Digital archiving of Malayalam content

C. V. Radhakrishnan

Abstract

This article will detail a workflow that leverages TEX and associated tools to produce multiple outputs — including (i) TEI XML, (ii) HTML, (iii) print-ready PDF, and (iv) smartphone-optimized PDF — all from the same LATEX-encoded source material in Malayalam, a language spoken in Kerala, India.

For the past fourteen years, Sayahna Foundation, a non-profit, has used this workflow to archive heritage books encoded in TEI XML and to generate HTML and PDF versions for immediate user access. Initially, TEX4ht did not support Indic scripts, so we developed an alternative method to convert LATEX content into other markup formats. Our approach involves typesetting the content to create a PDF output closely resembling a TEI XML document, then extracting the text to produce fully parsable XML output. The HTML version is then derived from these XML sources using custom XSLT.

This article aims to provide a detailed overview of this workflow, which may be beneficial for those interested in archiving Indic content in a future-proof, neutral format. Additionally, we will discuss a few limitations of the process and outline TODO items for potential improvements and collaborations.

1 The state of digital archiving of Indic content

The state of digital archiving in India is currently quite poor, indicating a significant gap in recognizing the importance of this crucial practice. A lack of understanding among the general population is evident. More worryingly, leading institutions such as governmental bodies and universities, which one would expect to champion digital archiving efforts, like their counterparts in nations with established successful systems, have thus far not prioritized its necessity. There seems to be a dominant viewpoint that printed material represents the definitive culmination of written content's life cycle. This belief has directed content processing technologies toward traditional typesetting and printing methods. As a result, outdated systems like PageMaker remain in active and widespread use across many regions of India.

Several crucial aspects are often overlooked, including the long-term preservation of source material, its accessibility, and its re-usability for future generations using evolving devices and technologies of the future. Additionally, the separation of content's structure from its format [18] is rarely considered, even by major publishers. Furthermore, there is a general disregard for neutral, device-, operating system-, and application-independent data formats.

The adoption of markup languages globally has revolutionized the preservation and re-usability of source material within the publishing industry. Archival formats such as the Standard Generalized Markup Language (SGML) [6, 20] and its more prevalent derivative, the Extensible Markup Language (XML) [5], have enabled publishers worldwide to archive content in future-proof, reusable formats, prioritizing structure over aesthetic formatting. However, this advancement has yet to significantly impact the Indian publishing industry, leaving it vulnerable to content loss during natural disasters. This risk was tragically realized in the 2018 Kerala floods, where many libraries and individuals lost irreplaceable collections of rare publications.

1.1 Archiving of Malayalam content

Kerala's archiving history mirrors that of the rest of India. Across the state, archiving has not been prioritized, a fact poignantly illustrated by the revised release of the Malayalam translation of *Hortus Malabaricus* [17], a 17th-century 12-volume Latin botanical treatise, documenting the flora of the Malabar coast. The first edition, published ten years prior, had its source material stored in obsolete PageMaker files. These files proved incompatible with the systems used for the revised edition, forcing a substantial effort to re-enter all the data and repeat the entire editorial process. Regrettably, the publisher still appears to overlook the critical need for preserving source material in future-proof data formats.

This context led Sayahna Foundation [13], a Trivandrum-based non-profit, to initiate action with the primary goal of showcasing best archiving practices and their inherent benefits: universal access, software freedom, re-usability, and the long-term preservation of sources. Additionally, the Foundation aimed to distribute software to the user community. As one might expect, Sayahna Foundation has consistently relied on TEX and its related infrastructure as its default typesetting system, enabling the development of a workflow that allows for the generation of multiple output formats from the same TEX source files.

2 Converting LATEX to archival formats: approaches and challenges

Our default typesetting system, LATEX, is excellent for producing high-quality printed documents. However, establishing archival versions from these sources

doi.org/10.47397/tb/46-2/tb143radhakrishnan-malayalam-archiving

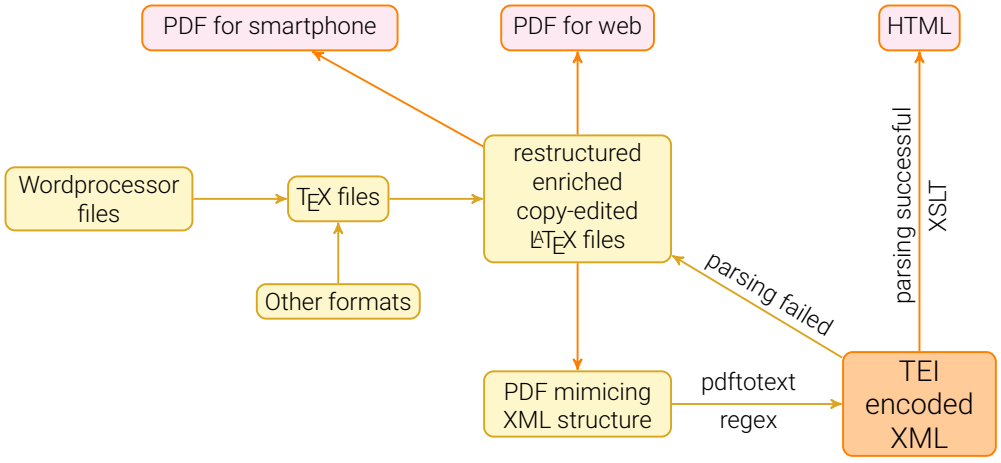

Figure 1: The schematic workflow diagram.

presents a unique set of hurdles. The challenge lies in LaTeX's sophisticated macro language and authors' frequent use of complex programming, which makes accurately translating these documents into other markup formats a formidable task. This is especially true for XML, given its distinct grammar and diverse DTDs. Therefore, our discussion will focus on developing a robust workflow to effectively transform these LaTeX files into formats suitable for long-term preservation and accessibility, often shifting from the visual presentation of a PDF to structured data formats like XML that can be readily parsed, indexed, and repurposed for future technologies.

