A fast *prakriyā* generator

Arun Prasad
Ambuda
arun@ambuda.org

Abstract

We present `vidyut-prakriya`, a program that generates Sanskrit words along with their Pāṇinian derivations. `vidyut-prakriya` implements more than 2,000 rules of the *Aṣṭādhyāyī* and has strong support for *tiṅanta*s, *kṛdanta*s, *taddhitānta*s, and *subanta*s, with partial support for *samāsa*s and accent. Our program compiles to fast native code and also compiles to WebAssembly for in-browser use. Informal benchmarks indicate that `vidyut-prakriya` is almost three orders of magnitude faster than comparable open-source systems. We end by discussing various applications of a fast *prakriyā* generator and directions for future work.

1 Introduction

Many Sanskrit programs use and rely on a *lemma list* that contains verb roots, nominal stems, and other headwords. For example, an electronic dictionary maps a list of lemmas to a list of definitions. Some programs also rely on a *word list* that contains inflected versions of various lemmas. For example, a more sophisticated electronic dictionary might accept an inflected word then show results for the word's underlying lemma. This distinction between lemma lists and word lists is simple but important. Whereas lemma lists might contain entries like **गम्** and **देव**, word lists might contain entries like **गच्छति**, **जग्मिवांसम्**, **देव्यै**, and **देवानाम्**.

While a word list is useful for dictionaries and other query interfaces, it has other applications as well. For example, programs that analyze Sanskrit sentences have a long history of using word lists internally,[1] which continues into modern approaches like Sandhan et al. (2022). Additionally, popular resources for students of Sanskrit grammar, such as Bodas (2023), display hundreds of thousands of verbal and nominal forms for students, and these forms are regularly consulted by modern-day Sanskrit communities. S. Prasanna (2022) also illustrates the utility of a word list for spellchecking by using both an explicit list of irregular forms and an implicit list that joins base words with a suffix table. We believe that word lists are likewise valuable to any application that cares deeply about correctness.

Given these applications, we believe that an even larger word list might better suit some of these needs, which makes creating such a list a valuable problem to pursue. But since the Sanskrit word list is infinite and grows recursively, we cannot represent it in a straightforward way. So in practice, an infinite word "list" is rather a finite program that can generate words as needed. The challenge, then, is to to create such a program so that it solves needs similar to the ones we describe above.

One promising strategy for creating such a program, as demonstrated by Bharati et al. (2006) and others, is to combine an ad-hoc list of attested forms with some method of abstraction, such as a finite state automaton, a statistical model, or a set of manually implemented rules.

[1] Hellwig (2009) and Goyal and Huet (2016) are particularly notable for their longevity and impact. We also have high regard for the *Saṁsādhanī* toolkit from the University of Hyderabad, but we are less familiar with how it works internally.

This approach works well if the ad-hoc list is sufficiently rich to expose all of the edge cases and subtleties of Sanskrit grammar, but we have found that approaches in this vein, as impressive as they are, are prone to over- and under-generating, which means that they might allow invalid words and reject valid ones. This kind of behavior is not always a problem, and it can even be preferable for some use cases; but, it is less suited for applications that care deeply about correctness.

An alternative strategy is to directly implement the underlying rules that generate these words. This is the approach most famously taken by the *Aṣṭādhyāyī*, which condenses the mechanics of Sanskrit grammar to roughly 4,000 short rules. When combined with secondary texts like the *Dhātupāṭha* and *vārttika*s from the commentarial literature, the *Aṣṭādhyāyī* provides a powerful system for generating an infinite number of Sanskrit words.

Accordingly, there are several systems that have implemented parts of the *Aṣṭādhyāyī*, each with different philosophies and goals. Mishra (2009), for example, proposes a formal structure that is highly Pāṇinian in spirit, with each sound in a word having a specific formal representation. Scharf (2015) likewise creates a meticulous formal representation of the *Aṣṭādhyāyī*'s rules in XML that can be converted to executable code. Goyal et al. (2009) use a simpler internal representation but also describe a rich model for conflict resolution between rules. Patel and Katuri (2015), meanwhile, avoid simulating conflict resolution but gain substantial performance benefits in return.

A program that implements the *Aṣṭādhyāyī*'s rules can usually also create a *prakriyā*, a step-by-step derivation that shows which rules of grammar act to create a specific inflected word. For students, a *prakriyā* explains and elucidates the principles of the grammar; for engineers, a *prakriyā* reveals the grammar's operations in case it needs to be debugged; and for downstream applications, a *prakriyā* provides detailed grammatical information about a given word. Thus a *prakriyā* is highly useful in multiple settings.

Rule	Result
3.2.123	भू + लँट्
3.4.78	भू + तिप्
3.1.68	भू + शप् + ति
7.3.84	भो + अ + ति
6.1.78	भव् + अ + ति
(final)	भवति

Table 1: An abbreviated *prakriyā* for the word भवति. Items on the left are references to specific rules in the *Aṣṭādhyāyī*. We have elided various minor rules.

This paper presents our work toward building a comprehensive *prakriyā* generator based on the *Aṣṭādhyāyī*. We have implemented just over 2,000 rules from the *Aṣṭādhyāyī*, and these rules span all major sections of the text. In addition, we have implemented around 500 rules from the *Uṇādipāṭha* and around 50 rules from the *Pāṇinīyaliṅgānuśāsanam*. Our main contributions are the generator itself and our specific engineering decisions, which we believe make a complete implementation of the *Aṣṭādhyāyī* more feasible and useful.

Section 2 describes our approach and the engineering and grammatical principles we follow in this work. Section 3 describes our program's high-level architecture, including which texts we use as the basis of our work. Section 4 describes our implementation, including our data model, pieces of our API, a few example rules, and a description of how we handle optional derivations. Section 5 describes our testing methodology, which draws primarily from the traditional grammatical literature. Section 6 describes the current state of our work and its limitations. The remaining sections describe applications of this work, directions for future work, and our overall conclusions.

2 Approach

We have viewed our program first as an engineering problem and second as a grammatical one. That is, we have prioritized those principles that allow us to build a safe, fast, and consistent program that produces accurate results. Since we are non-grammarians, our approach is to some extent *anti*-theoretical, meaning that we have used traditional grammatical literature primarily as a source of examples but have not deliberately followed a specific system of interpretation or rule ordering.

