get an inclusive list of of dhatus.

**Parameters**
- **start** – the first dhatu in the list
- **end** – the last dhatu in the list. If None, add until the end of the gana.

**index_map** = None
Maps a dhatu to its indices in *self.all_dhatu*.

**init** *(filename)*

**Parameters**
- **filename** – path to the Dhatupatha file
V
vyakarana.filters, 22
vyakarana.lists, 19
vyakarana.operators, 24
vyakarana.templates, 26