

**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING**

**THAPATHALI CAMPUS**

**Major Project Mid-Term Report**

**On**

**Unified Devanagari Rendering Engine for Nepali Language**

**Submitted By:**

|  |  |
| --- | --- |
| Amar Dura | [THA077BCT007] |
| Ayush Bhandari | [THA077BCT014] |
| Harish Joshi | [THA077BCT018] |
| Sugam Pokharel | [THA077BCT044] |

**Submitted To:**

Department of Electronics and Computer Engineering Thapathali Campus

Kathmandu, Nepal

January, 2025



**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING THAPATHALI CAMPUS**

**Major Project Mid-Term Report**

**On**

**Unified Devanagari Rendering Engine for Nepali Language**

**Submitted By:**

|  |  |
| --- | --- |
| Amar Dura | [THA077BCT007] |
| Ayush Bhandari | [THA077BCT014] |
| Harish Joshi | [THA077BCT018] |
| Sugam Pokharel | [THA077BCT044] |

**Submitted To:**

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

In partial fulfillment for the award of the Bachelor’s Degree in Electronics and Communication Engineering

**Under the Supervision of:**

Er. Praches Acharya

**Under the Co-supervision of:**

Er. Santa Bahadur Basnet

January, 2025

# ACKNOWLEDGEMENT

The project would not have been initialized if there was not the provision of the major project in the course structure. So, we want to be thankful to the department for providing the opportunity to start the project.

Throughout the project development and research, we had various difficulty and confusion. We want to be thankful to our supervisor Er. Praches Acharya sir and co-supervisor Er. Santa Bahadur Basnet sir for guiding us in solving the issues as well as continuously motivating us throughout the project.

Last but not the least, we want to express our gratitude towards our head of department Er. Umesh Kanta Ghimire sir and project coordinator Er. Saroj Shakya sir for any and all help and suggestions that have benefited us during the project.

|  |  |
| --- | --- |
| Amar Dura | THA077BCT007 |
| Ayush Bhandari | THA077BCT014 |
| Harish Joshi | THA077BCT018 |
| Sugam Pokharel | THA077BCT044 |

# ABSTRACT

The Nepali language, primarily spoken in Nepal and by minority communities in India, Bhutan, and Myanmar, is written in the Devanagari script with some distinct local typographic standards. With the rise of digital technologies, there is an increasing demand to digitize Nepali literature and documents. PDF generation plays a critical role in this process, as PDF remains the most widely used document format worldwide. However, generating PDFs for Nepali in Devanagari script faces challenges due to the lack of open-source systems capable of correctly rendering complex Devanagari text. This project addresses this gap by enhancing the Apache PDFBox library, enabling accurate rendering of Nepali text in Devanagari script. The improvements focus on ensuring compatibility with Unicode-compliant OpenType (OTF) fonts, resolving longstanding issues with older Nepali fonts that do not adhere to current OpenType standards. Additionally, the project provides recommendations for font designers and developers to align Devanagari fonts with modern typographic standards, fostering better digital representation of Nepali texts.

Keywords: *Devanagari Script, OpenType , PDF, Rendering Engine, Unicode Standard*

# TABLE OF CONTENTS

[ACKNOWLEDGEMENT iii](#_Toc188615958)

[ABSTRACT iv](#_Toc188615959)

[TABLE OF CONTENTS v](#_Toc188615960)

[List of Figures viii](#_Toc188615961)

[List of Tables ix](#_Toc188615962)

[List of abbreviations x](#_Toc188615963)

[1. INTRODUCTION 1](#_Toc188615964)

[1.1 Background 1](#_Toc188615965)

[1.2 Motivation 4](#_Toc188615966)

[1.3 Problem Definition 5](#_Toc188615967)

[1.4 Objectives 5](#_Toc188615968)

[1.5 Application and Scope 6](#_Toc188615969)

[1.5.1 Applications 6](#_Toc188615970)

[1.5.2 Scope 7](#_Toc188615971)

[1.6 Organisation of Project Report 8](#_Toc188615972)

[2. LITERATURE REVIEW 9](#_Toc188615973)

[3. REQUIREMENT ANALYSIS 14](#_Toc188615974)

[3.1 Project Requirement 14](#_Toc188615975)

[3.2 Feasibility Analysis 14](#_Toc188615976)

[3.2.1 Technical Feasibility 14](#_Toc188615977)

[3.2.2 Operational Feasibility 14](#_Toc188615978)

[3.2.3 Economic Feasibility 15](#_Toc188615979)

[4. THEORETICAL BACKGROUND 16](#_Toc188615980)

[4.1 TrueType Fonts (TTF) 16](#_Toc188615981)

[4.2 OpenType Font(OTF) 16](#_Toc188615982)

[4.2.1 Differences from TTF 17](#_Toc188615983)

[4.2.2 GSUB table 20](#_Toc188615984)

[4.2.3 GPOS table 22](#_Toc188615985)

[4.3 Rendering Rules 27](#_Toc188615986)

[4.3.1 Rules for Rakaar form 28](#_Toc188615987)

[4.3.2 Rules for Ligatures 30](#_Toc188615988)

[4.4 Unicode 32](#_Toc188615989)

[4.4.1 Unicode Support for Devanagari Script 33](#_Toc188615990)

[4.4.2 Font Use in the Case of Nepal 35](#_Toc188615991)

[4.4.3 TTX 35](#_Toc188615992)

[4.4.4 FontForge 38](#_Toc188615993)

[4.4.5 FontDrop 40](#_Toc188615994)

[4.5 Portable Document Format (PDF) 40](#_Toc188615995)

[4.5.1 Font Embedding 42](#_Toc188615996)

[4.6 Language & Library 43](#_Toc188615997)

[4.6.1 Java 43](#_Toc188615998)

[4.6.1 JVM 44](#_Toc188615999)

[4.6.2 pdfBox 44](#_Toc188616000)

[5. SYSTEM ARCHITECTURE AND METHODOLOGY 46](#_Toc188616001)

[5.1 System Architecture 46](#_Toc188616002)

[5.2 Methodology 48](#_Toc188616003)

[6. IMPLEMENTATION DETAILS 56](#_Toc188616004)

[6.1 Setting font 56](#_Toc188616005)

[6.2 Set text 56](#_Toc188616006)

[6.3 Glyph Substitution 57](#_Toc188616007)

[6.4 Algorithm for implementing OpenType Shaping for Devanagari Text 58](#_Toc188616008)

[6.5 Regex implementation 59](#_Toc188616009)

[7. RESULTS AND ANALYSIS 61](#_Toc188616010)

[7.1 Word List Samples 61](#_Toc188616011)

[7.2 Unicode Tokenizer Outputs 61](#_Toc188616012)

[8. REMAINING TASK 66](#_Toc188616013)

[8.1 Application of GPOS table 66](#_Toc188616014)

[8.2 Testing & validation 66](#_Toc188616015)

[8.3 Web Interface 66](#_Toc188616016)

[APPENDICES 67](#_Toc188616017)

[Appendix A: Gantt Chart 67](#_Toc188616018)

[Appendix B: Project Budget 68](#_Toc188616019)

[Appendix C: Rules for Rendering Devanagari, The Unicode Standards 69](#_Toc188616020)

[Appendix D: Test Data Details 72](#_Toc188616021)

[Appendix E: State transitions of Unicode encoded text tokenizer 73](#_Toc188616022)

[Appendix F: Character labels for Unicode encoded tokens 74](#_Toc188616023)

[Appendix G: Lookup Tables in OpenType Fonts 75](#_Toc188616024)

[Appendix H: Glossary 76](#_Toc188616025)

[Appendix I : Font Specification and Recommendations 78](#_Toc188616026)

[REFERENCES 79](#_Toc188616027)

# List of Figures

[Figure 1. 1 : Independent Vowels and Diacritics on the letter 'क'. 2](#_Toc188614631)

[Figure 1. 2 : List of Devanagari consonants used in Nepali 2](#_Toc188614632)

[Figure 1. 3 : Consonant ’b’ (’ब ’) with its diacritics 3](#_Toc188614633)

[Figure 1. 4 : Some common ligatures 3](#_Toc188614634)

[Figure 4. 1 : Some Conjunct forms in Lohit Devanagari font 17](#_Toc188615856)

[Figure 4. 2 : Glyf example 18](#_Toc188615857)

[Figure 4. 3 : cmap example from Noto Sans Devanagari 18](#_Toc188615858)

[Figure 4. 4 : Kerning table for Noto Sans Devanagari 20](#_Toc188615859)

[Figure 4. 5 : Organization of GSUB table 21](#_Toc188615860)

[Figure 4. 6 : GPOS table format 23](#_Toc188615861)

[Figure 4. 7 : OTF tables hierarchy 25](#_Toc188615862)

[Figure 4. 8: Conversion of fontfile into ttx format 36](#_Toc188615863)

[Figure 4. 9: xml view of fontfile 37](#_Toc188615864)

[Figure 4. 10: FontForge cmap table 39](#_Toc188615865)

[Figure 4. 11 : FontForge GPOS Table 39](#_Toc188615866)

[Figure 4. 12 : Graph of usage of PDFs compared to other document formats. 41](#_Toc188615867)

[Figure 5. 1: System Block Diagram 46](#_Toc188614904)

[Figure 5. 2 : DFD Level -0 48](#_Toc188614905)

[Figure 5. 3 : DFD Level -1 49](#_Toc188614906)

[Figure 5. 4 : GSUB Application Algorithm 50](#_Toc188614907)

[Figure 5. 5 : PDF Generation Algorithm 51](#_Toc188614908)

[Figure 5. 6 : PDF Rendering Algorithm 52](#_Toc188614909)

[Figure 5. 7 : State Diagram 53](#_Toc188614910)

[Figure 5. 8 : TTF Parser 54](#_Toc188614911)

[Figure 5. 9 : Use Case Diagram 55](#_Toc188614912)

[Figure 9. 1 : Gantt Chart 67](#_Toc188616240)

[Figure 9. 2 : State diagram for syllable tokenizer 73](#_Toc188616241)

[Figure 9. 3: Lookup Tables in OpenType Fonts 75](#_Toc188616242)

# List of Tables

[Table 9. 1 : Project Budget 68](#_Toc188615593)

# List of abbreviations

ASCII American Standard Code for Information Interchange

CV Consonant-Vowel

GPOS Glyph Positioning

GSUB Glyph Substitution

ISCII Indian Script Code for Information Interchange

JVM Java Virtual Machine

JWNZ Zero-Width Non-Joiner

JWZ Zero-Width Joiner

OTF OpenType Font

PDF Portable Document Format

TTF TrueType Fonts

UTF Unicode Transformation Format

# 1. INTRODUCTION

## 1.1 Background

Nepali is the most widely spoken language in Nepal. In addition to being the language of formal use all across the country, it is spoken by substantial minority communities in India, Bhutan and Myanmar along with diaspora communities all across the world. Linguistically, it falls on the Eastern Pahari branch of the Indo-Aryan languages and shares much of vocabulary as well as grammatical structure with other Pahari languages like Kumaoni and Garhwali. The language has evolved through centuries of cultural exchanges, absorbing influences from various neighboring languages, including Tibetan, Persian, and English. It holds a unique position in the South Asian linguistic landscape, bridging the languages of the Indian subcontinent with those of the Himalayas.