2.1 Engineering strategy

Broadly, we follow a variety of engineering principles that we have found to lead to maintainable and high-quality software. While these principles are not absolutes, they are generally true in our system:

- *Don't reinvent the wheel.* We acknowledge that *prakriyā* generators are well-trod ground. That said, we have aimed to create a generator that improves on the state well enough to become a community standard. In service of this goal, we have implemented our generator in Rust, a relatively new systems language that combines low-level control with high-level ergonomics. One happy consequence of this decision is that we can easily bind our program to other languages. In other words, we can readily provide the same core implementation in multiple environments and languages. As proofs of concept, we have created Python bindings[2] through Rust's `PyO3` library and JavaScript bindings through Rust's `wasm-pack` ecosystem.[3]

- *Don't repeat yourself (DRY).* We generally implement a rule in exactly one place in our code, which means that each rule we use has exactly one formal specification. Where necessary, we reuse rules sparingly through function calls. Likewise, we maintain a simple, linear, and predictable control flow as opposed to (for example) dynamically selecting and ranking rules within an event loop.

- *If it isn't tested, it's broken.* The *Aṣṭādhyāyī* contains thousands of interconnected rules, and an innocuous change to one rule can easily break dozens of others. Therefore, we generally test each rule with representative examples and counterexamples from the *Kāśikāvṛtti*. We have also an extensive set of additional tests from the *Siddhāntakaumudī*. We explain more about our testing procedure in section 5.

- *Speed is a feature.* A comprehensive test suite is useful only if it can run in a reasonable amount of time. While we avoid over-optimizing (*Premature optimization is the root of all evil*), we have done enough that our full test suite, which creates roughly 1.7 million words, runs in under a minute on our 2019 MacBook Pro. In particular, Rust has continued to pay dividends not only through its native speed but also due to its rich ecosystem of high-performance libraries.

- *Given enough eyeballs, all bugs are shallow.* All of our code is open-source and available online.[4] So far, five people have reported bugs and five have submitted code to our program. As more people use our program and its results, we increase the likelihood of finding mistakes and errors, which we can then correct and add to our test suite.

2.2 Grammatical strategy

As non-grammarians without much expertise in traditional grammar, we have laid aside issues of theory and set ourselves a much humbler task: to generate a list of valid Sanskrit words while adhering to Pāṇinian rules as closely as we can. Specifically:

[2] Available at `https://pypi.org/project/vidyut/`

[3] Demonstrated at `https://ambuda-org.github.io/vidyullekha`.

[4] You may find our code at `https://github.com/ambuda-org/vidyut` under the `vidyut_prakriya` directory.

- In our schema, a word is *valid* if and only if it is attested by a grammatical authority. So far, we have focused most closely on the *Kāśikāvṛtti* and the *Siddhāntakaumudī* (SK) as published at Bodas (2023). We hope to expand our set of examples as time allows.
- We follow Patel and Katuri (2015) by manually ordering rules and avoiding any explicit model of conflict resolution. This policy greatly increases the program's efficiency because the program can run code in a simpler and more predictable way, which helps both the compiler and the CPU process the program more efficiently.
- We interpret a rule in whatever way will let us generate valid words and avoid invalid ones. Generally, we have followed the interpretations of the *Kāśikāvṛtti*, but there is no broader principle we apply when interpreting rules. We do not model *anuvṛtti*.
- For ease of reference, we prefer to group rules by their ordering in the text. For example, our implementation of 7.2.61 is immediately next to our implementation of rule 7.2.62, and likewise these two rules appear in a similar place in our overall control flow. It is often not possible to follow this condition, but we do so where we can.

3 Architecture

This section explains the high-level architecture of our program, with a low-level view of implementation details in section 4.

3.1 Texts used

We focus on the *Aṣṭādhyāyī* and also include *vārttika*s from the *Kāśikāvṛtti* and the *Siddhāntakaumudī*. Our approach to *vārttika*s has been to first find examples from the grammatical literature that the program does not support and then implement the *vārttika*s necessary to support those examples. We have taken this approach because we think that doing so helps us better shape the overall design of the program.

We have consulted and used specific *paribhāṣā*s from the *Paribhāṣenduśekhara* only if doing so was necessary to resolve an explicit error in our program. For example, we have made use of **श्तिपा शपानुबन्धेन** ... for our *yaṅ-luk* derivations.

Our *Dhātupāṭha*, which comes from Bodas (2023), is a superset of *Dhātupāṭha*s from various sources, including the *Siddhāntakaumudī*, the *Mādhavīyadhātuvṛtti*, and the *Bṛhaddhātukusumākara*. The *dhātu*s in this list include *anubandha*s, and we have also modified the list to explicitly include accent. Some example entries in SLP1 transliteration include `RI\Y`, `qukf\Y`, and `qupa\ca~^z`. Our program is not coupled to any specific *Dhātupāṭha* and will accept any *dhātu* as long as the user defines its *gaṇa* and *antargaṇa* and correctly specifies its *anubandha*s and accent.

The other texts we use, including the *Uṇādipāṭha* and the *Gaṇapāṭha*, are as specified in the *Siddhāntakaumudī*.

3.2 Data model

Traditional grammar describes the items in a derivation with various labels, such as *pratyaya*, *āgama*, *dhātu*, *prātipadika*, *abhyāsa*, and so on. In our data model, all of these concepts are aspects of a more general notion that we call a *term*.[5] In our program, we explicitly model and store all of the following for a given term:

- The term's "visible" or surface representation, which will change according to the rules of the grammar.
- The term's instructional (**औपदेशिक**) form, which is how the term is first described in the grammar. This form includes accents and *anubandha*s where applicable.

[5] We do not know if this concept has a traditional name.

- All designations (संज्ञा) that are added by the rules of the grammar, as well as other properties that we describe in 4.1.
- All prior instructional forms, if full substitution applies by 1.1.55 (अनेकाल्शित्सर्वस्य).

For example, our program will represent the verb root डुकृञ् as having the instructional form `qukf\Y` (in SLP1 transliteration), the visible form `kf` (which might change to `kar`, `kAr`, etc. during the derivation), and the designations of *dhātu*, *aṅga*, *ḍvit*, and *ñit*. We also store that the root vowel is *anudātta* since this information is necessary to trigger certain rules.

All other details of our data model are less important, and we refer the reader to section 4.1 for details.