Generally, three main approaches address this conversion problem:

Parsing and emitting XML: This method involves directly parsing the LaTeX document and generating XML. Examples include `LaTeXML` [9] and `Tralics` [8]. This approach often requires developing a custom TeX implementation to handle the vast array of LaTeX functions and libraries, a monumental undertaking that can still encounter surprises with cleverly authored documents.

DVI-based extraction: This approach leverages the TeX engine. It involves injecting custom elements and attributes as `\special` commands into the DVI output, which are then extracted by a post-processor. `Hermes` [1] and `TeX4ht` [15] are examples of this method. This significantly reduces the burden of processing complex author macros and LaTeX package functions, requiring only a basic customization layer in the LaTeX macro language tailored to the target XML DTD.[1]

Variant markup with XSLT: This method involves creating a markup variant that mirrors LaTeX syntax, then post-processing it with XSLT[2] to generate either XML or LaTeX. Examples include `tbook` [2] and `TeXML` [11]. This approach is best suited for authors who want to produce both PDF output via LaTeX and XML/MathML for online delivery from the same source, but it's generally not ideal for high-volume text processing houses.

Each approach has its own merits and drawbacks. Based on experience, the second option, particularly using `TeX4ht`, stands out. `TeX4ht` offers exceptional opportunities, hooks, and tricks to process even the most complex LaTeX documents and generate any type of custom XML/MathML.

Unfortunately, we couldn't leverage `TeX4ht` for our transformation of Malayalam content due to its lack of support for Indic scripts at the time we began our work. This led us to adopt an alternative workflow, capitalizing on our existing expertise and familiarity with TeX.

3 The workflow

The schematic workflow, detailed in Figure 1, outlines our data acquisition and processing. Source material is generated either through manual LaTeX entry, adhering to our data enrichment scheme, or by scanning printed books and applying Optical Character Recognition (OCR). While modern OCR offers

[1] A DTD (Document Type Definition) is a set of rules defining the valid structure and elements for an XML or SGML document, ensuring consistency and enabling validation. For more information: `en.wikipedia.org/wiki/Document_type_definition`.

[2] Extensible Stylesheet Language Transformations (XSLT) [21] is an XML-based language used, in conjunction with specialized processing software, for the transformation of XML documents.

impressive accuracy, the resulting text requires thorough cleanup — a combination of automatic and manual processes. This manual cleaning stage is crucial for standardizing the text, a prerequisite for automating both XML and PDF generation from the same source inputs. This standardization and cleanup represents the most labor-intensive component of our archiving workflow.

In a recent development, several living authors have expressed interest in open-access publishing, pursued by Sayahna Foundation, choosing to release digital versions of their works under a free license. Their aim is to engage younger readers, who often show little interest in printed media. These works are typically submitted in word processor formats, specifically Microsoft Word or LibreOffice Writer.

To prepare these submissions for our workflow, we convert them to LaTeX format using the `Writer2LaTeX` plugin [3]. This tool generally performs a good job. However, it's vital to select the `ultraclean LaTeX` mode during conversion, which prevents the filter from adding extraneous formatting code that attempts to mimic the original word processor's layout, ensuring a cleaner and more manageable LaTeX output.

Once the input data is prepared, we follow these steps in sequential order:

1. We use LuaTeX to generate PDF output from the source files. This PDF is designed to visually mimic the appearance of the final XML document.
2. The `pdftotext` [4] utility extracts plain text from the generated PDF.
3. A minimal cleanup is performed to remove the empty elements and to correct any unwanted ligature formations. These often occur due to the loss of Zero Width Non-Joiner (ZWNJ, U+200C) characters in certain consonant combinations during the text extraction process.
4. The extracted text is then parsed against the XML DTD to identify any potential errors.
5. If errors are found, the source text is edited to resolve the issues, and the entire process is repeated until a fully parsable and error-free XML document is obtained.

The entire conversion and output generation workflow, encompassing all the steps we've discussed, is fully automated through the use of a few well-defined targets and a simple `Makefile`. This setup performs exceptionally well, running like a breeze within our GNU/Linux environment. This automation not only significantly reduces manual effort but also ensures consistency and efficiency throughout the entire content processing pipeline.

3.1 The choice of XML DTD

When it came to selecting an XML DTD for encoding our textual data, we considered several leading options, including the National Library of Medicine (NLM) DTD, DocBook, the Elsevier Journal Article DTD, and the Text Encoding Initiative (TEI) DTD. After careful deliberation, we ultimately chose TEI [14].

Our decision was primarily driven by TEI's comprehensive nature and broad applicability across numerous disciplines. Unlike more specialized DTDs, TEI offers robust support for everything from simple articles to heritage content, critical studies, complex movie scripts, linguistic analyses, lexicons, and visual arts, including excellent support for technical content with heavy math requiring MathML encoding. This versatility is crucial for our diverse range of materials.

In numerous private discussions, a recurring concern voiced by many social scientists highlights a significant disconnect with technologists regarding the intricacies of digital archiving. Social scientists frequently observe that technologists are generally wary of recognizing and implementing the specialized features crucial for preserving the nuanced integrity of digital humanities sources.