Although people sometimes use the Latin alphabet to write Nepali, especially for formal purposes, it is written primarily in Devanagari script. Devanagari is the most commonly used script all over northern India. Like with all other Indian scripts, it is descended from Brahmi script, that was first used by Emperor Ashoka of the Maurya dynasty in 3rd century BCE. As the script evolved, it is difficult to point to exact time frame for when the Devanagri script was first developed in distinction from earlier script forms as they are more in a spectrum than neatly divided boxes. Early forms of the script whose letters are visually close to Devanagari can be found as early as the 2nd century CE in Junagadh Rock Inscription of the Western Satrap Rudradaman. With more certainity some inscriptions of the imperial Rashtrakutas of Malkhed from the 8th century CE are identified as early form of Devanagari. From the 10th century onwards, it became the premier script to write most languages in northern India, a position it has retained since. Like all other scripts derived from Brahmi, Devanagari is an abugida. An abiguda is a type of segmental writing system in which a secondary vowel notation (often as a diacritic mark) is added to the primary consonant letter. Thus, an abugida (also called alphasyllabary or neosyllabary) differs from an alphabet (or alphabetic script) in which both the vowel and the consonant letters are of equal significance. The Latin script is an alphbetic script. Thus, in Latin the vowel ‘a’ and consonant ‘b’ are equal. The sequence of letters in Devanāgarī, as with most Brāhmic scripts, follows phonetic principles that

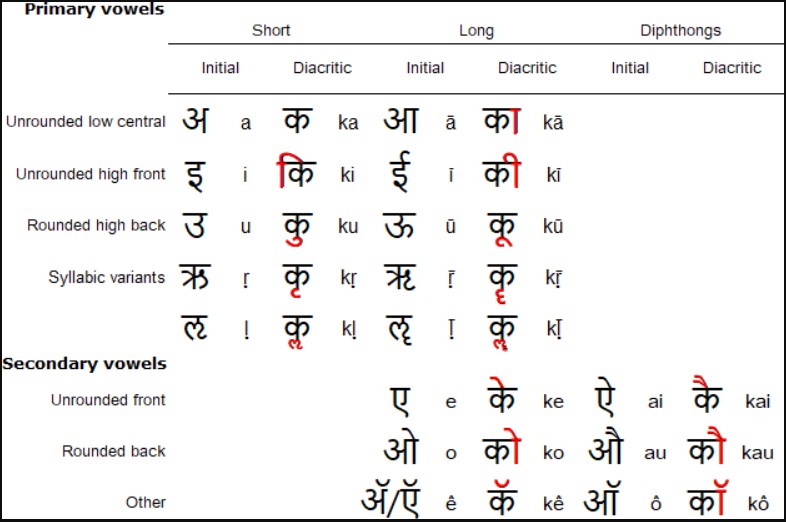


Figure 1. : Independent Vowels and Diacritics on the letter 'क'.

account for both the manner and place of articulation of the consonants and vowels they denote.The writing system encompasses both alphabetic and syllabic components, where vowels function independently and consonants combine with vowels to form consonant-vowel (CV) letters. In the simplest form, one consonant combines with one vowel to create a simple CV letter. The vowel ’a’ (’अ’) is usually assumed in the absence of diacritics. The following table of CV letters represent all the consonants in Nepali language as written in Devanagari script.

A table of different languages

Description automatically generated with medium confidence

Figure 1. : List of Devanagari consonants used in Nepali

When a CV letter has a vowel another than ’a’ ('अ'), diacritics are used on the letter to denote the specific vowel or diphthong, as can be seen from the following illustration using the letter ’b’ ('ब').

A screenshot of a computer

Description automatically generated

Figure 1. : Consonant ’b’ (’ब ’) with its diacritics

In an alphabetic writing system, the letters would be placed one by one without any modification from the base letter glyph. But in an abugida like Devanagari, the shapes of the letters can change depending upon the presence or absence of other letters before and after it. Thus, there are various combined characters and ligatures that are formed from the characteristics. While sometimes the combined characters can be visually similar to the base characters that were combined, oftentimes they are pretty different and not obvious at first.

Some examples of common ligatures are: The formation of composite characters

eA screenshot of a computer

Description automatically generated

Figure 1. : Some common ligatures

through the combination of multiple characters results in complex conjuncts, for example:

ङ् + ग = ङ्ग न् + न = न्न

ष् +ट् +र = ष्टर्

त् + र = त्र

ङ् + क् + ष = ङ्क्ष

With the exponential development of computer systems in the late twentieth century, it is but normal that the Nepali language speakers should be interested in using

Devanagari script and Nepali language using computer and over the internet. The initial development of Devanagari fonts for Nepali language was spurred largely by individuals working independently. While this gave Nepali speakers a useful way of writing Nepali, this also came with some problems. As individual fonts use different encoding methods (numeric codes that are mapped to character images or glyphs), they are naturally incompatible with each other. An additional problem is the lack of support in different devices and limited use and support. These problems were largely overcome by the use of Unicode based fonts.

Our work focuses on the standardized glyph ordering mechanism for writing systems in Devanagari script and the development of an open source unified rendering engine for Nepali language. This project aims to provide a reliable solution for accurate text rendering within the Java Virtual Machine (JVM) using the Apache PDFBox library, thereby enhancing the accessibility and usability of Nepali text across digital platforms and applications.

## 1.2 Motivation

The motivation behind our project arises from the critical need to address the challenges of accurately rendering Nepali text in the digital landscape. Despite Nepali’s rich linguistic heritage, the transition to multi-byte Unicode systems for Devanagari script has introduced complexities in character rendering, leading to inconsistencies across various applications and platforms. These rendering issues hinder the readability and usability of Nepali text, posing significant barriers to communication, education, and access to information for Nepali speakers worldwide.

A significant motivation for this project is the language barrier faced by many Nepali speakers who cannot read English. This becomes particularly problematic when essential information, household bills, documents, and printouts are available only in English. By standardizing the glyph ordering mechanism and developing an open-source unified rendering engine specifically for the Nepali language, we aim to overcome these challenges. Our goal is to empower users with a reliable, consistent, and accessible platform for creating, sharing, and interacting with Nepali text across digital medium.

## 1.3 Problem Definition

Our project addresses the deficiency of open-source unified technology within the Nepali writing system, which leads to inconsistencies in conjunct formation and character rendering across different applications. Current rendering engine of PDFBox fail to correctly display composite characters(द्य→द्य) and reorder glyphs(िक→क ि◌ ), resulting in differences in text presentation. Additionally, discrepancies in UTF codes across fonts further compound the issue, causing characters to be incorrectly displayed or even missing entirely(like क).

Despite the existence of closed-source rendering engines from major companies like Google and Microsoft, there remains a conspicuous absence of open-source solutions tailored specifically to these rendering challenges. This gap hampers the accessibility and usability of Nepali text in digital environments, hindering effective communication and information dissemination.

Therefore, the implementation of an open-source rendering engine within the Java Virtual Machine (JVM), utilizing the Apache PDFBox library, is imperative. By leveraging Apache PDFBox, an established open-source platform, the project aims to fill this void and provide developers with a reliable tool for accurately rendering Nepali text within PDF documents. This initiative not only promotes accessibility and inclusivity but also fosters collaboration and innovation within the open-source community, ensuring the continued improvement and development of solutions tailored to the Nepali language

## 1.4 Objectives

The main objectivess of the project is listed below:

1. To implement glyph ordering mechanism and standardize composite character representation
2. To develop an open source unified Devanagari rendering engine for JVM using Apache PDFBox

## 1.5 Application and Scope

### 1.5.1 Applications

The unified Devanagari rendering engine for the Nepali language is a versatile tool used in education, government, business, and cultural projects where creating PDFs with Nepali text is important. It helps in making accurate educational materials, government communications, marketing materials, and financial reports in Nepali. Overall, it’s a handy solution for anyone needing to work with Nepali text in PDF document. Although it is not possible to list all prospective fields of application, some more specific examples of the applications where the Devanagari rendering engine will be useful are as follows:

#### 1.5.1.1 Bill Generation

In many desktop and mobile application, some sort of PDF document is generated to assure the user that the transaction has been successfully performed. Some examples may include : Result score-sheet generation, bill generation, etc. With the widespread use of Mobile Banking and other payment gateways like eSewa, Khalti, etc, the issue of bill generation is not theoretical but something that affects a large number of people on a daily basis. Despite Nepali being the day to day language, the bills generated in these applications are almost invariably in English. The reason for this is mainly the difficulting in PDF rendering for Nepali in Devanagari correctly. The unified Devanagari rendering engine is perfectly suited for this task as most mobile applications are based, whether directly or indirectly, in Java Programming language.

#### 1.5.1.2 Media Monitoring Tools

In our modern digital world, various stakeholders (both governmental departments and ministries as well as non-governmental organizations like UNICEF, World Bank, etc) need to be constantly updated with the latest trends and advances related to their particular line of work and specialization. Similarly, major organizations generally need to track their appearance of newspapers(along with other media) to collect the data that will used for further analysis. For this, the written text is generally collected as the Web Versions of news may be subsequently updated, archived or be unavailable. As the volume of such news data is very large , it would not only be inefficient to collect those manually, but also incurs high cost and resources to do so. If a unified rendering engine for Nepali language as written in Devanagari script is used, such burdensome process can be greatly streamlined and simplified. This project focuses on rendering the scrapped news articles into PDF format, facilitating efficient distribution and dissemination of information.

#### 1.5.1.3 Text Processing Pipeline for AI

With the explosion of Large Language Models (LLM) within the past decade, it is but natural that people would want to enjoy the benefits of ML services based on Nepali language. This, naturally, requires a large amount of textual data before any further step can be taken on the ML training pipleline. In Nepali however, the preponderance of various mutually incompatible fonts and representation makes this difficult, not only in preprocessing the collected corpus but in collecting a large enough , quality corpus in the first place. The Unified Devanagari Rendering Engine would be greatly useful in the preparation of suitable Nepali corpus for training of Large Language Models.

#### 1.5.1.4 Governmental Outreach

Governmental agencies frequently interact with citizens through official documents, announcements, and reports. By adopting the rendering engine for government channels, such as websites and digital portals, governmental bodies can ensure accurate and effective communication with Nepali-speaking constituents. This fosters transparency and accessibility in governance, promoting citizen engagement and trust.

### 1.5.2 Scope

The unified Devanagari rendering engine for the Nepali language is designed with the following scopes:

* Java Ecosystem: Primarily targeted at applications running on the Java Virtual Machine, including web applications, desktop software, and enterprise systems.
* Open-Source Community: Available to the open-source community, encouraging contributions, improvements, and widespread adoption.
* Cross-Platform PDF Rendering: Ensures that Nepali text is rendered correctly in PDF documents on any platform that supports JVM, including Windows, macOS, and Linux.

## 1.6 Organisation of Project Report

The material in this project report is organized into eight chapters. After this introductory chapter introduces this project along with problem definition, objectives, scope and applications, there is chapter 2. The chapter 2 contains the literature review of vital and relevant publications, pointing toward a notable research gap. Chapter 3 discusses about the requirements for our project. It also discusses about the feasibility of our project. Then comes Chapter 4 containing our system architecture and methodology of our project. Chapter 5 contains the implementation details that has been followed till now. The results of the project at the end of the first part is presented in chapter 7 along with brief analysis. The remaining tasks to fulfill the objectives of the project are listed in chapter 7. The last chapter is the Appendices.

# 2. LITERATURE REVIEW

Devanagari, as with most pre-modern scripts of the world, was primarily used to write manuscripts by hand until the rise of the printing press. Until then there was, of course, no need for typefaces as everything was done manually. Unlike in East Asia or Europe, printing press did not become common in India until the early 19th century. Although some European residents in India had tried to print books in Devanagari script as early as the early 1600s due to the intricacies of the script and lack of adequate resources, it was not until 1796 that a Devanagari typeface for the printing press was designed for the printing of ‘Grammer of the Hindustanee Language’ by John Gilchrist [1] [1] . Although early typefaces were often error-filled and aesthetically unpleasant, steady progress of typography over the next century lead to far better results.

With the rapid increase in the use of computerized system to perform a large range of tasks in the 1980s and 90s, the need for font systems that could handle not only the Latin script for which such systems were initially developed but also for other scripts. To this end, the development of early versions of Unicode (as a universal character set) in 1988 helped greatly.

The research on Indian typefaces, which can be extended to the Devanagari script used in the Nepali language, identifies several critical challenges in digital display and design. These include the complexity of hinting due to intricate letter structures, the labor intensive process of developing glyph positioning and substitution instructions, and the absence of a standardized letter form grammar and visual language. Additionally, the lack of anatomical nomenclature and a grid system further complicates font production, while legibility parameters remain under-researched. Addressing these issues is essential for developing a Unified Devanagari Rendering Engine for Nepali, aiming to enhance readability and visual quality in digital environments. Although the research on effect of technology on digital Devanagari typeforms by Kinnat Sóley Lydon is only tangentially related to the present issue of unification and standardization, it does help to illustrate an important cause of the lack of proper rendering of Devanagari typeforms : font designers restricting themselves to standards that were held due to technological limitations in previous times and which have now been overcome in digital systems. An example of this phenomenon is that many fonts contains only around 255 glyphs, which was the limit in earlier methods although complete inventory for Devanagari needs around 1000 glyphs. This is a problem mainly for older TTF based fonts rather than newer OTF based fonts.

The development of Devanagari fonts for Unicode was introduced first in 1990 in ‘Minutes of SC2/WG2 Meeting 18 in Munich’ and has been going on steadily since then. Early efforts on standardization of Unicode based Devanagari fonts for Nepali was performed by Nepal Font Standardization Committee, a project with supervision from the government, and described in their ‘White Paper’[2] . It described the character sets and encoding for fonts that were in common use at that time (like: Sabdatara, Annapurna, etc). They not only provided useful information for unique features of Nepali in Devanagari ( as opposed to other languages like Hindi, Marathi, etc) like Three conjuncts, use of Chandrabindu , etc but also released a standard to be followed by font designers and implementer containing information like sorting rules, glyph design,etc. As the project didn’t themselves release any standardized font of their own and as their standard have not been followed completely, the importance of the White Paper is slightly diminished. The widespread use of older TTF based fonts like Preeti mean the standard was not used in them anyway.