3.3 Argument model

Since the *Aṣṭādhyāyī* is a precise system, we require the user to specify their input conditions in precise detail. For example, suppose that we wish to derive the word `kArayan`. To do so, we must tell the program that we wish to derive a masculine nominative singular *subanta* from the *kṛdanta* formed by adding *śatṛ* to the *sanādi-dhātu* formed by adding the suffix *ṇic* to the *mūla-dhātu ḍukṛñ* that is listed in *tanādigaṇa*. This complex idea becomes clearer when represented as an s-expression:

```
(subanta
    (kṛdanta
        (dhatu
            sanadi
            (dhatu mula qukf\Y tanādi)
            Ric)
        Satf)
    masc nom sg)
```

We have tried to strike a balance between precision and pedantry, and we use Rust's rich type system to guide the user toward a correct request. For example, a Rust program that tries to add a *kṛt* suffix to a *prātipadika* will fail at compile time because a *kṛt* suffix can be added only after a *dhātu*. Section 4.4 contains details on our specific representation in code.

3.4 Rule model

We model a rule as having two parts: a *filter* that matches certain conditions and an *operator* that applies some change to the grammar. In code, this model naturally maps to a simple `if` statement:

```
if filter(prakriya) {
  operator(prakriya);
}
```

The most important aspect of this model is that it does not model *anuvṛtti*: for each rule, the program must explicitly specify all of the conditions necessary for the rule to apply. By avoiding this critical part of the grammar, our program becomes simpler and faster.

That said, an analogue of *anuvṛtti* exists as follows. If multiple rules can apply only when some condition `x` is true, we have found it convenient to write these rules like so:

```
if x {
    if filter_1(prakriya) {
      operator_1(prakriya);
    } else if filter_2(prakriya) {
      operator_2(prakriya);
    }
}
```

3.5 Optional rules

Rules can also apply optionally under various conditions. Our program supports two kinds of option and models them in different ways.

First, we support rules qualified by words indicating an option (*vā*, *vibhāṣā*, *anyatarasyām*, ...), rules qualified by the name of some grammarian, and most rules defined only in a specific semantic sense as follows:

1. Suppose that the program sees optional rules A, B, and C during the derivation. By default, the program accepts each rule. In addition, it also records that rules A, B, and C apply optionally.

2. Once the derivation is complete, the program inspects the output of step (1) and notices that rules A, B, and C are optional. it then creates three new combinations from these rules, namely $(A, B, \neg C)$, $(A, \neg B)$, and $(\neg A)$. Here $\neg A$ means a rejection of A. These three "rule paths" are all added to a queue of unexplored combinations.

3. As long as the queue is non-empty, we pop a path from the queue and run the derivation again using the rule decisions that the path describes. If this process encounters new optional rules, we likewise create new rule paths for them. For example, if we find that we can follow the rule path $(A, \neg B, D, E)$, then we add $(A, \neg B, \neg D)$ and $(A, \neg B, D, \neg E)$ to the path.

To model a choice between three or more options, we split the rule into two binary options. For example, consider rule 3.1.40 (**कृञ्चानुप्रयुज्यते लिटि**), which provides a choice of three different verb roots (**कृ**, **भू**, and **अस्**) when deriving the periphrastic perfect tense. In this scenario, our program first chooses whether or not to use **भू** then chooses whether or not to use **अस्**.

This basic model has been workable for most rules but is too crude for *taddhita* rules, where a specific suffix is available under different rules in different meaning conditions. We instead model such rules as follows:

1. When using the program, the user can optionally request that a *taddhita* should be added only with a specific meaning.

2. When checking if a *taddhita* rule can apply, the program first checks if the user requested a specific meaning that is compatible with the rule. If so, and if the rule requires a different meaning condition, the program skips the rule. If the user did not request a specific meaning, the program simply adds the *taddhita* as long as at least one rule can provide it for the user's input conditions.

3.6 Conflict resolution

We provide no explicit model of conflict resolution. That is, if rule 2 can block rule 1, our program does not inspect the properties of these two rules and has no explicit logic to rank or decide which should apply. Instead, we take the approach similar to Patel and Katuri (2015) and implement these rules as follows:

```
if filter_2(prakriya) {
  operator_2(prakriya);
} else if filter_1(prakriya) {
  operator_1(prakriya);
}
```

We stress that this approach neglects a core issue in interpreting and modeling the *Aṣṭādhyāyī*. It provides no model of a rule's properties, no explicit encoding of common *paribhāṣā*s like **पूर्वपरनित्यान्तरङ्गापवादानामुत्तरोत्तरं बलीयः**, no convenient way to reorder or rerank rules (other than editing code), and no precise way of informing the user that one rule has blocked another.

As compensation, however, this approach provides code that is much simpler, much faster, and much easier to change and understand, and we believe that these advantages are highly compelling.

3.7 Rule ordering

Given the rule model we describe above, a natural question is how we decide whether one rule should apply before another. Simply, our ordering is ad-hoc and chosen in whatever way will correctly generate the test cases we describe in section 5 while adhering to the engineering principles we described in 2.1. While this approach is crude, our use of the "DRY" principle of non-repetition in code means that our program generally commits to a single ordering that applies consistently across *all* examples in our test suite.

We say that this principle applies *generally* because we have occasionally had to violate it by duplicating a rule. For a trivial example, we apply the *it-saṃjñā* rules and most *saṃjñā* rules whenever a term is added to the derivation. For a more interesting example, our system applies rule 6.1.66 (**लोपो व्योर्वलि**) twice: once before *guṇa* so that we can correctly derive **क्रोपयति** (from **क्रूय्** + **पुँक्** + **णिच्**) and once after *guṇa* so that we can correctly derive **अभूत्** (from **भू**).

3.8 Rule organization

Conceptually, we group our rules into two categories: *preparing* rules and *finalizing* rules. *Preparing* rules introduce terms to the derivation but apply few changes, and they include most rules in *adhyāya*s 1-5. *Finalizing* rules apply various phonetic changes to terms in the derivation, and they include most rules in *adhyāya*s 6-8. We have separated rules in this way so that we can support nested derivations, which we have tested in basic cases but not yet in recursive ones.

For example, let us continue with the example of `kArayan` that we mentioned in section 3.3. Given our specification, the program prepares each item from the innermost out then finalizes the derivation to create the requested output:

1. *Prepare* `qukf\Y` by adding it to the derivation and applying *it-saṃjñā-lopa* and other *saṃjñā* rules.