For social scientists, the value of a digital archive extends far beyond simple digitization. They emphasize the need to capture a wealth of contextual and historical data that reflects the organic evolution of language and text. This includes, but is not limited to:

Detailed manuscript descriptions: Providing metadata about the physical characteristics, provenance, textual variations found in manuscripts obtained across different locations and historical context of the original manuscript.

Representation of semantic variations: Tracking how the meaning and usage of words and phrases have evolved over centuries, offering critical insights into cultural and intellectual shifts.

Linguistic variations: Documenting changes in spelling conventions across different historical periods and diverse geographic locations, which are vital for linguistic analysis and historical scholarship.

Evolution of character sets: Capturing the dropping or adoption of characters over the centuries, and crucially, determining how a dropped character is accurately represented and noted in contemporary digital renditions.

These elements are not mere technical details; they are fundamental to the research methodologies of social scientists, enabling them to conduct deep textual analysis, trace intellectual lineages, and understand historical transformations. Unfortunately, many of the Document Type Definitions (DTDs) commonly employed in digital archiving projects lack the sophisticated capabilities required to encapsulate these vital scholarly requirements. The Text Encoding Initiative (TEI) stands out as a notable exception, offering the rich, semantic markup necessary to address these complex needs, thereby bridging this critical gap between technological implementation and scholarly imperative.

Beyond its inherent flexibility, another significant factor in our selection of TEI was the extensive software infrastructure available for it. This ecosystem provides invaluable tools for generating TEI XML from various input sources and for transforming TEI-encoded XML into a multitude of human-readable outputs. Furthermore, TEI's strong affinity for and compatibility with LaTeX, largely thanks to the remarkable contributions of Sebastian Rahtz [10] during his tenure at the TEI Consortium, was a decisive advantage. This synergy streamlines our workflow significantly. We also plan to leverage the TEI DTD for our ambitious Malayalam Lexicon project, further solidifying its importance to our long-term goals.

3.2 The general structure of a TEI-encoded XML document

A TEI-encoded XML document has a standard, hierarchical structure designed for describing written texts in any discipline. At its highest level, a TEI document is contained within a single `<TEI>` root element.

Inside the `<TEI>` element, there are two primary components:

`<teiHeader>`: This is the metadata section of the document. It contains crucial information about the digital file itself and about the text being encoded. This includes details like the title, author, publication information, encoding methods used, revision history, and more.

`<text>`: This element contains the actual content of the document. It typically has a flexible structure that can include:

- `<front>`: For any front matter (e.g., title pages, prefaces, dedications).
- `<body>`: For the main content of the text, often organized into logical divisions like chapters and various section levels encoded in nested `<div>` elements, paragraphs (`<p>`), or lines (`<l>`) for poetry.
- `<back>`: For any back matter (e.g., appendices, bibliographies).

This clear separation of metadata from content, combined with TEI's rich set of elements for various textual features (names, places, dates, literary devices, formulae, graphic, tabular material, etc.), makes it highly versatile for scholarly encoding and analysis.

3.3 The metadata

The TEI DTD imposes a highly structured and comprehensive metadata requirement, meticulously organized within the `<teiHeader>` element. This header serves as the "virtual title page" of a TEI document, providing essential information about the digital file itself and the text it contains. While the TEI guidelines allow for great descriptive sophistication, certain child elements within the `<teiHeader>` are mandatory to ensure a minimal yet robust description.

Primarily, the TEI header is composed of five distinct components, though not all are strictly required for every valid TEI document:

`<fileDesc>`: This is the sole mandatory top-level component of the `<teiHeader>`. It provides a full bibliographic description of the electronic file. Within `<fileDesc>`, three sub-elements are themselves mandatory:

- `<titleStmt>`: Contains information about the title of the electronic text and those responsible for its creation (e.g., `<title>`, `<author>`, `<respStmt>`).
- `<publicationStmt>`: Details the publication of the electronic text, including distributors, publishers, and availability.
- `<sourceDesc>`: Describes the original source from which the electronic text was derived, if applicable (e.g., a printed book, manuscript).

`<encodingDesc>`: documents the relationship between an electronic text and the source or sources from which it was derived.

`<profileDesc>`: provides a detailed description of non-bibliographic aspects of a text, specifically the languages and sub-languages used, the situation in which it was produced, the participants and their setting.

`<xenoData>`: provides a container element into which metadata in non-TEI formats may be placed.

`<revisionDesc>`: summarizes the revision history for a file in the form of a list.

By adhering to this rigorous structure, the TEI DTD ensures that encoded texts are not only human-readable but also machine-actionable, facilitating long-term preservation, discoverability, and interoperability across diverse digital archiving projects. The source listing given in the Listing 1 exemplifies the exhaustive nature of the metadata.