Pat Hall at Language Technology Kendra, Patan, Nepal reviewed the problems that Unicode possessed, even though it is said to accommodate the standardization of writing/representation of many languages around the world.[3] It focuses on the difficulties encountered while representing various languages used within Nepal. Moreover, it focuses on representation of Nepal Bhasa. Nepal Bhasa uses, more frequently, Ranjana and Prachalit Lipi. Although some work has been done to represent these scripts in computers, the representation has not been in standard forms. It also states that there are some hack fonts and encodings even in Unicode representation. An important topic that is brought in the paper is that due to various causes, the development of Unicode fonts have tended not towards the Unification of fonts , as was intended, but towards separation of fonts. Similarly, the problem of coding being done by activists with limited knowledge was discussed in the paper.

Santa B. Basnet and Trishna Singh [4] through their work devised the framework for the unification of different fonts (true type fonts and Unicode) used to represent Devanagari scripts. The framework was used to convert Nepali texts written in TTF to Unicode. For that, 8 sets of TTF were considered and the corresponding Unicode encodings were generated, which gave the average accuracy of 99.90 %. The test was performed for 3000 sampled unique words extracted from a news corpus having 700000 documents. The report highlights the encoding representation of Unicode(multi-byte font) and TTF(single-byte font). Moreover, different TTF font families also exhibit different encodings. So, in order to standardize and convert from one font to the other the framework was required as the direct conversion is impossible due to the presence of different glyphs. In order to cope up with these difficulties, the framework first tokenized the characters, converted the fonts, re-arranged the glyphs according to rules based on the glyph category types and then combined to form the whole word.

The method for rendering text on screen was proposed in [5] . The Unicode Standard does not define character glyph. It defines how characters are interpreted, not how glyphs are rendered on the screen. Context analysis is necessary for many scripts to present the correctly shaped and combined glyphs. A character or a composite character can have different visual representations(called shapes or glyphs) depending on context. Here, a general approach to encoding script-specific rendering rules was presented that is based on the Unicode character set and using finite state transducers. The associated program was written in JAVA, which makes it portable to many environments. This approach was demonstrated with writing rules for some languages that use the Arabic script and a short example that renders certain Hindi words.

The computational architecture for representing Indic language was proposed by S. P. Mudur et. al. in 1997 [6] . In the research, they signified the different nature of Indic language due to which the shaping process of characters in the kernel level becomes difficult. This nature in the Indic language arises from differences in glyphs of any characters depending upon the use cases. They proposed 9 different processes to shape Indic scripts in computers. In order to complete all of the 9 processes to represent the Indic script, they gave 12 different rules that had to be considered. In the paper, how the characters (more often syllables) are represented in unicode fonts and TrueType Open(TTO) fonts. Nepali language, which is written in Devanagari script, being Indic language could follow the similar rules and processes to render the scripts in JVM.

Unlike the case with open source projects, Microsoft has developed its own shaping models for Devanagari. The <deva2> [7] shaping model developed by Microsoft includes detailed rules and guidelines for substitution , reordering and positioning of glyphs for correct rendering of Devanagari text . It also specifies local features for different rendering of the same character in different languages and includes Nepali in its list of supported languages. One of its significant drawbacks is that it is proprietary and not open source. Thus, it is difficult to integrate it to softwares that are not included in the Microsoft ecosystem. Nevertheless, the detailed guidelines and documentation of Microsoft’s Devanagari shaping model is useful for the present project as we aim to achieve similar results in JVM what Microsoft has achieved in its own ecosystem.

J Shimada [8] provides a narrative of the rivalry between the two rival font standards of TrueType and PostScript technologies throughout the late 1980s and 1990s to capture the market for digita typography. Due to the monopoly of Adobe (which had developed the PostScript technology) on Type 1 format, Apple and Microsoft set aside their rivalry to develop a alternative to PostScript.Thus, they developed TrueType.The rivalry, often termed ’font wars’ , between the two standards spread not only within the respective companies but also throughout the font design community and their loyalty. While the intense narrative provided by Shimada is certinly interesting on its own, the more significant thing for the present project is that the end of the rivalry in the late 1990s lead to the devcelopment of the Open Type initiative , which allowed designers and users to approach both TrueType and Type 1 formats in the same ways. The OpenType acted as an abstraction on top of both the previous formats and thus provide a more standard interface for everyone to use.

Ishida (2014) offers an in-depth guide on rendering Indic scripts, with a focus on text layout and typographic challenges.[10] The study highlights critical aspects such as vowel reordering, glyph shaping, and the interaction of diacritical marks in complex scripts like Devanagari. It also explores the role of Unicode in achieving consistency across platforms. The documentation also underscores the significance of dynamic glyph substitution and accurate positioning to ensure readability in digital formats. Although the documentation is geared towards the rendering of Indic scripts for web interfaces and not for other file formats, it stands as a key reference for understanding the technical demands of Devanagari rendering.

Chandra (2020) discusses the challenges and obstacles in digital typeface of Indic scripts.[11] The paper clearly points out various issues in typeface development for Indic scripts, including the lack of letterform grammar and visual language of typeforms. As most Indic scripts share similar features, the problems identified by the paper are certainly crucial to the understanding of and the finding solutions to the lack of unification in Devanagari script for Nepali language.

# 3. REQUIREMENT ANALYSIS

## 3.1 Project Requirement

* Integrated Development Environment (IDE) for Java development (e.g. IntelliJ IDEA, Eclipse, NetBeans, VSCode).
* Java Development Kit(JDK)
* Apache PDFBox library for PDF document manipulation and rendering.
* Version Control System (e.g., Git) for collaborative development and code management.
* Issue Tracking and Project Management: Github Issues
* Diagramming Tool: draw.io
* Font Modification : TTX, FontForge. FontDrop
* LaTeX for documentation
* Additional Resources: Official documentation for Java, Apache PDFBox, The Unicode Standards, Nepali Brihat Shabdakosh

## 3.2 Feasibility Analysis

### 3.2.1 Technical Feasibility

PDFBox is a robust Java library that supports creating and manipulating PDF documents, which is critical for developing a rendering engine for the Devanagari script. The development will use the Java Development Kit (JDK), IDEs like IntelliJ IDEA, and version control with Git. Essential technologies include PDFBox for PDF handling, Java for application logic, and Devanagari fonts for accurate text rendering. Standard student workstations with adequate RAM, storage, and modern processors will suffice for the project’s development needs.

### 3.2.2 Operational Feasibility

The project team will consist of four undergraduate students. The team will need to be proficient in Java and familiarize themselves with PDFBox and the Devanagari script through workshops and training sessions. The timeline will account for initial learning and development, regular coding sessions, and continuous testing.Potential risks such as the learning curve for new technologies and integration challenges with PDFBox will be mitigated through thorough research, training, and regular testing.

### 3.2.3 Economic Feasibility

The primary costs for the project will be minimal, consisting mainly of basic development tools and resources available to students. PDFBox is open-source, eliminating licensing costs. The budget will include expenses for any additional software tools and minimal hardware upgrades if required. The project provides significant educational value and addresses a high demand for Devanagari tools in Nepali language applications, potentially offering revenue through licensing and support services, and benefits from community contributions as an open-source project.

# 4. THEORETICAL BACKGROUND

## 4.1 TrueType Fonts (TTF)

TTF stands for True Type Fonts. TTF was first developed in the 1980s by the combined effort of Apple and Microsoft to solve the then raging problem that there were no fonts that could be used seamlessly on both set of software galaxy. Thus, it replaced the earlier PostScript font format. TrueType fonts use a scalable outline font technology, where the font shapes are defined by mathematical curves rather than a fixed pixel grid. Due to the use of mathematical curves rather than fixed pixel grids, TTF fonts are easily scalable. As stated earlier, TTFs are great for cross-platform compatibility. There are, however, some serious defects in TTFs including the limited presence (or rather a great lack of) ligatures and conjunct characters, and contextual alternates.

## 4.2 OpenType Font(OTF)

OpenType Font is a more recently developed by the combined effect of Microsoft and Adobe in the late 1990s. The major development in the OTF is that it has Unicode support, and has support for a vastly larger number of typographical features, such as ligatures, contextual alternates, etc. Unlike the TTFs, OTF introduces the concept of GSUB and GPOS tables where complex conjunct characters and ligatures can be formed from component glyphs rather than every character being stored separately. The downside of OTF compared to TTF is that it has larger file size. Therefore, for slow devices and for those devices that work on a very large amount of data, OTF’s performance is a little slower. Figure 4.1 shows some common conjunct forms in Lohit Devanagari which is an OTF font.

A screenshot of a computer

Description automatically generated

Figure 4. : Some Conjunct forms in Lohit Devanagari font

### 4.2.1 Differences from TTF

Both TTF and OTF encode Devanagari (and other scripts) using a set of tables which map and define relationship between glyphs and their representation. Some of these tables are shared between the two whereas others are found only in one of them. As stated earlier, OTF introduces the concept of GSub and GPos which are absent in TTF. For Devanagari, the TTF based fonts (eg: Preeti, Kantipur, etc) are based on ASCII are not mapped directly whereas OTF based fonts are generally Unicode compliant and the glyphs correspond directly to unicode.

While both TrueType and OpenType are scalable, vector-based font formats, the key difference is the level of support for advanced typographic features. OpenType fonts offer a much broader range of typographic capabilities, making them more suitable for professional-grade typography and multilingual publishing along with the use of lookups(GSUB, GPOS tables).

The common tables present in both TTF and OTF include:

**1. glyf**

The glyf table contains the actual shape of the glyph to be rendered. In TTF, the glyphs are defined by quadratic Bezier curves and in OTF by cubic curves. Cubic curves allow for more complex glyph shapes. Figure 4.2 illustrates the glyph shaping for Devanagari character ढ for Lohit Devanagari font.

A screen shot of a drawing

Description automatically generated

Figure 4. : Glyf example

A screenshot of a computer

Description automatically generated**2. cmap**

Figure 4. : cmap example from Noto Sans Devanagari

cmap table maps the glpyhs to their corresponding encoding. In OTF, the mapping is typically Unicode compliant but in some TTF fonts the encoding may be ASCII based as well (eg: Preeti).Figure 4.3 shows a portion of the cmap table for Noto Sans Devanagari. It shows both the glyph and the corresponding Unicode numbering that the glyph is encoded in.

**3. loca**

The ’loca’ table stands for index to location . It is a critical structure that maps each glyph ID (GID) to its corresponding position in the glyf table. The entries in the loca table are offsets relative to the start of the glyf table, and these offsets must be in ascending order.

The length of a glyph’s data is calculated as the difference between two consecutive offsets: loca[n+1] - loca[n].

To determine the length of the final glyph, the ’loca’ table includes an additional entry for the end of the last glyph.

For example in devanagarai fonts, each character or conjunct ligatures (e.g., ”क्” + ”ष” → ”क्ष”) is represented by a glyph. The loca table helps locate these glyphs’ data efficiently in the glyf table, ensuring accurate rendering. In this case, the ”क्ष” conjunct’s offset would be found between entries for the constituent glyphs, which are: ”क” and ”ष”.

So, loca table complements the GSUB table and allows the rendering enginers to handle glyphs correctly and display smooth text, especially for scripts like Devanagari where dynamic combination of glyphs into ligatures mean that extensive substitutions are required and different glyphs require different combinations.

**4. Hmtx**

Hmtx table for for TTF and OTF contains contains the horizontal metrics (width and left side bearing) for each glyph. It can also contain verticel metrics for some scripts where they are required. Vertical metrics are not generally included in htmx for Devanagari OTF.

**5. head**

head table includes the header information and includes metadata for the script.

**6. maxp**

maxp table includes information about the maximum profile information about the font. The maximum profile information includes max glyphs , max points etc.

**7. kern**

kern table includes information about kerning distance between specific glyph pairs. The table is similar for both TTF and OTF, but OTF includes information about contextual kerning and TTF usually doesn’t. Figure 4.4 shows the kern table for Noto Sans Devanagari font.

A table with numbers and numbers

Description automatically generated

Figure 4. : Kerning table for Noto Sans Devanagari

**8. POST**

POST table includes information for PostScript output and includes things like font metrics. PostScript is a page description language and is most commonly used in electronic and desktop publishing contexts.

The advanced tables present in OTF but absent in TTF include :

### 4.2.2 GSUB table

GSUB stands for Glyph Substitution table. A basic assumption in OTFs is that , although a single character is often rendered as a single glyph, the two are not necessarily the same. Any given text in a script is encoded as a sequence of characters which are mapped into glyphs using the cmap table. When multiple characters are required to form a conjunct ligature, this is not done simply as with single character to glyph mapping but through the rules of substitution.

OTF fonts have defined set of rules for substituion of glyph before rendering to achieve certain forms like ligatures. A list of the substituion rules are listed in the GSUB table based on the cmap and glyf tables as well as anchors. In Devanagari , GSUB is mainly used to to rearrange diacritics that are typed later but are shown earlier (eg: ि) as well as for conjunct ligatures.