2. *Prepare* `Ric` by adding it to the derivation and applying the same rules as in (1).

3. *Finalize* to create the root `kAri`. Our current implementation always runs this phase after adding *sanādi* suffixes because doing otherwise causes the program to fail on some of our test examples.

4. *Prepare* `Satf~` by first adding `la~w` to the derivation then replacing it with `Satf~`, as specified by 3.1.124 (**लटः शतृशानचावप्रथमासमानाधिकरणे**). We then apply *it-saṃjñā-lopa* and other *saṃjñā* rules. `Satf~` also conditions the addition of `Sap`, so we add it then apply the same rules as in (1).

5. *Prepare* `su~` by adding it to the derivation and applying the same rules as in (1). The derivation at this stage is `kAri + a + at + s`.

6. *Finalize* to apply substitutions and sound change rules. The final word is `kArayan`.

4 Implementation

Here we explain some implementation details from our system, which amounts to around 25,000 source lines of code excluding tests. All code in this section is adapted directly from our implementation, with light edits for clarity and readability.

We store text data internally with SLP1, which simplifies our underlying logic and reduces the computational overhead for core operations. For details on SLP1, see Scharf and Hyman (2012).

4.1 Core data types

We start by presenting the core data types in our program. Almost all of our program's rules involve testing and transforming the data types below.

Our most basic data type is a *Term*, which is a string annotated with additional metadata:

```
struct Term {
    upadesha: Option<String>,
    text: String,
    sthanivat: String,
    tags: EnumSet<Tag>,
    gana: Option<Gana>,
    antargana: Option<Antargana>,
    lakshanas: Vec<String>,
    svara: Option<Svara>
}

enum Tag { .. }
enum Gana { .. }
enum Antargana { .. }
enum Svara { .. }
```

Here are some of the key terms in the example above:

- `struct` indicates a heterogeneous collection of fields.
- `enum` indicates an *enumeration*, i.e. a choice of one item among the members in a list.
- `Vec` indicates a list of elements.
- `Option` is how Rust models a field that can be either present or absent.
- `EnumSet` is a third-party library that lets us model a set of `enum` values.

The most important fields on `Term` are `text`, which is the surface representation of this term; `upadesha`, which is the instructional (**औपदेशिक**) representation as enunciated in works like the *Dhātupāṭha*; and `lakshanas`, which contains a history of substitutions for this term, as per rule 1.1.62 (**प्रत्ययलोपे प्रत्ययलक्षणम्**). In particular, `upadesha` lets our code process a `Term` in predictable ways despite any phonetic changes to the surface form in `text`. `sthanivat`, which we named by reference to rule 1.1.56 (**स्थानिवदादेशोऽनल्विधौ**), is necessary for certain rules in our implementation, particularly those that deal with *dvitva* on a causative root.[6] `svara` specifies the accent type and (if applicable) which vowel in the term the accent should apply to.[7]

The `Tag` enum generalizes the *saṃjñā* concept from traditional grammar. In addition to including traditional *saṃjñā*s, `Tag` also contains various flags that are useful for the derivation. For example, we indicate that a term's last *a* vowel has been deleted through the tag `FlagAtLopa`.

We model a derivation as a list of `Term` structs along with additional metadata:

```
struct Prakriya {
    terms: Vec<Term>,
    tags: EnumSet<Tag>,
    history: Vec<Step>,
    config: Config,
    rule_decisions: Vec<RuleChoice>,
```

[6] We chose the name `sthanivat` for this concept for lack of a better name. We wish to assure the reader that we are aware of how important rule 1.1.56 and have modeled it appropriately throughout the program.

[7] Rust `enum`s are more powerful than simple enumerations and can also contain extra data. This type of model is more properly known as a *sum type*.

```
    lakara: Option<Lakara>,
}

enum RuleChoice {
    Accept(Rule),
    Decline(Rule),
}

enum Rule {
    Ashtadhyayi(String),
    Kashika(String),
    Dhatupatha(String),
    Unadi(String),
    Linganushasana(String),
    Kaumudi(String),
}
```

The syntax here is broadly similar to the `Term` illustration above. For brevity, we have elided the `Config` and `Step` structs, which are straightforward. In the definitions of `RuleChoice` and `Rule`, note that Rust's `enum` type can also associate data with each enum variant.

After `terms`, the most noteworthy fields here are `history` and `rule_decisions`. `history` stores each rule applied in the derivation along with its result. `rule_decisions` stores specific decisions on optional rules that were encountered during the derivation, and we refer the reader to section 3.4 for details.

The strength of this data model is that we can split a word's derivation into separate strings that each have their own metadata, and we have found such a data model to be highly convenient for most rules. That said, its main limitation is that it is not a hierarchical representation, meaning that we cannot easily take either a broader or narrower view:

- *The broader view:* Our representation is most convenient when one term represents one core notion, such as a pratyaya. But during a derivation, we might introduce various *āgama*s that come between a *pratyaya* and the base it follows. In this situation, we most frequently resort to a `TermView`, a new data structure that abstracts over multiple `Term`s. While `TermView` is useful and has reasonable ergonomics, it ultimately highlights a weakness of the underlying data model.

- *The narrower view:* Our representation assumes that each sound in a derivation belongs to exactly one term. However, this assumption does not always hold. For example, certain derivations apply rules that fall in the jurisdiction of rule 6.1.84 (एकः पूर्वपरयोः), which states that a single substitute should be treated as part of both the previous and the following items. Our data schema above cannot model this, and we have worked around it with a small hack: we annotate the first term in the sandhi combination with the `FlagAntyaAcSandhi` tag and block rules like 8.2.39 (झलां जशोऽन्ते) from applying between two terms if the first term has this tag. In the future, we might implement an explicit model of rule 6.1.84 that our API can check.

A more Pāṇinian data structure might be a list of string spans that each have their own metadata. But in our view, it is not obvious how to create a clean and efficient API for working with such spans. Despite its limitations, our list of `Term`s has been successful so far.