Listing 1: The listing of the metadata in a TEI XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<TEI xmlns="http://www.tei-c.org/ns/1.0"
 xml:lang="ml">
 <teiHeader>
  <fileDesc>
   <titleStmt>
    <title type="full">
     <title xml:lang="ml" level="a"
      type="main">
      അക്ക മഹാദേവിയുടെ വചനങ്ങൾ
     </title>
     <title xml:lang="en" type="main">
      Akka Mahadeviyude Vachanangal</title>
     </title>
     <author>K. Satchidanandan</author>
     <respStmt>
      <resp>Data entered by</resp>
      <name>the author</name>
     </respStmt>
     <respStmt>
      <resp>Typeset by</resp>
      <name>JN Jamuna</name>
     </respStmt>
      <respStmt>
       <resp>Edited by</resp>
       <name>PK Ashok</name>
     </respStmt>
     <respStmt>
      <resp>Encoded by</resp>
      <name>JN Jamuna</name>
     </respStmt>
     <sponsor>Sayahna Foundation</sponsor>
     <funder>River Valley Technologies</funder>
   </titleStmt>
   <publicationStmt>
    <publisher>Sayahna Foundation</publisher>
    <pubPlace>Trivandrum, India</pubPlace>
    <date when="2022">October 18, 2022</date>
    <address>
     <addrLine>JWRA 34, Jagthy</addrLine>
     <addrLine>Trivandrum 695014</addrLine>
     <addrLine>India</addrLine>
    </address>
    <availability>
     <p>The text of the original item is copy-
      righted to the author...license info.</p>
    </availability>
   </publicationStmt>
   <sourceDesc>
    <biblFull>
     <titleStmt>
      <title type="full">
       <title xml:lang="ml" level="a"
        type="main">
       അക്ക മഹാദേവിയുടെ വചനങ്ങൾ</title>
       <title xml:lang="en" type="main">
        Akka Mahadeviyude Vachanangal</title>
      </title>
      <author>K. Satchidanandan</author>
     </titleStmt>
     <extent>
      <measure unit="pages" quantity="20">
       20 pages of source material</measure>
     </extent>
     <publicationStmt>
      <publisher>Sayahna Foundation</publisher>
      <pubPlace>Trivandrum, Kerala</pubPlace>
      <date>2020-10-24</date>
     </publicationStmt>
    </biblFull>
   </sourceDesc>
  </fileDesc>
 <profileDesc>
  <langUsage>
   <language ident="ml">Malayalam</language>
  </langUsage>
  <textClass>
   <keywords>
    <term>Poems</term>
    <term>K. Satchidanandan</term>
    <term>Akka Mahadeviyude Vachanangal</term>
     ...
   </keywords>
  </textClass>
  <textDesc n="poem">
   <channel mode="w">digital delivery</channel>
   <constitution type="single"/>
   <derivation type="original"/>
   <domain type="art"/>
   <factuality type="fact"/>
   <interaction type="complete"/>
   <preparedness type="prepared"/>
   <purpose type="inform" degree="high"/>
  </textDesc>
  <settingDesc>
   <setting>
    <name>Trivandrum, Kerala, India</name>
    <time>2020-10-24</time>
   </setting>
  </settingDesc>
 </profileDesc>
</teiHeader>
```

Given that the comprehensive metadata required for the `<teiHeader>` cannot be directly extracted from standard LaTeX input, we've implemented a specialized workflow. This involves creating an additional input document specifically for metadata, coupled with the development of a dedicated LaTeX package to process it. This package dynamically generates the necessary TEI header elements, populated with their legally admissible values.

While the TEI DTD allows for an incredibly exhaustive metadata description, our approach focuses on efficiency without compromising essential information. Therefore, our document metadata is not as expansive as the DTD permits; instead, it meticulously includes only the mandatory elements and a select few others critical for our archiving goals.

Specifically, our `<teiHeader>` includes:

`<fileDesc>`: This crucial element contains the title of the work in both English and Malayalam, the author's name, and a detailed `<respStmt>`. The `<respStmt>` lists the individuals responsible for various stages of the XML generation process, such as data entry, proofing, typesetting, and encoding, alongside sponsor and funder information.

`<publicationStmt>`: This section provides comprehensive publication details and crucial licensing information for the digital version.

`<sourceDesc>`: Here, we capture the full bibliographic information of the original content, complemented by details regarding the first publisher and the size of the original media.

`<profileDesc>`: This component provides essential context about the text itself, including language information, relevant keywords, and descriptive summaries of both the text's content and its original setting.

This streamlined yet robust metadata strategy ensures that our TEI-encoded documents meet the essential requirements for discoverability and long-term preservation, balancing thoroughness with practicality.

4 The packaging

Now, let's delve into the practical aspect of packaging our LaTeX libraries to execute the transformation process.

As detailed in our workflow (Section 3), our primary strategy involves creating PDF output that visually mirrors the structure of the target TEI XML document. This requires developing custom LaTeX packages to redefine standard LaTeX functions and macros.

```
<figure  rend="fcenter" type="gre">
<graphic url="images/akka-131.png"
width="170.71652pt" rendition="gre"/>
</figure>
<p style="indent">
</p>
<p style="indent"></p>
<lg xml:id="lg0.0.0.4">
<l> ചേറെന്റെ ദേഹം,</l>
<l> എൻ ദേഹിയാകാശവും:</l>
<l> ഏതെന്റെയാക്കണം,</l>
<l> ദേവ, ഞാനെങ്ങിനെ,</l>
<l> ആരായ് കരുതണം നിന്നെ?</l>
<l> എൻ തൃഷ്ണകൾ</l>
<l> നീളെത്തകർക്കുക,</l>
<l> ഹേ, മല്ലികാർജ്ജുനാ! </l>
</lg><!--end of "verse"-->
(വചനം 12)