GSUB tables usually define specific substitution rules under certain common classes and so on in a hierarchy as can be seen in Figure 4.5. At the highest level, seven types of glyph substitutions are suppported in GSUB table in fonts used for international typography:

1. Single substitution
2. Multiple substitution
3. Alternate substitution
4. Ligature substitution
5. Contextual substitution
6. Chained contexts substitution
7. Reverse Chaining contextual single substitution

A diagram of a system

Description automatically generated

Figure 4. : Organization of GSUB table

Various rules for substitutions in GSUB table for Devanagri scripts are discussed in subsection 4.3(Rendering Rules).

Image 4.5 shows the organization of a GSUB table. A Lookup List or table defines the rules of substitution for different types. The rules of substitution may differ according to the information contained in the Script List and Language System Info which often act as sub-tables and are pointed by offset from the GSUB header.

A simplified procedure for accessing GSUB information can be illustrated by the following :

1. Read the script in the script table ( Script List tag).
2. If script tag is valid (i.e. the tag refers to a supported language), search for the language in the Language System Info table (langsys tag).
3. If the language is presemt in the Language System Info table, use the language. Otherwise use the default language pointed by langsys.
4. Read the input glyph string and the featureTag for all the features. Select all the feature tables which apply for the read input string.
5. The feature table point to the Lookup List table for data needed to implement all possible glyph substitutions. Apply the glyph substitutions from the Lookup table.

### 4.2.3 GPOS table

GPOS stands for Glyph positioning table. It defines certain features so as to fine tune the position of glyph in a syllable. For example, the position of certain diacritics like ◌ु depends upon the syllable. Thus, the GPOS features are used to place the glyph at the correct position after all the substitution has been made according to the GSUB table.

For simple glyphs that are not combined with above or below glyphs may be positioned simply with reference to placement and advance values depending upon the virtual pen point. For more complex situations, more flexibility is required for the precise control of how the positioning of individual glyphs is done.

A diagram of a system

Description automatically generated

Figure 4. : GPOS table format

Like GSUB, GPOS defines its rules in hierarchies as depicted in figure 4.6. At the top, eight types of actions for positioning and attaching glyphs are recognized by the GPOS table:

1. single adjustment
2. pair adjustment
3. cursive attachment
4. mark-to-base attachment
5. mark-to-ligature attachment
6. mark-to-mark attachment
7. Contextual positioning
8. Chained contexts positioning

GPOS table is organized in a similar way to the GSUB table.Image 4.6 shows the organization of a GPOS table.

Script List and Language System Info are first checked to get info about the language and script of the font in the exact same way as in the GSUB implementation. The only difference between the two is that in GPOS, the features implement work in changing the position of the glyphs and not in substituting one glyph for another as in GSUB table.

As the Script and Language System Info are used by both GPOS and GSUB, they are briefly explained as follows:

**Script Tag**

Script tags are present in OTFs and generally, but not always, corresponds to a Unicode script. For example, Greek script is represented by the script tag ’grk’ and its features correspond to the features of Greek script as defined in Unicode. Although this correspondence between script tags and the respective Unicode script generally holds true, script tags are sometimes correlated to a particular OpenType Layout Implementation rather than the Unicode encoding. So, a single Unicode script may have two or more script tag. In the case of Devanagari script, there are two script tags : ’deva’ and ’deva2’ which correspond to two OpenType Layout implementations. ’deva2’ is the more modern of the two and generally is more advanced and provides more features compared to ’deva’. The hierarchy of different tags in an OTF font can be visualized from figure 4.7.

A diagram of a language system

Description automatically generated

Figure 4. : OTF tables hierarchy

The list of script tags is called a ScriptList table. The ScriptList table is used both by the GSUB and the GPOS tables to access what specific substitution or positioning apply to that specific script.

Multiple langsys tags may be related to a single script tag.So, in case of a modern Devanagari OTF, there are two script tags ’deva’ and ’deva2’ which respectively may have multiple languages correspond to them through langsys tags like ’NEP’, ’HIN’, ’MAR’, etc.

A simple illustration of different tags and tables in OTF is shown in figure 4.7.

**Language System (langsys)**

The language system supports language-specific rules. This is specially useful in scripts like Devanagari which are used by a large number of languages between whom typographic standards may differe quite a lot. Thus, many typographic features in common use in Hindi (nukta) or Sanskrit (avagraha) are not in common use in Nepali although all use the same devanagari script.

Although included under the heading of tables, langsys can more accurately be described as tag present in the tables present in OTFs. langsys tags in OTF tables are identifiers that represent specific typographic conventions associated with how text in a given script is displayed. These conventions may vary based on language, usage context (e.g., academic, phonetic, mathematical), or publication style. Each tag aims to capture these conventions for rendering text accurately in contexts like glyph substitutions, positioning, or stylistic alternates.

Thus, The language system supports language-specific rules. This is specially useful in scripts like Devanagari which are used by a large number of languages between whom typographic standards may differe quite a lot. For example, many typographic features in common use in Hindi (nukta) or Sanskrit (avagraha) are not in common use in Nepali although all use the same devanagari script.

Although the system tag is asscociated with language, it may not always correspond one to one with language. The tables present with the language system tag mainly describes a set of conventions for writing a certain language in a certain scripts. So, it is not guarenteed that the conventions that differ between languages A and B should also differ for languages B and C.

For Devanagari script, multiple languages are present in modern fonts that contain system tag. For example, Nepali, Hindi and Marathi are represented in language system tag as “NEP”, “HIN” and “MAR” respectively.

## 4.3 Rendering Rules

The rendering rules listed in Appendix C are the rules defined by Unicode Standard Core Specification for the latest Unicode version. In the following section, these rendering rules are described with corresponding examples.[9]

As described in the rule **R1** in the appendix, if any whole consonant or a ligature is followed by VIRAMA sign as the last character in a word or if VIRAMA sign is followed by JNWJ, then the sign is shown explicitly.

Example:

क + ◌् → क्

ज्ञ + ◌् → ज्ञ्

Here, क् and ज्ञ् are considered to be dead consonants.

According to the rule described in the rule **R2**, the ’Reph’ form of RA is represented by RAsup. It occurs when the dead consonant of RA is followed by a consonant.

Example:

RAd + TAl → KAl + RAsup → Desired output

र् + क → क + ◌र् → र्क as in नर्क

The rule **R3** gives the rule for Reph in the case of conjunct ligature. According to the rule, if half form of RA is followed by, half form of consonant and a whole consonant such that the half form of consonant and a whole consonant form a conjunct ligature, then RAsup gets applied to the whole conjunct ligature.

Example:

RAd + Cd + C → conjunct\_ligature +RAsup → Desired output

RAd + KAd + SAl → K.SSA +RAsup → Desired output

र् + क् + ष → क्ष + ◌र् → र्क्ष

The rule **R4** can be taken as the extension of the rule R3. It says if the same text sequence is encountered, but the half form of consonant and a whole consonant do not form a conjunct ligature, then the REPH sign is positioned at the base glyph of the consonant cluster.

Example:

RAd + Cd1 + C12 → Ch1 + C12 + RAsup → Desired output

RAd + TAd + YAl → TAh + YAl + RAsup → Desired output

र् + त् + य → त् + य + ◌र् → र्त्य

In some cases of Nepali language, half form of RA when precedes some consonants (like य, ह) should be represented ass eyelash-RA instead of RAsup(REPH). So rule 5a) gives the rule for the formation of eyelash-RA. It says if half-form of RA is followed by ZWJ, the eyelash-RA gets formed. Example

RAd + ZWJ → RAh

र् + ZWJ → र्

### 4.3.1 Rules for Rakaar form

The rules **R6** and **R7** deal with the computational forms for the Rakaar formation. Both of the rules follows the following forms:

Cd + RAl → Cn + RAsub → Desired output

However, rule **R6** shows the Rakaar formation for the characters where there are absence of sticks. In such cases, subscript nonspacing mark RAsub gets placed below the consonant,

Example:

TTAd + RAl → TTAn + RAsub → Desired output

ट् + र → ट + ◌र् → ट्र

Similarly, the rule **R7** shows the Rakaar formation in consonants with having sticks. In this case, RAsub when replaced to the consonants, gets attached to the stick to form a new ligature.

Example:

MAd + RAl → MAn + RAsub → Desired output

म् + र → म + ◌र् → म्र

The rule R8 describes what happens when a RAd is preceded by another dead consonant.

As RAd is also a dead consonant, the virama remains. Example:

TAd + RAdTAn + RAsub + VIRAMAnT.RAd

त्+ र् → त + ◌र् + ◌् →

Thus, this rule prefigures the rule **R12** which deals with how conjunct ligature can act as a dead consonant.

The rule **R10** describes the placement of modification marks especially bindu. The bindu is applied to the syllable as a whole and thus should be placed at the last. Any possible vowels present in the syllable should come before the vowel.

Example:

KAn + AAvs + CANDRABINDUn

क + ◌ा + ◌ँ → काँ

In the example, the bindu is placed above the matra and not above the base glyph. In the representation, it follows after the dependent vowel.

### 4.3.2 Rules for Ligatures

The following rules up-to rule **R14** are mainly concerned with ligature formation. Although they define general rules for ligature formation, the actual representation actually depends upon the presence and availability of glyphs in the individual font.

The rule **R11** describes the general rule of a how a ligature forms, i.e a ligature forms if a dead or non-dead consonant is preceded by a dead consonant and the ligature is available.

Example:

JAd + NYAlJ.NYAn

ज् + ञ → ज्ञ

And,

ट् +ठ → ट्ठ

The rule **R12** describes that even a conjunct ligature can act as a whole consonant as well as a dead consonant to form the more complex ligatures. It also indicates that even some of the conjunct ligatures can have half-forms. Example

SAd + TAd + RAn → SAd + T.RAn → S.T.RAn

स् + त् + र → स् + त्र → स्त्र

And,

K.SSAd + YAl → K.SSh + YAn

क्ष् + य → क्ष्य

The rule **R13** describes the application of RAsub . When a nominal consonant or a ligature is followed by a RAsub , they can join to form multipart conjunct character. In overall effect, it is similar to rule **R6** but whereas rule **R6** dependent consonant is followed by RAl , here the nominal consonant is followed by a RAsub .

Example:

KAn + RAsubK.RAn

क + ◌र् → क्र

The rule R14 illustrates the unusual or nonstandard location of matras to the based consonant. It occurs only in two cases - RAl with UVS or UUVS.

Example:

RAl + UVS → RUn

र + ◌ु → रु

And,

RAl + UUVS → RUUn

र + ◌ू → रू

The rule 15 deals with the rearrangement of dependent vowel IVS. It states that the dependent vowel IVS is always placed at the leftmost part of the cluster. If constant cluster forms ligature, the matra is shown left to the ligature. This reorder mechanism is rquired for only short Ikar vowel(ह्रस्व इकार).

Example:

TAd + RAl + IVS → T.RAn + IVS → IVS + T.RAd

त् + र + ि → त्र + ि → त्रि

The rule 16 deals with the reordering of dependent vowel IVS in the case of explicit virama either due to the use of ZWNJ or due to the absence of half-form of any consonant or due to the absence of glyph in the font.

Example

TAd + ZWNJ + RAl + IVS → TAd + IVS + RAl

त् + ZWNJ + र + ि → त् रि

## 4.4 Unicode

Unicode is text-encoding standard that unifies the encoding of most of worlds scripts and more. It provides a unified standard that enables programmers and others to encode text in almost all major languages in the world using almost all the major scripts used in the world. As of 2024, the latest version of Unicode is 15.1 which defines 149813 characters and 161 scripts in addition to emojis, and so on.

Unicode defines three standard encoding UTF-8, UTF-16 and UTF-32 according to the number of bits used for the encoding. Due to the ease of use as well as backward compatibility with other encodings like ASCII standard, UTF-8 is the most widely used standard. Unicode assigns a unique number for every character and enables encoding and writing systems for all languages around the world. The development of a multi-byte Unicode system for Devanagari script introduces a different style of glyph ordering system than the traditional single byte encoding True Type Fonts (TTF). Different interest groups and lack of appropriate technology in the Nepali writing system introduces variations on conjunct formation and character rendering in different applications.

### 4.4.1 Unicode Support for Devanagari Script

In early stages of development of computers, the system was geared towards encoding mostly Latin alphabet as the development in the field was led by American and European scholars whose native languages were written in Latin alphabet. Thus, the original ASCII encoding scheme, developed in the 1960s, encoded 128 characters of Latin script, mostly for English language. Other scripts, including Devanagari, were not supported. Due to this, it was well nigh impossible to work with Devanagari script in digital computers. The time and resources required to work with Devanagari texts were consequently much higher than for corresponing volume of text written in Latin alphabet.