4.2 API

On top of these data types, we have created a rich API that lets us easily inspect, test, and transform our derivation state. Below is a representative example of our `Term` API:

```
impl Term {
    fn antya(&self) -> Option<char> {
        self.text.chars().rev().next()
    }

    fn has_antya(&self, char: c) -> bool {
        self.antya() == Some(c)
    }

    fn set_antya(&mut self, s: &str) {
        let n = self.text.len();
        if n >= 1 {
            self.text.replace_range(n - 1..n, s);
        }
    }
}
```

Here, `antya` returns the `Term`'s final character, or `None` if the term is empty. Likewise, `has_antya` tests whether the `Term` has a given final sound, and `set_antya` modifies the last sound of the `Term` in-place. (`set_antya` accepts multiple characters to better support certain kinds of substitutions.)

In our production code, `has_antya` uses Rust's support for generic arguments to also accept a *set* of sounds, which lets us test (for example) whether a given `Term` ends in a specific *pratyāhāra*.

`Prakriya` likewise exposes a high-level API for changing internal state. These functions primarily accept *closures*, which are akin to inline functions. We use closures in part for readability and in part because we found it easier to do so while complying with Rust's semantics. Below is a representative example of our `Prakriya` API:

```
impl Prakriya {
    fn run(&mut self, rule: Rule, operator: impl Fn(&mut Prakriya)) -> bool {
        operator(self);
        self.step(code);
        true
    }

    fn run_optional(&mut self, rule: Rule, operator: impl Fn(&mut Prakriya)) -> bool {
        if self.is_allowed(rule) {
            operator(self);
            self.step(rule);
            true
        } else {
            self.decline(rule);
            false
        }
    }
}
```

4.3 Implementing rules

Together, this combination lets us tersely express rules in a human-readable way without sacrificing performance:

```
    let yi_kniti = next.has_adi('y') && next.is_knit();
    if anga.has_upadesha("SIN") && next.is_sarvadhatuka() {
        prakriya.run_at("7.4.21", anga_index, |term| term.set_text("Se"));
```

```
} else if anga.has_upadesha("SIN") && yi_kniti {
    prakriya.run_at("7.4.22", anga_index, |term| term.set_text("Say"));
}
```

Note the explicit ordering of rules, the relatively terse code, and the use of `if-else` chains to implement rule blocking. Our production code is largely the same, but it heavily abbreviates common terms like `prakriya` and `term` and contains some method names that are legacies of earlier stages of development.

4.4 Argument types

Our public API requires users to define their input conditions with precise types. To continue the example from section 3.3, our program models the input conditions for `kArayan` as follows, with some minor syntax elided for clarity:

```
let kr = Dhatu::mula("qukf\\Y", Tanadi).with_sanadi(&[Sanadi::Ric]);
let karayat = Krdanta::new(kr, Krt::Satf);
let karayan = Subanta::new(karayat, Pum, Prathama, Eka);
```

4.5 Performance

Since our full test suite has more than a million examples, we wish to ensure that our program completes in a reasonable amount of time. To that end, we have taken reasonable steps to ensure that the developer experience does not suffer.

Perhaps the most significant decision here is our choice of the Rust programming language. Rust's combination of speed and memory safety makes it an attractive choice for high-performance programming projects, as noted in work like Bugden and Alahmar (2022). In addition to Rust's native capabilities, we wish to highlight three other features that have benefited our program:

- *Strong tooling.* The default Rust installation includes a tool called `cargo`, which manages dependencies, builds code for different environments, runs tests, lints and formats code, and catches common style problems. `cargo` has allowed us to set aside ancillary concerns and focus on implementing rules, and it has also made it easier for us to onboard new contributors to our work.

- *Useful libraries.* Rust maintains a centralized package repository where library versions are guaranteed to be stable and available. For example, the `compact_str` library gives us access to memory-efficient string types that can be stack-allocated if they are sufficiently short. As another example, the `rayon` library provides a lightweight library for parallel execution on iterators, which allows us to more quickly generate a very large word list.

- *Easy reuse.* Rust is easy to bind to other languages and environments, which removes the toil of porting an implementation to another setup. As a proof of concept, we have created the `vidyut` Python library, which is available on Python's standard package index. We have also created a WebAssembly build that can run on a user's device without an internet connection.

5 Testing

The *Aṣṭādhyāyī* generates an infinite number of words, and it is impossible to test them all. Therefore, any implementation of the *Aṣṭādhyāyī* must have a robust testing strategy to justify some level of correctness.

We have tested our program with three kinds of tests: *unit tests* from the *Kāśikāvṛtti*, *regression tests* from the *Siddhāntakaumudī*, and *snapshot tests* that monitor changes over time. Our unit tests and regression tests combine to just under 32,000 lines of source code, and our snapshot test suite contains just under 1.7 million examples.

5.1 Unit tests

Generally, the ancient grammatical literature illustrates the function of a rule with various examples and counterexamples, which establish and limit the rule's scope respectively. We have especially leaned on the *Kāśikāvṛtti* for these examples, since it works through the *Aṣṭādhyāyī* rule by rule and generally limits its commentary to one rule at a time.

A typical unit test appears as follows:

```
#[test]
fn sutra_3_1_68() {
    assert_has_lat(&[], &dhatu("BU", Bhvadi), &["Bavati"]);
    assert_has_tip(&[], &dhatu("qupa\\ca~^z", Bhvadi), &["pacati"]);
}
```

We have created a variety of test functions like `assert_has_lat`, `assert_has_tip`, etc. to verify certain forms. In the example above, the first argument is for *upasarga*s, and `&[]` indicates that we wish to derive the given form without *upasarga*s. Note that `"qupa\\ca~^z"` contains both *anubandha*s and accent marks, both of which are necessary for the program to run correctly. Note also that we use `assert_has_tip` to restrict the output for **पच्** to use only `parasmaipada` endings, if they are available for the root. If we used `assert_has_lat` instead, then the program would produce both **पचति** and **पचते** and the test would fail.

We implement a rule's tests by including all examples mentioned in the *Kāśikāvṛtti*'s commentary on that rule, to the extent that we are able to. We fail to do this either if we are unclear on the intended result or if bugs or other technical limitations prevent us from implementing a given example. For now, we mark these challenging examples with `TODO`s, or in more significant cases, we disable the test entirely. For example, we have disabled the test for rule 3.2.12 (**स्तम्बकर्णयो रमिजपोः**) because we have not implemented the rule that allows non-deletion of the case suffix, which means we cannot produce the expected results **स्तम्बेरम** and **कर्णेजप**.