<lg xml:id="lg0.0.0.5">
<l> പട്ടുനൂൽപ്പുഴു സ്വന്തം</l>
<l> മഞ്ജയിൽ നിന്നും വീടു</l>
<l> കെട്ടിട്ടും പോലെ, സ്വന്ത-</l>
<l> മുടലിൻ നാരിൽ ചുറ്റി-</l>
<l> ച്ചുറ്റിത്തൻ ശ്വാസം മുട്ടി-</l>
<l> പ്പിടഞ്ഞു ചാവും പോലെ</l>
<l> കത്തുന്നു ഞാനും, ജീവൻ</l>
<l> കൊതിക്കും കൊതികളാൽ.</l>
<l> വെട്ടുക പ്രഭോ, ജീവ-</l>
<l> തൃഷ്ണകൾ, പുറത്തേയ്ക്ക്</l>
<l> കാട്ടുക വഴി, ചെന്ന-</l>
<l> മല്ലികാർജ്ജുനദേവാ! </l>
</lg><!--end of "verse"-->
```

Figure 2: A typical example of a PDF page generated from LaTeX sources with XML elements typeset. The content is part of a poem. (A full XML version is available at: `books.sayahna.org/xml/satchi-akkamahadevi.xml`)

While tools like `TeX4ht` commonly embed XML elements directly into the output via TeX's `\special` primitive, we deliberately avoided this approach. Our rationale was pragmatic: to ensure the generated PDF serves as a valuable diagnostic tool. By keeping the PDF free of embedded `\special` commands, it functions as a clear visual representation of the document's inherent structure. This allows us to easily identify and locate problems when errors arise during XML parsing against the DTD. A quick glance at a specimen page from such a PDF (Figure 2) demonstrates this concept in practice, showing a visual guide that directly correlates to the intended XML structure.

In developing these LaTeX libraries, standardizing our input files and adhering to a specific LaTeX tagging pattern significantly simplified the process. However, we still encountered persistent challenges, particularly concerning the precise handling of paragraph elements (`<p>...</p>`) and the accurate splitting of chapters within a book, specifically, for standalone XML versions of each chapter with mandatory metadata and backmatter like a colophon. Addressing these structural nuances was crucial for faithful XML representation, and our custom LaTeX packages were designed to precisely control the output to achieve the desired TEI XML structure.

Let's examine how a section-level heading macro is redefined for our XML transformation. In a visually-oriented output format like PDF, a heading simply serves as a title, signaling the start of section to the reader. These headings typically feature visual attributes such as bold typefaces, increased vertical spacing, and other stylistic embellishments to catch the eye.

However, in an XML document, a section represents a logical segment of the document. This fundamental difference necessitates not only a starting element to mark its beginning, but also an end-of-section element to clearly delineate its logical conclusion.

4.1 Inserting end-of-section elements

Unlike LaTeX environments, which have explicit `\begin{environment}` and `\end{environment}` commands, the `\section` command doesn't require a corresponding `\endsection` macro. This means the block of text under a `\section` command is not delimited from a programmatic perspective. Several other LaTeX commands, like `\part`, `\chapter`, `\item`, etc., also share this characteristic.

To understand how we address this challenge for XML, first refer to the code listing in Listing 2, which illustrates the structure of section elements in our XML output, using `<div>...</div>`.

Listing 2: The listing of the section element in a TEI XML document.

```
<body>
 <div type="part">
  <head>Fallacies of Authority</head>
  <p>The subject of which is Authority in
    various shapes, and the object, to
    repress all exercise of the reasoning
    faculty.</p>
  <div n="1" type="chapter">
   <head>The Nature of Authority</head>
   <p>With reference to any proposed measures
    having for their object the greatest
    happiness of the greatest number [...]</p>
   <div n="1.1" type="section">
    <head>Analysis of Authority</head>
    <p>What on any given occasion is the
     legitimate weight or influence to be
     attached to authority [...] </p>
   </div>
   <div n="1.2" type="section">
    <head>Appeal to Authority, in What Cases
         Fallacious.</head>
    <p>Reference to authority is open to the
     charge of fallacy when [...] </p>
   </div>
  </div>
 </div>
</body>
```

The provided source listing for the `\section` macros (see listings 3–5) offers enough insight into how end elements are precisely inserted at the conclusion of each section segment within the document body.

Our approach involves defining a macro called `\EndSec`. This macro, in turn, calls another macro, `\__e_sec:` which is responsible for actually inserting the end-of-section elements. By default, `\__e_sec:` is initially defined to do nothing more than what `\EndPar` does.

Here's how it works: Every time a `\section` command is encountered, it first triggers an `\EndSec` call. This ensures that the end-of-section element is inserted before a new section begins. Consequently, when the very first `\section` command is executed, no end-of-section tags are inserted, as `\__e_sec:`'s default behavior is to do nothing.

However, after that initial execution of `\__e_sec:` which does nothing, the `\section` command redefines `\__e_sec:` to be `\e__sec_i:`. This redefined macro is designed to perform certain checks and, crucially, to insert the `</div>` tag — which marks the logical end of a section segment in our XML output. The code listing in Listing 3 will enlighten on this point.

Listing 3: Code listing of redefined section macros.

```
\bool_new:N \g_sec_bool
\bool_new:N \g_nosec_bool
\cs_new:Nn \__e_sec:
  { \EndPar \prg_do_nothing: }
\NewDocumentCommand \EndSec { } { \__e_sec: }
\cs_new:Nn \__e_sec_i:
  {
   \bool_if:NTF \g_subsec_bool
    { \__e_subsec:                }
```

```
   { \__e_par: \ParOff }
  \bool_gset_false:N \g_subsec_bool
  \__hb: \Tg</div> \__hb:
 }
```

The checks performed by the `\__e_sec_i:` command are *crucially important* for maintaining the precise logical structure of our XML output. This is because a new main section can begin while the currently active heading level might be a `subsection`, `subsubsection`, `paragraph`, or even a `subparagraph`.

To handle this, `\__e_sec_i:` is designed to recursively check the state of `\__g_subsec_bool`. If `\__g_subsec_bool` is true (indicating an active deeper sub-level), `\__e_subsec:` is invoked. This macro then performs its own checks on its immediate child heading, and this cascading process continues until a false condition is met — meaning all nested sub-elements have been properly closed.

This meticulous, recursive checking ensures the *correct number of* `</div>` *elements are inserted*. The count of these closing tags is precisely determined by the specific heading level that was active when the new main section was initiated, thereby guaranteeing a perfectly formed and logically delimited XML structure.

4.2 Inserting start-of-section elements

Now, let's discuss how we insert the necessary elements at the beginning of each section. Recall that LaTeX's `\section` command has two variants: *numbered* and *unnumbered* (starred) sections. Since XML often requires unique `id` attributes for its elements, we've designed two distinct start section macros to handle these variants: `\__b_sec:nn` for numbered headings and `\__b_starsec:n` for unnumbered ones. You can see their implementation in the macro code in Listing 4.

Listing 4: Code listing of redefined section macros (continued).