In the 1980s, the Government of India developed the ISCII scheme. ISCII stood for Indian Script Code for Information Interchange and was modelled after ASCII but geared towards the digital encoding of various Indic scripts, including Devanagari. It defined a standard for encoding characters and provided a way to represent the script in digital form. While it was a step forward in the right direction, the lack of standardization and support in limited set of platforms meant that ISCII could not be adopted widely beyond India. Even in India, the scheme did not lead to an expected explosion in the adoption of use of Devanagari and other scripts.

A host of lesser known encoding schemes existed for Devanagari in the 1980’s and 90’s that were developed by individual companies for the propietary use. Due to this diversity and lack of standardization, these developments did not lead to widespread adoption of these encoding systems by the general public.

Finally, Unicode was released in 1991. Along with the Latin alphabet, it included a large number of other scripts used by languages all over the world including Devanagari. Unlike the previous encodings, Unicode defines the glyphs and the rules for combining glyphs to form different ligatures but the actual implementation of the rendering rules is actually performed by the rendering engine. Some common rendering engines that support Unicode Devanagari include: HarfBuzz, Uniscribe, Pango, Core Text, Skia ,etc. Even in its early days, Unicode proved to be an improvement over the existing ISCII and other proprietary encoding schemes. Due to these reasons, we have chosen Unicode for our project:

**1. Ligatures**

ISCII had a limited set of ligatures that were hard coded within specific implementations. So, the resulting text would not render as needed due to lack of certain ligatures. Due to the hard coding of ligatures, dynamic formation of ligatures while actually writing the text was not possible. Unicode overcomes all these problems. It allows for font developers to implement a large number of ligatures by implementing rules for combining glyphs. The ligatures are then formed in dynamic time according to the rules.

**2. Character Representation and Rendering**

In ISCII , ligatures and compound characters are generally represented as individual characters.This leads to inconsistencies and other problems. In Unicode, each glpyh is represented separately and are combined whenever necessary to form ligatures. For example, ”क” (U+0915) + ”◌ा” (U+093E) = ”का”

**3. Flexibility**

While Unicode supports combining of glyphs to form ligatures whenever necessary, it also supports precomposed ligatures formed from glyphs.

So, Combining: ”क” (U+0915) + ”◌ा” (U+093E) = ”का” Precomposed: ”का” can also exist as a single character.

**4. Normalization**

Unicode supports normalization so that even when same ligatures are formed from combining different glyphs, they can be treated equivalently.

For example, क + ◌ा + ◌े = को and क + ◌ो = को are equivalent, even though they are formed differently. This, is specially important for searching and indexing the text, especially in large amounts.

**5. Standardization**

Unicode provided a consistent encoding scheme, provided rules and guidelines for rendering engines so that there is standardization. Thus, developers of rendering engines as well as font developers can work using a standard scheme.The users also can use different fonts without worrying about the changes that are affected.

**6. Cross-Platform Support**

Unlike the encoding developed by Microsoft and other proprietary encodings developed by individual companies, Unicode does not depend on the device. Provided that a compatible rendering engine is used, the same rendering is displayed on different devices and operating systems.

### 4.4.2 Font Use in the Case of Nepal

In the specific case of Nepal, both TTF and OTF fonts are in common use. “Preeti”, “Kantipur”, etc are some of the more prominent TTF fonts, whereas “Kalimati”, “Mangal” etc are some popular OTF based fonts. During our own interviews in the publishing field, “Preeti” was found to be more in use as compared to other fonts.

### 4.4.3 TTX

TTX stands for TrueType XML or TrueType Extensible Markup Language. But rather being a markup language, it is a versatile file format designed to store data in a human-readable, XML-based structure. It is used to represent complex information in easily inspectable, editable and manipulatable text format. This format is commonly used in various domains, enabling a structured approach to handling data that can be processed by both humans and machines. This file format facilitates tasks such as data inspection, modification, and transfer across different systems and tools.

TTX is a Python based tool that is based on XML and primarily used for analyzing, modifying and debugging font files. It is a part of python based FontTool library. It allows dumping a font file (.ttf or .otf files) to .ttx file, where the properties and the hierarchy of the various tables and tags used in the font could easily be observed.

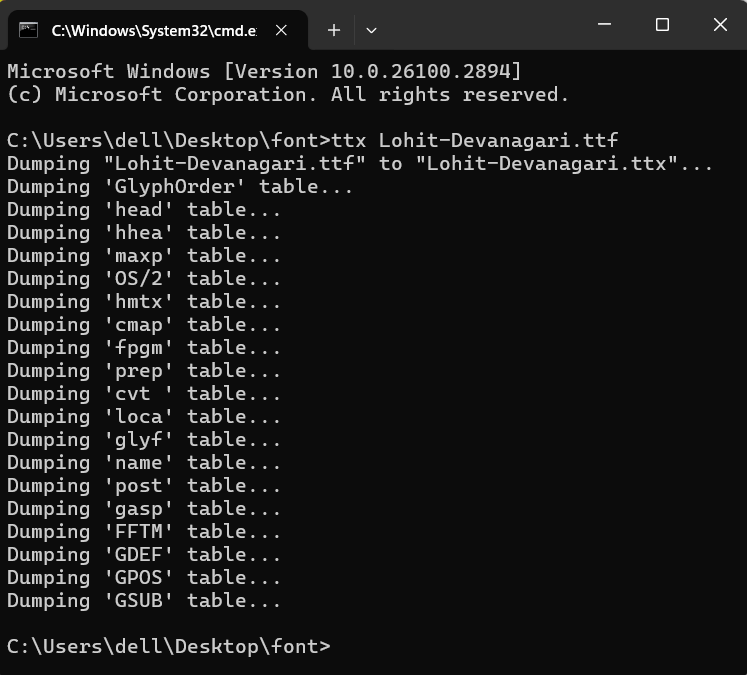


Figure 4. : Conversion of fontfile into ttx format

The process of dumping ttf file into ttf file could be observed as shown in the figure above. While dumping, each table gets extracted one by one and a new ttx file gets created. The ttx file thus created could be used using notepad++ and the following structure may be seen.

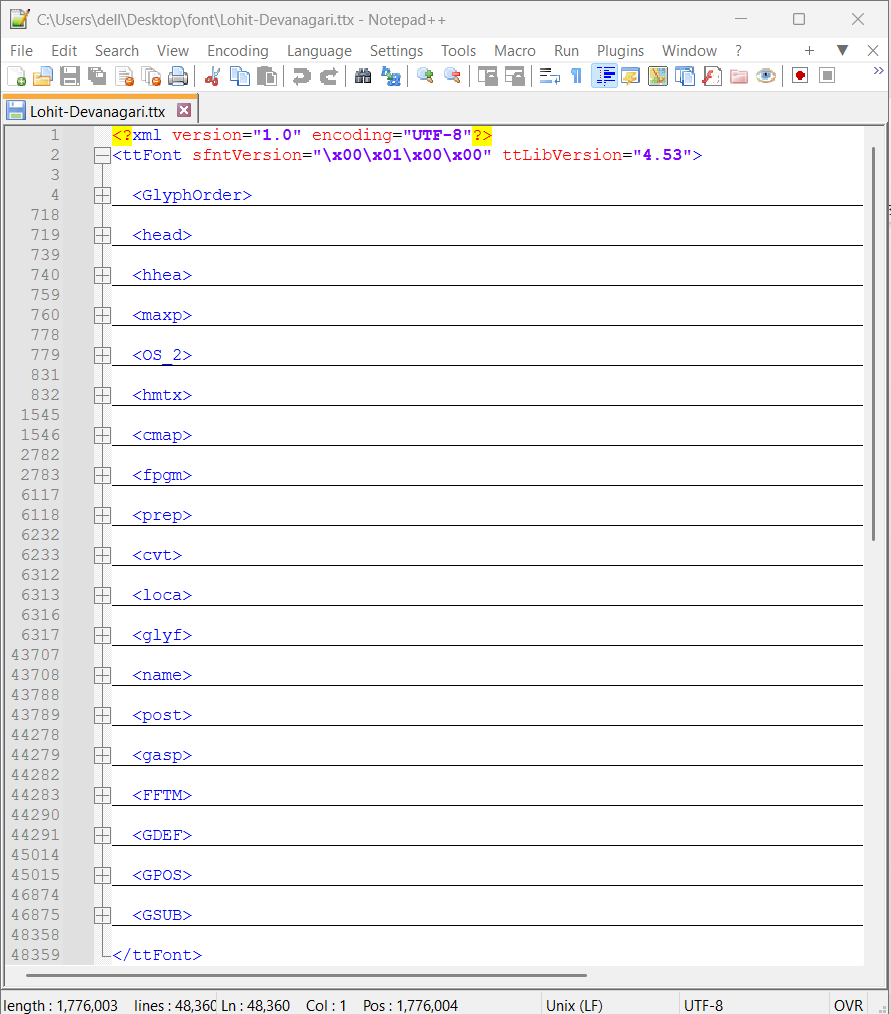


Figure 4. : xml view of fontfile

The file contains the details of various tables available in TTF font file.

For example:

<GSUB>

<Version value="0x00010000"/>

<ScriptList>

<!-- ScriptCount=3 -->

<ScriptRecord index="0">

<ScriptTag value="DFLT"/>

<Script>

<DefaultLangSys>

<ReqFeatureIndex value="65535"/>

<!-- FeatureCount=1 -->

<FeatureIndex index="0" value="6"/>

</DefaultLangSys>

<!-- LangSysCount=0 -->

</Script>

</ScriptRecord>

This helps to visualize the properties of each table and its structure. Also, we could modify some contents of the table (XML file). Then, using this tool, the modified TTX files could be compiled back into a new font file.

### 4.4.4 FontForge

FontForge is a tool used to view the font information and edit it. FontForge is free and open-source software. In the project, it is utilized to view the glyphs and the GSUB and the GPOS rules. Using the software, one can view subtables of GSUB and SPOS table, and the rules implementation. Also, one can view information such as which lookup table is utilized by which scripting language.

fontForge can also be used to view the general glyphs. Each glyphs when hovered displays its name. The glyphs viewed in the below figures are only the glyph available in the file.

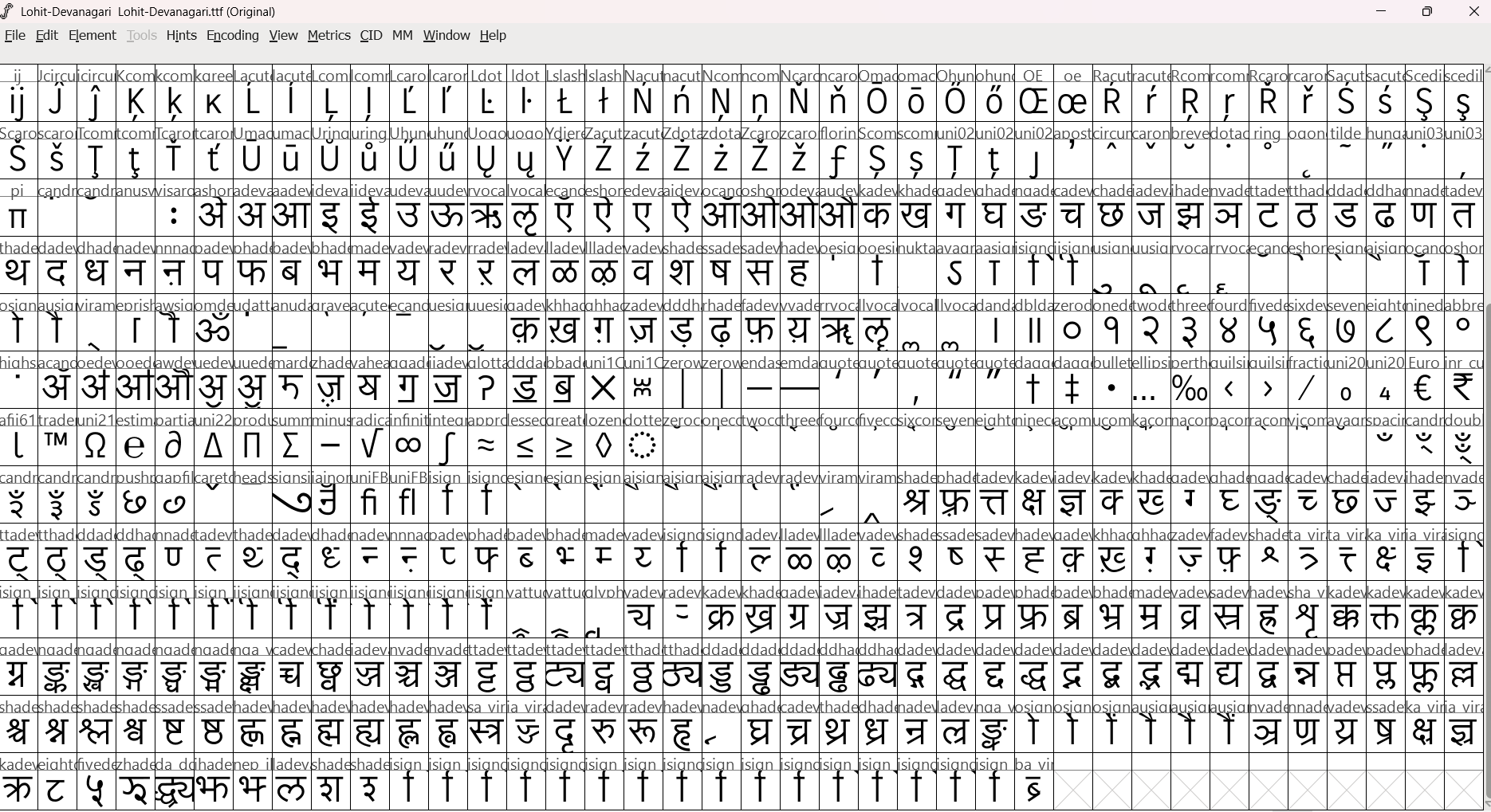


Figure 4. : FontForge cmap table

Also, the GSUB table could be viewed. GSUB tables are divided into lookup tables and rules are defined within lookup tables. Different lookup tables are present for incorporating different types of features.