5.2 Regression tests

Although the *Kāśikāvṛtti* is thorough, its examples are sometimes insufficient to verify that the program is working correctly. In these cases, we have drawn examples from the *Siddhāntakaumudī* (SK), which tends to focus more on the overall *prakriyā* rather than on specific rules. For example, we once noticed a bug in our program where we failed to produce both **ऊर्णुनविथ** and **ऊर्णुनुविथ** (**ऊर्णुञ् + सिप्, लिँट्**). We found an illustration of this case in Kaumudī 2447 and accordingly implemented it as a test:[8]

```
#[test]
fn sk_2447() {
    let urnu = d("UrRuY", Adadi);
    assert_has_sip(&[], &urnu, Lit, &["UrRunuviTa", "UrRunaviTa"]);
    assert_has_tip(&[], &urnu, Lut, &["UrRuvitA", "UrRavitA"]);
    assert_has_tip(&[], &urnu, Lot, &["UrROtu", "UrRotu", "UrRutAt"]);
    assert_has_mip(&[], &urnu, Lot, &["UrRavAni"]);
    assert_has_iw(&[], &urnu, Lot, &["UrRavE"]);
}
```

Our test suite currently includes almost all examples from *prakaraṇa*s 8-13, which deal with *subanta*s, and 43-58, which deal with *tiṅanta*s. As our program stabilizes, we expect to add more and more test cases from the *Siddhāntakaumudī*.

[8] Some readers might ask how this test passes given that words like `UrRutAd` are also valid forms. Briefly, we have configured our test functions so that they avoid generating nosily duplicative forms. Otherwise, our test logic would be both more tedious to write and more cumbersome to read.

5.3 Snapshot tests

Our largest test suite is a *snapshot test* that deterministically generates around 1.7 million words, including all basic verbs in *kartari-prayoga* and *karmaṇi-prayoga*, all *kartari* and *karmaṇi* forms of *sannanta*, *ṇijanta*, *yaṅanta*, and *yaṅluganta* verbs, and a variety of *kṛdanta*s as well. We store the hashes of these files as part of our test suite, which means that if any result in any file changes, our test suite will raise an error that a human being must manually review.

Snapshot testing demonstrates stability but does not prove correctness. Even so, it is a useful tool for verifying that a rule change does not have unintended consequences.

6 Results

6.1 Implemented rules

	Pada 1	**Pada 2**	**Pada 3**	**Pada 4**
Adhyaya 1	27 / 75	31 / 73	72 / 93	36 / 110
Adhyaya 2	41 / 72	9 / 38	2 / 73	35 / 85
Adhyaya 3	117 / 150	139 / 188	79 / 176	38 / 117
Adhyaya 4	86 / 178	73 / 145	87 / 168	111 / 144
Adhyaya 5	64 / 136	64 / 140	73 / 119	62 / 160
Adhyaya 6	117 / 223	8 / 199	38 / 139	143 / 175
Adhyaya 7	80 / 103	112 / 118	84 / 120	86 / 97
Adhyaya 8	0 / 74	65 / 108	59 / 119	33 / 68

Table 2: Rule implementation of 2071 total rules by *adhyāya* and *pāda*. This is an undercount that includes only those rules that might show up in a *prakriyā*. *Paribhāṣā*s and simpler *saṃjñā* rules like **ह्रस्वं लघु** are not counted here.

Table 2 shows our total count of implemented rules by *adhyāya* and *pāda*. Rules are drawn broadly from all sections of the *Aṣṭādhyāyī*, with the notable exception of *pāda* 8.1.

- *Tiṅantas* have been our primary focus. Our system includes support for all *lakāra*s and *prayoga*s and implements almost all of the *pada* rules in section 1.3, including rules that depend on a specific *upasarga*. Our system also supports *sannanta*, *ṇijanta*, *yaṅanta* and *yaṅluganta* roots, with experimental support for denominative (**नामधातु**) roots. Example words that our program produces include **गच्छति**, **जिगमिषति**, **सञ्चस्कार**, **सिस्रावयिषति/सुस्रावयिष्यति**, **अर्दिधिषति**, and **पापचिषते**.

- *Kṛdantas* have been a secondary focus, and our system here has broad coverage for a variety of common suffixes, including **घञ्**, **ल्युट्**, **ण्वुल्**, **शतृ**, **क्त्वा**, **क्त**, **क्वसुँ**, and the like. We have also implemented support for around 500 rules from the *Uṇādipāṭha*. Example stems include **गत**, **अभ्यान्त/अभ्यमित**, and **जग्मिवस्/जगन्वस्**.

- *Taddhitāntas* likewise have broad coverage, and they explicitly model a variety of fine-grained meaning conditions with a Rust `enum`. We have met with some challenges here when implementing rules that match against a semantic class, such as "part of the body" as used in 4.3.55 (**शरीरावयवाच्च**). For our coverage here, see *adhyāya*s 4 and 5 in table 2.

- *Subantas* have strong support, but we have not run a formal evaluation against systems like Patel and Katuri (2015). As a rough measure of quality: excluding 8 sutras, we support all examples in *prakaraṇa*s 8-13 of the *Siddhāntakaumudī*.

- *Samāsa*s have basic support. We have implemented many of the rules from *pāda*s 2.1, 2.2, and 6.3 and support a basic model of *upapada-samāsa*s so that we can implement the various

kṛt-pratyaya rules in *pāda* 3.2. Otherwise, initial attempts to model this section have not felt satisfying since so many rules depend on specific semantic conditions and are relatively less mechanical than other sections of the text. One option here is to formally model these semantics in Rust then expose the Rust model through an API.

- *Vedic rules* have been a recent focus, and we currently support around ten of them.
- *Accent rules* have partial support. Currently, our accent model is limited to simple patterns like लित्, चित्, पित्, रित्, and the like. For details on our data model for accent, see 4.1.

6.2 Testing

Test coverage broadly follows the pattern of table 2, but we have disabled around 7 percent of sutra tests. This 7 percent figure is misleadingly high for two reasons. First, we have disabled tests where the system generates results that we don't know how to reconcile with the commentaries due to our own lack of grammatical knowledge. For example, the commentary on rule 7.2.58 (गमेरिट् परस्मैपदेषु) proposes the form सङ्गंसीष्ट as a counter example, but our system also generates सङ्गसीष्ट. We believe that both are correct but have not yet spent the time to vet this form. Second, we have ignored a test for a rule when any one of its examples is generated erroneously, no matter how rare that example might be.