```
\cs_new:Npn \__b_sec:nn [#1] #2
 {
  \refstepcounter{section}
  \bool_gset_false:N \g_subsec_bool
  \bool_gset_true:N \g_sec_bool
  \__hb:
  \Tg<div ~type="section"~
    xml:id="sec\thesection"~n="\thesection">
  \__hb:
  \Tg<head ~type="sechead">#2\Tg</head>
  \__hb:
 }
\cs_new:Npn \__b_starsec:n #1
 {
  \refstepcounter{lsection}
  \__hb:
  \Tg<div ~type="lsection">
  \bool_gset_false:N \g_subsec_bool
   \bool_gset_true:N \g_sec_bool
  \__hb:
  \Tg<head ~type="lsechead">#1\Tg</head>
  \__hb:
 }
```

The primary purpose of these macros is twofold: to increment the relevant section counter and to insert the XML elements. For numbered sections, this involves adding a `<div type="section" xml:id="sec1">` element (where `sec1` would increment for each section), followed by the heading text itself. This heading text is then enclosed within `<head type="sechead" n="1"> ... </head>` tags. The unnumbered, or starred, variant functions almost identically, with one key difference: it uses a separate counter named `lsection` (mimicking the naming convention found in `TeX4ht`) instead of the standard section counter and does not add `xml:id`.

Listing 5: Code listing of redefined section macros (completed).