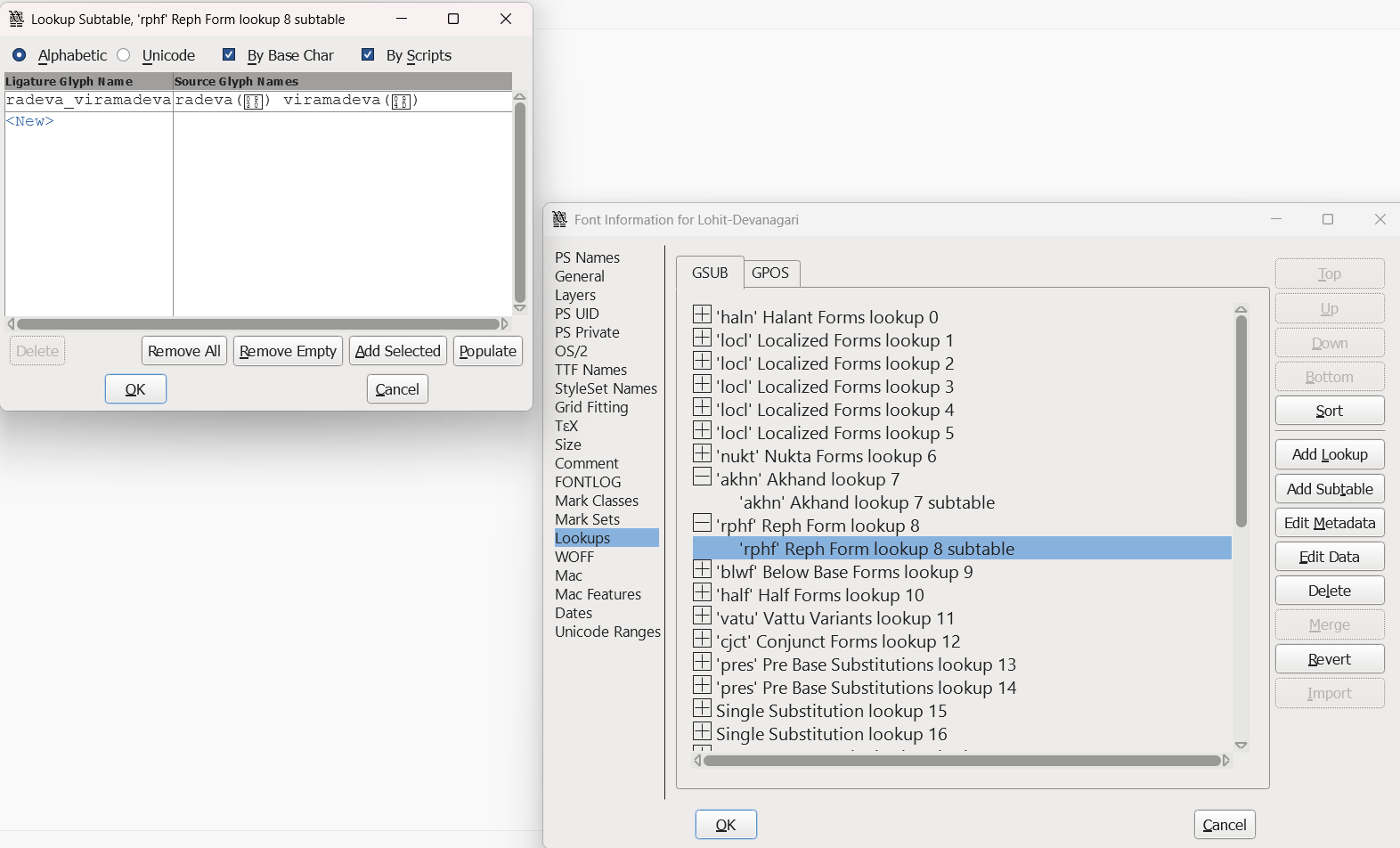


Figure 4. : FontForge GPOS Table

### 4.4.5 FontDrop

FontDrop is an online tool for viewing font information. It allows to view the script, langsys information, glyph IDs and glyph names. It is utilized for testing the algorithm as its UI is better for searching glyph that should be formed and its correspondig IDs. As the algorithm implemented results in reordered and substituted glyph IDs, it is impportant to find the glyph IDs of the glyphs that should have been formed.

## 4.5 Portable Document Format (PDF)

PDF stands for Portable Document Format is a standard file format used for displaying texts as well as graphics. It is an Open Standard (ISO 32000-1:2008) that is maintained by Adobe Inc. who introduced the format in 1992 to address the lack of a universal file format whose document layout and appearance is preserved across various devices. It was first defined in the famous Camelot whitepaper in 1990 by a team of developers in Adobe headed by John Warnock .PDFs are device and platform independent and thus can be viewed identically over diverse set of devices and applications. This attribute of the PDF is achieved through embedded fonts, graphics, and metadata within the file itself, creating a self-contained document structure. Thus, users need not care about whether or not some font is installed in their device to view the document.

PDFs are encoded using a binary format, starting with a header that specifies the PDF version, followed by a body containing objects (e.g., text, fonts, images) that define the document’s content. Objects are organized into streams and dictionaries, which reference a cross-reference table to enable precise data retrieval. This internal structure is further reinforced by compression and encryption options, ensuring security and efficiency in file storage.

PDF lacked native support for Devanagari ( and other non-Latin scripts). However, with PDF being released as Open Source in 2008 ( it was proprietary of Adobe Inc before), the support for non-Latin scripts have increased dramatically.

A graph of different colored bars

Description automatically generated with medium confidence

Figure 4. : Graph of usage of PDFs compared to other document formats.

In the present project, we have focused only on PDF and not other formats due to the following reasons:

**1. Popularity & Volume of Use**

According to the CommonCrawl database (as of 2021) PDF is the third most popular file-format (coming after HTML and XHTML). Compared to other comparable document formats like Docs, Epub, Djvu,etc PDF is by far the most used and consistently accounts for about 85-90% of all digital document formats on the Web.Figure 4.8 shows the use of PDF on the internet compared to other document formats. Thus, any project concerned with rendering on digital medium would have to deal with PDF first and foremost.

**2. Consistency & Portability**

PDFs guarantee visual fidelity across platforms. This is particularly vital for Devanagari, where even minor deviations can alter meaning. Due to this cross- platform consistency , one need not deal with different platforms and devices separately but can focus on the rendering issue without reference to specific devices.

**3. Opensource Standard**

PDF is an open source standard. Thus, anyone from around the world can contribute to the PDF project. The same cannot be said of the other rival formats which are mostly proprietary of certain corporations . (like Docx, pptx, etc of Microsoft).

**4. Security and Accessibility**

Password protection and tagging features enhance document security and accessibility, ensuring compliance with modern usability standards.

### 4.5.1 Font Embedding

As stated earlier, one of the most important properties of PDFs is that they are displayed the same regardless of the device or the platform that they are viewed in or the fonts installed on them. This is achieved by a process called font embedding. The PDF format supports embedding fonts to preserve document fidelity, particularly in professional and design-centric environments. The process of font embedding is generally during the creation or generation of the pdf. The font is first identified and converted to to a format compatible with PDF specification and then embedded into the encoding of the PDF itself. The embedding may be full or subset. In full embedding, the whole data provided by the font file is embedded into the pdf encoding. In subsetting, only the glyphs that are actually used in the pdf document are actually embedded into the encoding. In addition to full and subset embedding, a third method called font referencing is sometimes used. In font referencing, the PDF only references system-installed fonts instead of actually embedding them in the file structure itself. Thus, the rendering of font referenced PDF depends upon whether or not the specified font is actually present in the viewer’s system. While this helps minimize the size of the file, it may result in font substitution and layout discrepancies if the font is unavailable. Furthermore, font referencing contradicts the usual features of PDF including platform-independent consistency and portability. So, font embedding is usually encouraged.

In terms of programming, the binary data that is obtained from the font file is actually saved as a stream within a PDF Content Stream object. There are 14 fonts that are considered core to the PDF viewer and assumed that every PDF viewer would contain them. As a result, they are not embedded in the PDF file as it would be redundant and space consuming. They are called Standard fonts. Some examples include, Times- Roman, Helvetica, Courier, etc. These fonts only support Western character set over Latin script.

Apart from the standard 14 fonts, the fonts used in PDF are generally classified into two types:

**1. Type1 fonts**

Type 1 fonts are single-byte fonts and thus only support 256 glyphs. Type 1 fonts are available only for Western languages written in Latin script. As the number of characters is limited, full embedding is feasible for type 1 fonts without dramatically increasing the size of the PDF file. Some examples include, Times New Roman, Bookman, Avant Garde, etc.

**2. Type0 fonts**

Type 0 fonts are multibyte fonts and thus support a large character set of available glyphs. These are available for many non-western scripts which are generally more complex as compared to the simpler Latin alphabet. Type 0 fonts are usually Unicode compliant. As the number of characters is very large, full embedding is usually not feasible for type 0 fonts without dramatically increasing the size of the PDF file. Thus, Type 0 fonts are usually embedded using CID (Character Identifier) keyed and use the Cmap files to map character codes to glyphs. Eg, SimSun for Chinese, Meiryo for Japanese, Devanagari for Nepali etc.

## 4.6 Language & Library

### 4.6.1 Java

Java is a popular, object-oriented programming language. It was developed in the 1990s by Sun Microsystems (now Oracle Co.) with the principle of WORA(“Write Once , Run Anywhere”). The WORA principle signifies that the code can be executed over any device that has a Java Virtual Machine (JVM) installed. While this may not seem particularly noteworthy now, it was significant in the early days of High level programming and this cross-platform capability was of immense importance in the high rates of adoption of Java by developers across various programming domains. Some features include:

1. Object oriented
2. Platform Independence
3. Robust and Secure
4. Multithreading
5. Rich Standard Library
6. Strong Community Support

### 4.6.1 JVM

The Java Virtual Machine (JVM) is a cornerstone of the Java platform, providing the essential infrastructure for executing Java applications across diverse environments.It is an integral component of the Java Runtime Environment (JRE) that enables Java applications to run on any device or operating system that has the JVM installed. It acts as an intermediary between Java bytecode (the compiled form of Java programs) and the underlying hardware and operating system, allowing Java to maintain its ”Write Once, Run Anywhere” (WORA) capability.

### 4.6.2 pdfBox

pdfBox, developed by Apache, is a popular open-source library for Java language designed for working with PDF documents. It provides useful and comprehensive set of tools to perform a range of operations involving PDF files including creating, generating, manipulating, etc. PDFBox offers a user-friendly API that simplifies the process while maintaining high performance and flexibility. The features that lead to the choice of pdfBox over other similar libraries include:

1. Native Unicode Support
2. Free and Open Source
3. Rich Feature Set
4. Language Support (with JVM)
5. Proven Use Cases / Standard Library

# 5. SYSTEM ARCHITECTURE AND METHODOLOGY

## 5.1 System Architecture

**A diagram of a diagram

Description automatically generated**

Figure 5. : System Block Diagram

The major components of the Apache PDFBox pdf generation engine include: word tokenizer, reordering rules, glyph substitution (GSUB) features implementation and PDF generation. The inputs to the system are text input and font file. The output is a pdf file.

**Input Devanagari text:**

The input to the system is Devanagari text data from various sources as string of Devanagari characters. The text may be from a text file or text database. For proper processing of the text, it should contain the Devanagari characters as specified by the Unicode standard in the Devanagari block i.e. U+0900 to U+097F

**Word Tokenizer**

The input text is a string of Devanagari characters. The whole text cannot be processed at once, so it has to be broken into the smaller unit. For simplicity, the input string is broken into a list of string/word based on space or whitespace character. Then, we obtain a list of words to be processed one after another. The latter part of the system process a word at a time rather than the whole input.