The main comparison we wish to present here is with `SanskritVerb` (Patel (2023)), an extension of the work in Patel and Katuri (2015) with support for *kartari-tiṅanta*s. `SanskritVerb` is a standard implementation in the open-source community and is used in popular projects like Bodas (2023).

We performed an exhaustive comparison against all basic *kartari-tiṅanta*s generated by `SanskritVerb`, which total to around 200,000 forms. After fixing bugs in our setup, `vidyut-prakriya` features various small gains over `SanskritVerb`. A sample:

- Support for optional *karmaṇi-luṅ* forms (अप्यायि, अतायि, अदीपि).
- Support for optional *tanādi-luṅ* forms (अतत, अतथाः).
- Support for periphrastic perfect derivations with अस् and भू (चोरयामास, चोरयाम्बभूव).
- Support for optional forms like स्तभ्नोति, स्तुभ्नोति, and so on.
- Broad support for *ubhayapada* derivations in *curādi-gaṇa*.
- Stronger support for short vowel lengths in certain *curādi-gaṇa* roots, commonly known as *mittva*.
- Support for extra *ātmanepada* forms for various roots, such as स्था per rule 1.2.23.

Most differences are of this kind. They are small, incremental improvements based on supporting or tweaking an existing rule. One notable gain is that we support rule 3.1.31 (आयादय आर्धधातुके वा), which allows a wide variety of optional forms for various roots listed in rules 3.1.28 to 3.1.30. We can support rule 3.1.31 only because our program has strong support for optional derivations.

As time allows, we hope to also run a comparison with the Saṁsādhanī system from the University of Hyderabad.

6.3 Performance

Performance comparisons are an inexact art. This is especially so when comparing two very different systems. With that caveat, we describe an inexact performance comparison between `vidyut-prakriya` and `SanskritVerb`. Specifically, we measure how long it takes each program to generate all basic *kartari-tiṅanta*s for a verb root.

Setup	Time per root (ms)
`SanskritVerb`	1400
`vidyut-prakriya` (re-compile after code change)	7.0
`vidyut-prakriya` (re-compile without code change)	3.5
`vidyut-prakriya` (no re-compile)	3.3
`vidyut-prakriya` (no re-compile, no *prakriyā*s)	2.4

Table 3: Comparison of different setups when generating all basic *kartari-tiṅanta*s for a verb root. All setups were run on the same machine and timed with the `time` command. Results show the mean over a representative sample of roots (100 for `SanskritVerb` and 2200 for `vidyut-prakriya`). Here, "no *prakriyā*s" means that *prakriyā* logging is disabled.

All tests were run on the same machine and timed with Unix's `time` command, and these tests compare both systems as they existed around October 2023. We have not run an updated evaluation due to time constraints, but we believe that this comparison is still informative because the code paths being tested have not substantially changed since then.

Table 3 presents the results of this comparison under different scenarios. If using `vidyut-prakriya` as a release build with no extra compilation, we see that `vidyut-prakriya` is more than 400 times faster than `SanskritVerb`. If we disable *prakriyā* logging for `vidyut-prakriya`, we find that it is almost 600 times faster.

The most obvious explanation for this difference is that `vidyut-prakriya` is written in Rust while `SanskritVerb` is written in PHP, which is generally a much slower language than Rust. In addition, `SanskritVerb` tends to implement rules with regular expressions, which means that both checking whether a rule can apply and applying a rule to the derivation is relatively costly. In comparison, `vidyut-prakriya` tends to perform direct comparisons on stack-allocated strings, which is a much faster procedure.

PHP and other interpreted languages certainly have their own advantages. For example, testing a PHP code change on a single example is much faster than doing the same in Rust, which requires a compile step of several seconds. But if we wish to generate a *large* number of words, then a compiled project like `vidyut-prakriya` is much more effective. This is especially true for the millions of words we generate currently, but the same principle holds even for a few thousand.

6.4 Bugs and errors

`vidyut-prakriya` has a variety of small errors that we explicitly track in our unit and regression tests. These errors are typically limited to rare or unusual forms. Two examples, both from the *Siddhāntakaumudī*:

- **सुषुषुपतुः** — This is the third-person dual perfect of **ञिष्वपँ शये** as combined with the prefix **सु**. Currently, our program does not implement rule 8.3.88 (**सुविनिर्दुर्भ्यः सुपिसूतिसमाः**), so it incorrectly applies **षत्व** to **स्वप्** after **द्वित्व** and derives the wrong form **सुसुषुपतुः**.
- **सखा** – This is the masculine nominative singular form of **सखी** as derived from **सखि** + **क्यच्** + **क्विँप्**. Our current logic does not apply 7.1.93 (**अनङ् सौ**), so we instead derive the wrong form **सखीः**. **सखी** should not be confused with the much more common **सखि**, whose forms we derive correctly.

Errors like these are eminently fixable, and we can do so safely given our extensive test suite.

7 Applications

Our program's errors are currently limited to rare forms, and these errors are decreasing over time as our test suite expands. Given this work's current standing and future trajectory, we

think it is reasonable to explore some applications of its output.

7.1 As a word generator

We envision using the output of our word generator in one of two ways. Software with limited resources can use our program as-is and generate words on demand as requested by the user. For software with more resources, we have found that a finite state transducer (FST) is highly effective at storing Sanskrit words in a space-efficient way. Briefly, an FST generalizes the prefix tree and suffix tree into a single data structure, such that items that tend to share prefixes and suffixes can be stored with less memory. In one early experiment, we found that we could store tens of millions of Sanskrit words and their basic properties (person, number, etc.) at an amortized cost of roughly one byte per word. For details on this data structure and the specific Rust library we use, we refer the reader to Gallant (2015).

7.2 As a *prakriyā* generator

As a *prakriyā* generator, our program has obvious relevance to anyone who wants to understand how the *Aṣṭādhyāyī* might derive a specific word form. But there are applications beyond this narrow scope as well.

One tool we envision is an electronic grammar interface that is analogous to an electronic dictionary. Suppose a student could enter a word into a search interface and see a rule-by-rule grammatical breakdown of that word. With some translations of key rules, the same interface could offer explanations to students without a grounding in traditional grammar and thus comment on important high-level features in a user-friendly way. In doing so, such a tool could help ameliorate one of the key problems that Sanskrit students face: understanding the structure and function of a given word.