```
\RenewDocumentCommand \section { s o m }
 {
  \IfBooleanTF { #1 }
    {\starsection}
    {\nsection}
 }
\NewDocumentCommand \nsection { o m }
 {
  \EndDiv
  \__e_sec:
  \ParOff
  \__b_sec:nn [#1]{#2}
  \cs_set_eq:NN \__e_sec: \__e_sec_i:
  \__no_indent:
  \ParOn
 }
\NewDocumentCommand \starsection { m }
 {
  \EndDiv
  \__e_sec:
  \ParOff
  \__b_starsec:n { #1 }
  \cs_set_eq:NN \__e_sec: \__e_sec_i:
  \__no_indent:
  \ParOn
 }
```

Thus, the `\section` command determines whether to call `\starsection` or `\nsection` based on the presence of a `*` character appended to the call. Both `\starsection` and `\nsection` perform essentially the same core task, with `\nsection` having the additional capability to handle an optional argument passed to the original `\section` command. See the source listing in Listing 5.

Within their execution, these functions first invoke the `\EndDiv` command. This is crucial for properly closing any `<div>` element that might have been opened by a preceding `<div>` generating command before the new section is initiated; in the absence of any such, `\EndDiv` does nothing.

The `\EndDiv` call is immediately followed by a call to `\__e_sec:`, which handles the insertion of any necessary end-of-section elements, and then `\__b_sec:nn` to insert the start-of-section elements for the new section. In between, `\ParOff` is called to suppress the insertion of `<p>` elements that are illegal until the heading is closed.

Following these insertions, `\__e_sec:` is remapped to `\__e_sec_i:`, preparing it for subsequent checks. Finally, `\__no_indent:` is executed to add an attribute `style="noindent"` to the immediately-following `\par`, and then `\ParOn` is called. The `\ParOn` command is vital as it re-enables the insertion of `<p>` elements by subsequent `\par` commands within the new section, ensuring proper paragraph segmentation in the XML output.

The preceding explanation should provide a comprehensive understanding of how a non-delimited LaTeX command, such as `\section`, can be redefined to produce a properly delimited element structure in the XML output. This approach is critical; for instance, the initiation of `\backmatter` automatically triggers the closure of all open `<div>` levels, from `\chapter` down to `\subparagraph`, ensuring a complete and valid XML hierarchy for the `mainmatter`.

In contrast to these non-delimited segments, LaTeX environments (such as `\begin{verse} ... \end{verse}`) present a far simpler challenge for inserting corresponding start and end XML elements. Their inherent delimited tagging structure in the LaTeX document allows for straightforward redefinition. Consequently, all LaTeX commands and environments utilized in our documents are systematically redefined to produce the desired XML. For enhanced maintainability and ease of debugging, these redefinition packages are modularized, as detailed in the following list:

1. `ml-tei-back.sty`
2. `ml-tei-body.sty`
3. `ml-tei-fig.sty`
4. `ml-tei-front.sty`
5. `ml-tei-hooks.sty`
6. `ml-tei-meta.sty`
7. `ml-tei-tbl.sty`

All are loaded from the main package, `ml-tei.sty`.

4.3 From PDF to XML to HTML

Once the PDF is generated, we use the `pdftotext` binary [4] for text extraction, a tool that performs this task faithfully. The resulting text is then cleaned, removing unwanted line spaces, any empty elements (mostly paragraph elements), and correcting the occasional unwanted ligature formation. This cleaned text is subsequently parsed against the TEI DTD. While this process generally yields excellent results, errors do arise from time to time. In such instances, we address the errors in the LaTeX source documents and repeat the process until a perfectly parsed XML output is achieved.

For generating the HTML version, we employ a specially crafted XSLT stylesheet named `sfn-tei-html.xsl`, in conjunction with Saxon Home Edition version 10.5 [12]. Saxon is a comprehensive suite of tools designed for XML document processing. Its core component is a robust XSLT, XQuery, and XPath processor that can be operated via the command line or integrated into applications. Notably, Saxon fully implements the W3C's XSLT 3.0 Recommendation.

A screenshot of the HTML[3] generated from the XML sources provided in Figure 2 is displayed in Figure 3.

5 Generating diverse PDF outputs

The same LaTeX source document that drives our XML generation is also leveraged to create two distinct types of PDFs:

1. A regular PDF optimized for printing.
2. A smartphone-compatible PDF, specifically formatted to match the geometry of mobile screens, catering to modern reading habits.

While formatting a regular PDF is straightforward, the phone-compatible version warrants further explanation.

Our smartphone-optimized PDF features variable page heights. Crucially, page breaks occur only at the start of each new section, allowing sections to extend to any length. The sole limitation here is the maximum page length imposed by PDF specifications themselves. The package developed to achieve this smartphone compatibility owes a significant debt to