**Reordering Rules**

In Devanagari script, the vowel sign (short ’i’) has a unique placement rule. When

ि◌ is written with a consonant, it appears visually before the consonant, even through it phonetically comes after the consonant. The र् reph when combined with another consonant takes special position. The reph(र्) is visually placed above the subsequent consonant or cluster, but it phonetically comes before it. For appropriate visual representation of such character/glyph in each word, reordering rules should be applied when applicable.

**Reordering Rules**

This module is responsible for reordering of the characters in a syllable. It receives the syllables from the syllable tokenizer. The characters are reordered using the rules that are taken from standard Nepali writing system and Unicode standards for Devanagari script.

**GSUB Features**

GSUB also known as Glyph Substitution is defined as the process of substituting a number of characters with a single character. The character that replaces a number of characters, should be present in the font file. These characters can be seen inside a GSUB table of font files.

**GPOS Features**

GPOS also known as Glyph Positioning includes the gap between characters and positioning of above base characters and below base characters. This is also included in the font file.

**Font File**

The Font File is a opentype font file which contains the information about the shape of glyphs. The OpenType font file provides the lookup table with language specific glyphs, GSUB table for contextual substitution and GPOS table for accurate positioning.

**PDF Generation**

After the glyph substitution and glyph positioning all the outputs are integrated and then PDF layouts are defined using pdfbox’s features to produce pdf containing the devanagari text.

**PDF File**

The final output of the system is a pdf file containing devanagari text. The output pdf file may be a bill for billing system, or news clips for media moitoring system.

## 5.2 Methodology

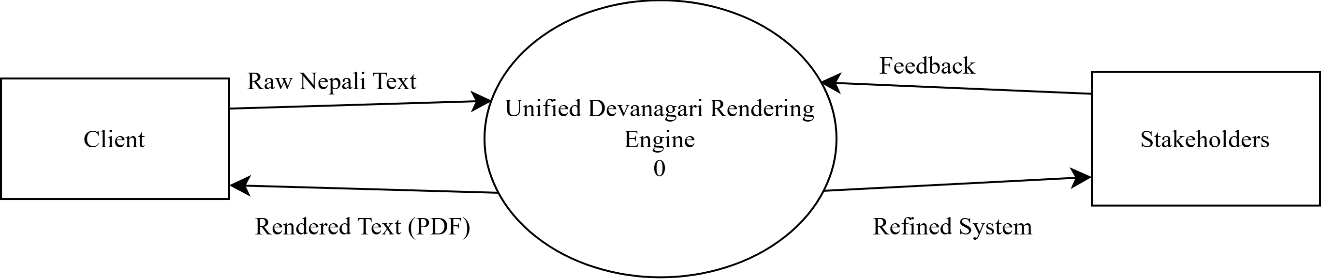


Figure 5. : DFD Level -0

A group of ovals with text

Description automatically generated

Figure 5. : DFD Level -1

A black and white sign

Description automatically generated

Figure 5. : GSUB Application Algorithm

A black and white background with white squares

Description automatically generated

Figure 5. : PDF Generation Algorithm

A black and white rectangular sign

Description automatically generated with medium confidence

Figure 5. : PDF Rendering Algorithm

A screenshot of a phone

Description automatically generated

Figure 5. : State Diagram

A screenshot of a cell phone

Description automatically generated

Figure 5. : TTF Parser

A black background with white ovals

Description automatically generated

Figure 5. : Use Case Diagram

# 6. IMPLEMENTATION DETAILS

For pdf generation using pdfbox user needs to create a ”PDPageContentStream” which contains a public method to ”showText”. This method takes string as input. Before showing a text, the content stream needs to be provide with a PDType0Font or PDType1Font font.

## 6.1 Setting font

On setting a font to the content stream, it adds the current font into the font stack of the stream. Later, the ”GsubData” is parsed from the font file. The appropriate ”GsubWorker” is selected based on the data obtained from the font file. The script tag of the font is used to decide the gsubworker to be implemented on the contentstream. The script tag for Devanagari may be ”deva” or ”dev2”. In case of no ”GsubData” found in the file, no GSUB and GPOS features will be used.

The gsubworker includes ”CmapLookup” for character-to-glyph mapping and the ”ScriptFeature” for all the available Open Type features contained in the font file. These features are used for pdf rendering.

## 6.2 Set text

Text can be shown only after setting a font to the content stream. As the font is already set, appropriate ”GsubWorker” instance is available, it will be used for the input text.

It starts with the input text for glyph substitution. At first, the entire chunk of text is broken into words by splitting it with space, from where a list of string is obtained. For each non empty words in the array, open type features are applied.

For further processing, single word is taken at a time, the glyph ids are obtained from ”CmapLookup” using the codepoints. This converts characters into glyphIds.

The glyph ids for particular word is further processed for reordering and GSUB feature implementation. The ”GsubWorker” selected for the font set for the content stream applies transformation to the original glyph ids of the word. If no ”GsubWorker” is selected then no transformations are applied and the original glyph ids are returned.

While applying the transformations by the ”GsubWorker”, the reordering of the reph(र्) is done at first followed by the reposition of the short ’i’ (ि◌). These reorderings are simple for general cases, but the special cases are also addressed based on the Nepali devanagari writing system.

Once the reordering is complete, the next step is to apply the substitution of the glyphs based on the GSUB lookups from the font file. The sequence in which the features are applied is very important. Features in order as per Open Type for Devanagari fonts: ”locl”, ”nukt”, ”akhn”, ”rphf”, ”rkrf,”blwf”, ”half”, ”vatu”, ”cjct”, ”pres”, ”abvs”, ”blws”, ”psts”, ”haln”, ”calt”. The detail description of each features is already discussed in section [from the theoretical background]. Once the substitution is applied, the processed glyph sequence is returned back to the ”PDContentStream”, where it is again encode into multi-byte encoding for use in PDF content stream.

Finally, the encoded text is written into the ”OuptutStream” with the help of ”COSWriter”. This class acts as an in-memory representation of a PDF document. It will output the given text/byte string as as a PDF object. Finally it is rendered as a PDF file document.

## 6.3 Glyph Substitution

In order to apply the gsub features for specific ”ScriptFeature” all the glyph substitutions are parsed as Set<List<Integer>> . Here, it is a set of all the unique entries for the gsub feature and a entry contains the information for a substitution of a glyph or a group of glyphs.

The current glyph replacement algorithm is based on the regex based implementation. All the available substitutions for the particular script feature forms the regular expression pattern which is used by the ”CompoundCharacterTokenizer”. Then the input glyph sequence is also tokenized with the tokenizer and split to get the tokens. The tokens is again a ”List<List<Integers>>” which is the tokenzied output of input glyphs. Then for each chunk in the tokens list. The replacement (if present) is applied from the script feature’s substitutions list. Finally the processed glyph is returned for processing for next script feature.

This process is repeated for all the script features.

## 6.4 Algorithm for implementing OpenType Shaping for Devanagari Text

Algorithm 1 OpenType Shaping for Devanagari Text

1: **Input**: Unicode encoded string representing Devanagari text.

2: **Output**: Properly shaped Devanagari text.

3: Read and tokenize the input text into individual words.

4: **for** each word in the input **do**

5: Reorder the position of र + ◌् for reph formation.

6: Reorder the position of ि◌.

7: **end for**

8: Identify all the script features for the correct font.

9: **for** each word **do**

10: **for** each script\_feature **do**

11: Create regex pattern from available substitutions for particular script feature.

12: Search for the patterns in the input word.

13: Apply substitution for identified patterns.

14**: end for**

15: **end for**

16: **for** each word **do**

17: Apply the GPOS features.

18**: end for**

19: Handle special characters like zero-width joiner (ZWJ) and zero-width non-joiner (ZWNJ):

1. ZWJ prevents the formation of conjuncts.

2. ZWNJ prevents the formation of both half-forms and conjuncts.

20: Concatenate the processed glyphs to form the final shaped text.

21: Return the final output.

This algorithm for text shaping is based on the OpenType shaping specification.

## 6.5 Regex implementation

The current tokenizer for GSUB features implementation uses regex-based approach that splits a string representation of glyph IDs into tokens based on provided substitution patterns. This mechanism is used for enabling the glyph substitution features.

**Pseudo code:**

Input:

originalGlyphs: A list of integers representing the glyph IDs. allGlyphIdsForSubstitu- tion: A set of lists, each representing a substitution pattern of glyph IDs.

Output:

A list of tokens(sublists fo integers), each representing a segment of originalGlyphs split by substitution patterns.

Steps:

1. Preprocessing the patterns

Each list of glyph IDs in allGlyphIdsForSubstitution is converted to a string delimited by a separater (”\_”). The patterns are sorted by length in descending order to prioritize longer matches.

2.Convert original glyphs to string

Similarly the originalGlyphIds is also transformed into a string representation where each glyph ID is delimited by a separator (”\_”).

3. Create regex pattern

All the substitution patterns are combined into one by joining them with a logical OR(|) to form the final regex pattern.

4. Split using regex

The regex pattern created from the list of the available substitution is used to split the glyph string from originaGlyphIds into tokens.

5. Convert tokens back to integer list

Now, each token is splitted by the separator (\_) and resulting substring is parsed into integers.

6. Substitution of glyph sequence

For each tokens in the list, the gsub feature table is scanned for replacement. If a substitution is present, then the token is substituted by the appropriate glyph sequence from the lookup table.

7. Result after GSUB

Finally the gsub feature implemented glyph sequence is obtained.

# 7. RESULTS AND ANALYSIS

## 7.1 Word List Samples

The sample of word list prepared from Nepali Brihat Shabdakosh are as follows: Table 7.1: Syllable Decomposition of Words in Devanagari

The words are categorized based on the number of the syllables present in the word. The system will be tested for words with different number of syllables. Further, the categorization may be done based on the type of the composite formation or the number of half consonants in a consonant cluster or syllable.

## 7.2 Unicode Tokenizer Outputs

Table 7.2: Decomposition of Words with Character Labeling

Word Decomposition

ʒी ʒी{WC\_HLN\_WC\_HLN\_WC\_DV}

˃चट्ठी ˃च{WC\_DV}, ट्ठी{WC\_HLN\_WC\_DV}

गतम् ग{WC}, त{WC}, म्{WC\_HLN}

कम्प्युटर क{WC}, म्प्यु{WC\_HLN\_WC\_HLN\_WC\_DV},

ट{WC}, र{WC}

जलिवद्युत् ज{WC}, ल{WC}, िव{WC\_DV},

द्यु{WC\_HLN\_WC\_DV}, त्{WC\_HLN}

स्वदेशीकरण स्व{WC\_HLN\_WC}, दे{WC\_DV}, शी{WC\_DV},

क{WC}, र{WC}, ण{WC}

पदपȼरवतर्न प{WC}, द{WC}, प{WC}, ȼर{WC\_DV},

व{WC}, तर्{WC\_HLN\_WC}, न{WC}

आवश्यकतापूfत आ{WV}, व{WC}, श्य{WC\_HLN\_WC}, क{WC},

ता{WC\_DV}, पू{WC\_DV}, fत{WC\_HLN\_WC\_DV}

अन्तररािष्टर्यकरण अ{WV}, न्त{WC\_HLN\_WC}, र{WC}, रा{WC\_DV},

िष्टर्{WC\_HLN\_WC\_HLN\_WC\_DV}, य{WC}, क{WC}„ र{WC}, ण{WC}

उपमहानगरपािलका उ{WV}, प{WC}, म{WC}, हा{WC\_DV}, न{WC},

ग{WC}, र{WC}, पा{WC\_DV}, िल{WC\_DV}, का{WC\_DV}

With the implementation of the Unicode tokenizer, each character in a word can be tokenized and labeled to appropriate group. This information will be used for further processing the the input and implementation of the OpenType features from the font file. The character output of the Unicode tokenizer needs further processing to add the positional information to each token or character.

The current version of PDFBox uses the regular expression based tokenizer. As the previous works were based on regular expression based tokenizer, the proposed syllable based tokenizer is not used in the current implementation. The glyph reordering mechanism and glyph substitution features are implemented using the existing tokenizer.

The updated version of the PDFBox is tested on 6 different unicode based devanagari fonts.(Noto Sans Devanagari, Tiro Devanagari, Lohit Devanagari, Nirmala UI, Kokila, Mangal). The input text for the test is shown in figure 7.1.

A black and white text

Description automatically generated

Figure 7.1: Test Sequence

The selection of the text sequence is based on the analysis of the output from the previous version of PDFBox. The basic reordering rules for reph sequence {र + ◌्} and short ’i’

{ि◌} were already implemented, but for the accurate rendering of the Nepali Devanagari

text it was not sufficient. Therefore, detailed implemented was needed for accurate visual representation. The result of the previous version and the current version of the PDFBox is show in table 7.3.