7.3 As a reference implementation of the *Aṣṭādhyāyī*

As our program progresses, we expect that others can start to use it as a reference implementation of the *Aṣṭādhyāyī*. By this, we mean that our program might become a baseline for new implementations or an experimental tool for answering certain kinds of grammatical questions. For example, someone who wishes to model conflict resolution more explicitly might adapt our API and test suite to a new core implementation. Or, a user might examine the impact of tweaking a rule's position or definition by running our program against our full test suite.

Again, we stress that our program takes no perspective on *anuvṛtti* or conflict resolution, and it is unclear what value our program could offer for exploring them, if any.

8 Future work

`vidyut-prakriya` offers several promising directions for future work.

The most promising direction is to continue implementing the rules of the *Aṣṭādhyāyī* and its secondary texts, provided that those rules are relevant for generating words. Our main ambition is to implement all such rules, including Vedic rules (i.e. marked with **छन्दसि**, **मन्त्रे**, **ऋचि**, and so on), the rest of the *Uṇādipāṭha*, and the Phiṭ Sutras.

Another direction is to tweak our *prakriyā*s and rule order to better conform to the conventions of modern grammarians. For example, our system derives **बभूव** by applying rule 6.1.8 (**लिटि धातोरनभ्यासस्य**) before the introduction of **वुँक्-आगम** by rule 6.4.88 (**भुवो वुग्लुङ्लिटोः**), which we have heard is unusual. Having a robust test case helps us make these kinds of changes with confidence.

A third direction is to add support for non-Pāṇinian usages like **उप** + **आस्** + **त्वा** → **उपासित्वा** (Pāṇinian **उपास्य**),[9] which commonly occur in the Itihasas.

A fourth direction is to allow more user control over our program's execution flow. Our program allows this already to a limited extent by allowing the user to disable certain optional

[9]This form is allowed by 7.1.38 (**त्त्वाडपि छन्दसि**), but our program treats such a derivation as Vedic and does not model that the form is also allowed in the Itihasas.

rules. With tighter code discipline, we might be able to extend this behavior to all other rules as well. Our most ambitious idea in this direction is to use Rust's rich macro system to inspect our existing rules and reorder them according to whatever criteria the user desires. Unfortunately, we expect that supporting this functionality properly would require a near-total rewrite of our code and its rules, which means that this direction is probably best served by an entirely new project.

9 Conclusions

This paper has presented a new Pāṇinian word generator that we developed by focusing first on producing a strong program. Our generator's speed and performance support a large test suite, which in turn permits faster progress on implementing rules. Over time, we expect to continue increasing our program's test coverage and improving the quality and range of its output.

Despite its limitations, we believe that `vidyut-prakriya` represents a major step forward for Sanskrit word generators, and we look forward to creating a program that generates all Pāṇinian words.

Acknowledgements

The authors wish to thank Neelesh Bodas for feedback on specific *prakriyā*s, and for his work on the invaluable `ashtadhyayi.com`; Dhaval Patel for guidance on the design of SanskritVerb; Madhav Deshpande for his comments and guidance; Shreevatsa Rajagopalan for his work on `vidyut-prakriya`'s WebAssembly build and online demo, and for help and advice in preparing this paper; Vikram Bhaskaran, Prasanna Venkatesh T S, Yash Khasbage, and Sourya Kakarla for their other contributions to `vidyut-prakriya`; and Shruthi Raghuraman for her love and support. All errors are my own.

References

Akshar Bharati, Amba Kulkarni, V Sheeba, and Rashtriya Vidyapeetha. 2006. Building a wide coverage Sanskrit morphological analyzer: A practical approach. 01.

Neelesh Bodas. 2023. Ashtadhyayi. `https://ashtadhyayi.com`. Accessed: 2023-09-28.

William Bugden and Ayman Alahmar. 2022. Rust: The programming language for safety and performance.

Andrew Gallant. 2015. Index 1,600,000,000 keys with automata and Rust. `https://blog.burntsushi.net/transducers/`. Accessed: 2023-10-01.

Pawan Goyal and Gerard Huet. 2016. Design and analysis of a lean interface for Sanskrit corpus annotation. *Journal of Language Modelling*, 4(2):145–182, Oct.

Pawan Goyal, Amba Kulkarni, and Laxmidhar Behera. 2009. Computer simulation of Aṣṭādhyāyī: Some insights. In Gérard Huet, Amba Kulkarni, and Peter Scharf, editors, *Sanskrit Computational Linguistics*, pages 139–161, Berlin, Heidelberg. Springer Berlin Heidelberg.

Oliver Hellwig. 2009. SanskritTagger: A stochastic lexical and POS tagger for Sanskrit. In Gérard Huet, Amba Kulkarni, and Peter Scharf, editors, *Sanskrit Computational Linguistics*, pages 266–277, Berlin, Heidelberg. Springer Berlin Heidelberg.

Anand Mishra. 2009. Simulating the Pāṇinian system of Sanskrit grammar. In Gérard Huet, Amba Kulkarni, and Peter Scharf, editors, *Sanskrit Computational Linguistics*, pages 127–138, Berlin, Heidelberg. Springer Berlin Heidelberg.

Dhaval Patel and Shivakumari Katuri. 2015. Prakriyāpradarśinī - an open source subanta generator. 16th World Sanskrit Conference.

Dhaval Patel. 2023. SanskritVerb. `https://github.com/drdhaval2785/SanskritVerb`. Accessed: 2023-10-01.

S. Prasanna. 2022. Spellchecker for Sanskrit: The road less taken. In Md. Shad Akhtar and Tanmoy Chakraborty, editors, *Proceedings of the 19th International Conference on Natural Language Processing (ICON)*, pages 290–299, New Delhi, India, December. Association for Computational Linguistics.

Jivnesh Sandhan, Rathin Singha, Narein Rao, Suvendu Samanta, Laxmidhar Behera, and Pawan Goyal. 2022. TransLIST: A transformer-based linguistically informed Sanskrit tokenizer.

Peter M. Scharf and Malcolm D. Hyman. 2012. Linguistic issues in encoding Sanskrit.

Peter Scharf. 2015. An XML formalization of the Aṣṭādhyāyī. 16th World Sanskrit Conference.