[3] The reader can view the live HTML page, and find links to the XML and PDF versions in the backmatter, at `books.sayahna.org/html/satchi-akkamahadevi.html`.

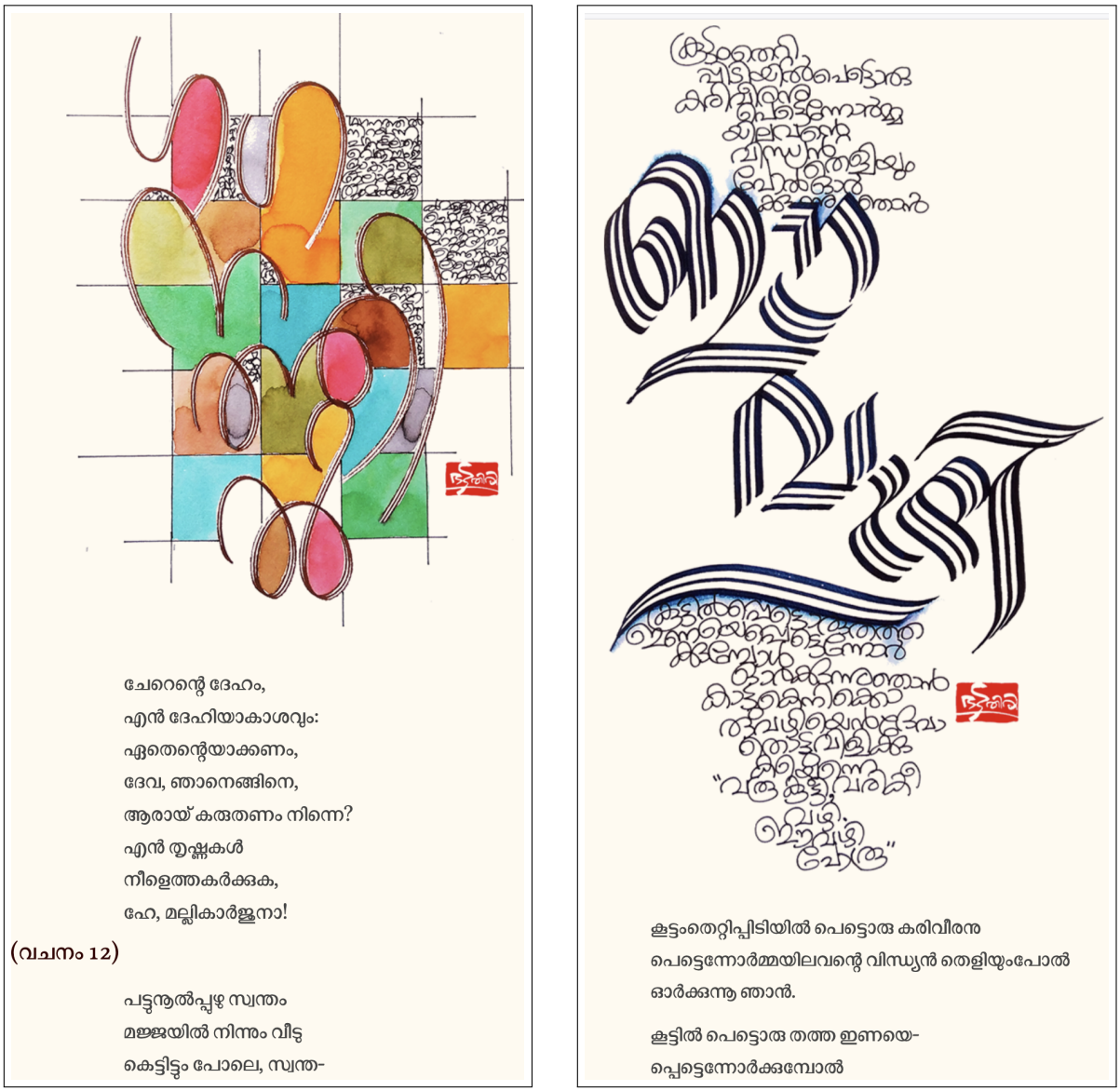

Figure 3: Screenshots of the HTML generated from the XML sources provided in Figure 2.

Boris Veytsman and Michael Ware [16, p. 263]. Their concept of an output routine that enables the typesetting of exceptionally long pages is a core feature we gratefully acknowledge, and it forms the bedrock of this advantageous capability. Beyond this innovative aspect, the formatting package for smartphone PDFs does not introduce any other notable complexities.

6 Future directions and community support

Currently, our XML generation workflow lacks support for mathematical content. For our primary focus on language, literature, and humanities, this isn't a significant limitation. However, it prevents us from archiving invaluable works like the 16th-century Malayalam astronomical treatise, *Yuktibhāṣā* [7]. This remarkable book, which predates Newton and Leibniz by two centuries in its discussion of time series and calculus [19], remains outside our archival reach due to this missing capability.

Furthermore, when our initial conversion filter was developed, the concept of *hooks* hadn't yet been introduced in LaTeX. As a result, our `\par` token processing relies on a less sophisticated approach which now requires a thorough rewrite. This also means our `ml-tei-hooks.sty` package needs a complete revisit.

The good news is that `TeX4ht` has recently gained the capability to process non-Latin scripts.

This development is pivotal, as it allows us to re-implement our conversion process using `TeX4ht`, specifically to leverage its excellent mathematical content processing features.

Finally, we've observed a few undesirable stray ligature formations during the text extraction process that warrant a thorough investigation.

These are critical areas for improvement, and any assistance from the user community in addressing these items would be greatly welcomed.

7 Data availability

All of the source code of the project along with basic user documentation is available at `gitlab.com/sayahna/archiving` under free software licenses, such as the LPPL.

References

[1] R. Anghelache. Hermes — a semantic XML+MathML+Unicode e-publishing/self-archiving tool for LaTeX authored scientific articles. 2006. Visited: 2025/08/07. `hermes.roua.org/`

[2] T. Bronger. The `tbook` System for XML Authoring. 2004. Visited: 2025/05/01. `tbookdtd.sourceforge.net/tbookman/index.html`

[3] The Document Foundation. `Writer2LaTeX`, a LaTeX/BibTeX export filter and front end for Writer. 2025. Visited: 2025/06/03. `extensions.libreoffice.org/en/extensions/show/writer2latex-1`

[4] Glyph & Cog, LLC. `pdftotext`. 2025. Visited: 2025/06/03. `www.xpdfreader.com/pdftotext-man.html`

[5] C. Goldfarb, P. Prescod. *Charles F. Goldfarb's XML Handbook*, ch. Why XML. Charles F. Goldfarb definitive XML series. Prentice Hall PTR, 2004.

[6] C. Goldfarb, Y. Rubinsky. *The SGML Handbook*, ch. A Structured Overview of SGML, pp. 121–214. Clarendon Press, 1990.

[7] Jyēṣṭadēvan. Yuktibhāṣā. 2020. `books.sayahna.org/ml/pdf/yukthibhasha.pdf`

[8] Marelle Team. `tralics`: A LaTeX to XML translator. 2017. Visited: 2025/05/23. `www-sop.inria.fr/marelle/tralics/`

[9] B. Miller, D. Ginev. `LaTeXML`: A LaTeX to XML/HTML/MathML Converter. 2024. Visited: 2025/05/10. `math.nist.gov/~BMiller/LaTeXML/`

[10] F. Mittelbach, J. Richmond (trans.). Sebastian Patrick Quintus Rahtz (13.2.1955–13.3.2016). *TUGboat* 37(2), 2016. `tug.org/TUGboat/tb37-2/tb116rahtz-mitt.pdf`

[11] O. Paraschenko. `TeXML`: An XML syntax for TeX (LaTeX, ConTeXt). 2004. Visited: 2025/05/10. `getfo.org/texml/index.html`

[12] Saxonica. What is Saxon? 2025. `www.saxonica.com/html/documentation10/about/whatis.html`

[13] Sayahna Foundation. 2013. Visited: 2025/06/01. `books.sayahna.org`

[14] TEI Consortium. TEI P5: Guidelines for Electronic Text Encoding and Interchange. 2025. Visited: 2025/05/31. `tei-c.org/release/doc/tei-p5-doc/en/Guidelines.pdf`

[15] TeX Users Group. `TeX4ht`. 2022. Visited: 2025/01/02. `tug.org/tex4ht/`

[16] B. Veytsman, M. Ware. Ebooks and paper sizes: Output routines made easier. *TUGboat* 32(3), 2011. Proceedings of TUG 2011. `tug.org/TUGboat/tb32-3/tb102veytsman-ebooks.pdf`

[17] Wikipedia. Hortus Malabaricus. 2006. Visited: 2025/05/01. `en.wikipedia.org/wiki/Hortus_Malabaricus`

[18] Wikipedia. Separation of content and presentation. 2017. Visited: 2025/05/31. `en.wikipedia.org/wiki/Separation_of_content_and_presentation`

[19] Wikipedia. Yuktibhāṣā. 2022. `en.wikipedia.org/wiki/Yuktibh%C4%81%E1%B9%A3%C4%81`

[20] World Wide Web Consortium (W3C). On SGML and HTML. 2018. Visited: 2025/04/02. `www.w3.org/TR/html4/intro/sgmltut.html`

[21] World Wide Web Consortium (W3C). XSL Transformations (XSLT), version 2.0 (second edition). 2021. Visited: 2025/06/02. `www.w3.org/TR/xslt20/`

◇ C. V. Radhakrishnan
River Valley Campus, Malayinkeezh
Trivandrum 695571, India
`cvr (at) river-valley (dot) org`
`http://river-valley.com`
ORCID 0000-0001-7511-2910