Table 7.3: Output of the previous and current version of Apache PDFBox

|  |  |
| --- | --- |
| Before | Now |
| A black text on a white background  Description automatically generated | A black text on a white background  Description automatically generated |

The major problems that were identified in the previous version were:

* Position of reph in the consonant cluster with other half consonants as in वर्त्स्य
* Position of short ’i’ ि◌ with reph as in टर्कि
* Half character form at the end of the word as in गर्छन्

The major cause of these problems was the lack of language/script specific knowledge. The previous version addressed the general rules for implementing the OpenType features present in any open type layout fonts. This helped to apply the GSUB features for Devanagari text but couldnot address all the cases of use.

In the case of वत्स्यर् , the rule for positioning of the reph was incomplete where the position of reph in the consonant cluster containing the multiple half consonant characters was not correct.

In the case of टfक , the position of reph and dependent vowel ि◌ is not properly set. This issue was not only in the case of ि◌ but also for all other dependent vowels. So, the problem was solved by addressing the position of reph and the dependent vowel. For correct visual representation reph is always positioned after the vowel. Later, while applying the GSUB feature the sequence may be replaced by another ligature if available in the font.

In the case of गछर् न्, the problem here was with the representation of the half character at the end of the word. The previous implementation used general substitution for consonant + halant sequence with its half form without considering its position in the character sequence. The half consonants at the end of the word is written in its halant form rather than its half character form. This problem was solved, by checking the position of half consonant before applying the half character forms.

# 8. REMAINING TASK

## 8.1 Application of GPOS table

The application of the features extracted from the GPOS table in order to correctly position above-base glyphs and below-base glyphs with respect to the base glyph is still remaining to be done. The application of features from GPOS tables will be performed in the next phases of the project.

## 8.2 Testing & validation

The testing and validation for the present project is carried out in two steps. In the first step, sequence matching is performed between the correct sequence of glyphs and sequence of glyphs obtained after substitution. In the second step, the correctness of the rendered pdf is measured visually.

While small-scale tests for correctness of substituted and reordered sequence with respect to rules for which code has been written has been performed, large-scale test for correctness and accuracy of the substituted and reordered sequence of glyphs remains to be done. Both of the testing steps will be performed in the next phases of the project.

## 8.3 Web Interface

A web interface for seeing the rendering in real time which gives generated pdf for provided Devanagari input is to be developed for the ease of presentation. The demo web interface remains to be built.

# APPENDICES

## Appendix A: Gantt Chart

A screenshot of a computer

Description automatically generated

Figure 9. : Gantt Chart

## Appendix B: Project Budget

|  |  |
| --- | --- |
| **Particulars** | **Amount** |
| Printing Expenses | 4000 |
| Book | 750 |
| **Total** | 4750 |

Table 9. : Project Budget

## Appendix C: Rules for Rendering Devanagari, The Unicode Standards

In the next set of rules, the following notation applies:

* Cn: Nominal glyph form of consonant C as it appears in the code charts.
* Cl: A live consonant, depicted identically to Cn.
* Cd: Glyph depicting the dead consonant form of consonant C.
* Ch: Glyph depicting the half-consonant form of consonant C.
* Ln: Nominal glyph form of a conjunct ligature consisting of two or more component consonants. A conjunct ligature composed of two consonants X and Y is also denoted X.Yn.
* Rasup: A nonspacing combining mark glyph form of U+0930 devanagari letter ra positioned above or attached to the upper part of a base glyph form. This form is also known as repha.
* Rasub: A nonspacing combining mark glyph form of U+0930 devanagari letter ra positioned below or attached to the lower part of a base glyph form.
* Vvs: Glyph depicting the dependent vowel sign form of a vowel V.
* VIRAMAn: The nominal glyph form of the nonspacing combining mark depicting U+094D devanagari sign virama.

The rules are :

**R1**: When a consonant Cn precedes a VIRAMAn, it is considered to be a dead consonant Cd. A consonant Cn that does not precede VIRAMAn is considered to be a live consonant Cl.

**R2**: If the dead consonant RAd precedes a consonant, then it is replaced by the superscript nonspacing mark RAsup , which is positioned so that it applies to the logically subsequent element in the memory representation.

**R3**: If the superscript mark RAsup is to be applied to a dead consonant and that dead consonant is combined with another consonant to form a conjunct ligature, then the mark is positioned so that it applies to the conjunct ligature form as a whole.

**R4**: If the superscript mark RAsup is to be applied to a dead consonant that is subsequently replaced by its half-consonant form, then the mark is positioned so that it applies to the form that serves as the base of the consonant cluster.

**R5**: In conformance with the ISCII standard, the half-consonant form RRAh is represented as eyelash-RA. This form of RA is commonly used in writing Marathi and Newari.

**R5a**: For compatibility with The Unicode Standard, Version 2.0, if the dead consonant RAd precedes zero width joiner, then the half-consonant form RAh , depicted as eyelash- RA, is used instead of RAsup .

**R6**: Except for the dead consonant RAd , when a dead consonant Cd precedes the live consonant RAl , then Cd is replaced with its nominal form Cn , and RA is replaced by the subscript nonspacing mark RAsub, which is positioned so that it applies to Cn.

**R7**: For certain consonants, the mark RAsub may graphically combine with the consonant to form a conjunct ligature form. These combinations, such as the one shown here, are further addressed by the ligature rules described shortly.

**R8**: If a dead consonant (other than RAd) precedes RAd, then the substitution of RA for RAsub is performed as described above; however, the VIRAMA that formed RAd remains so as to form a dead consonant conjunct form.

**R9**: The nukta sign, which modifies a consonant form, is placed immediately after the consonant in the memory representation and is attached to that consonant in rendering. If the consonant represents a dead consonant, then NUKTA should precede VIRAMA in the memory representation.

**R10**: Other modifying marks, in particular bindus and svaras, apply to the orthographic syllable as a whole and should follow (in the memory representa tion) all other characters that constitute the syllable. The bindus should follow any vowel signs, and the svaras should come last. The relative placement of these marks is horizontal rather than vertical; the horizontal rendering order may vary according to typographic concerns

**R11**: If a dead consonant immediately precedes another dead consonant or a live consonant, then the first dead consonant may join the subsequent element to form a two-part conjunct ligature form.

**R12**:A conjunct ligature form can itself behave as a dead consonant and enter into further, more complex ligatures.

**R13**: If a nominal consonant or conjunct ligature form precedes RAsubas a result of the application of rule R6, then the consonant or ligature form may join with RAsub to form a multipart conjunct ligature (see rule R6 for more information).

**R14**: In some cases, other combining marks will combine with a base consonant, either attaching at a nonstandard location or changing shape. In minimal rendering, there are only two cases: RAl with Uvs or UUvs.

**R15**: When the dependent vowelIvsis used to override the inherent vowel of a syllable, it is always written to the extreme left of the orthographic syllable. If the orthographic syllable contains a consonant cluster, then this vowel is always depicted to the left of that cluster.

**R16**: The presence of an explicit virama (either caused by a ZWNJ or by the absence of a conjunct in the font) blocks this reordering, and the dependent vowelIvs is rendered after the rightmost such explicit virama.

## Appendix D: Test Data Details

Table 8.3: Distribution of Entries by Number of Syllables with Sample Data

|  |  |  |
| --- | --- | --- |
| **Number of Syllables** | **Count** | **Sample Data** |
| 1  2  3  4  5  6  7  8  9  10 | 110  9981  22153  27048  11508  3682  597  119  21  2 | ʒी  ˃चट्ठी गतम् कम्प्युटर जलिवद्युत्  स्वदेशीकरण पदपȼरवतर्न आवश्यकतापूfत अन्तररािष्टर्यकरण उपमहानगरपािलका |
| **Total** | **75221** |  |

## Appendix E: State transitions of Unicode encoded text tokenizer

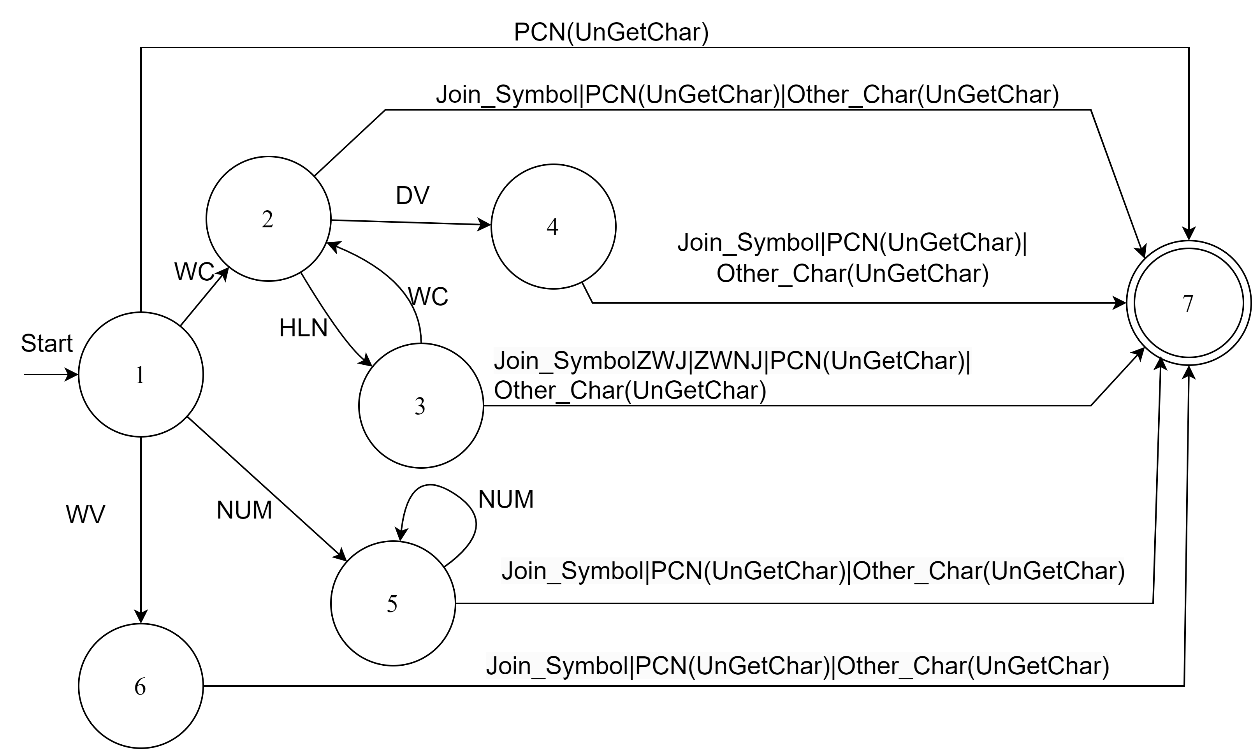
****

Figure 9. : State diagram for syllable tokenizer

## Appendix F: Character labels for Unicode encoded tokens

1. Whole Consonants (WC)
2. Whole Vowels (WV)
3. Dependent Vowel (DV)
4. Halanta (HLN)
5. Numbers (NUM)
6. Punctuation Marks (PCM)
7. Zero-Width Joiner (ZWJ)
8. Zero-Width Non-Joiner (ZWNJ)

## Appendix G: Lookup Tables in OpenType FontsA screenshot of a computer Description automatically generatedA screenshot of a computer Description automatically generated

(a) GSUB Table (b) GPOS Table

Figure 9. : Lookup Tables in OpenType Fonts

## Appendix H: Glossary

**Above-Base Glyph** : In a conjunct character, above-base glyphs are those glyphs that are physically located above the base glyph.

**Base Glyph** : In a conjunct character, base glyph is the main glyph on which the other glyphs are added as necessary.

**Below-Base Glyph**: In a conjunct character, below-base glyphs are those glyphs that are physically located below the base glyph.

**Character**: Character is the written representation of a certain sound.

**Conjunct Character** : Unique characters that are formed by the combination of two or more different characters are called conjunct or combined characters. Multiple glyph may be used to render a conjunct character.

**Dead Consonant**: The consonants at the ending of words that do not possess any inherent vowel are called Dead Consonant.

**Glyph**: Glyph refers to the shape, design or format of a character. The same character may have different glyph depending upon certain factors like font, styles, etc.

**Ligature**: Ligatures are conjunct characters that are displayed using a single glyph.

**Matra**: Matra are the non-base glyphs that are added to base glyphs in order to indicate the vowel in the syllable. They are also called diacritics. Each independent vowel has its own corresponding matra.

**NBSP**: NBSP stands for Non-Breaking Space. It is a non-printing character that prevents automatic line break at the position that it stands in.

**Post-Base Form**: In a conjunct character, post-base glyphs are those glyphs that are physically located after the base glyph.

**Pre-Base Form** : In a conjunct character, pre-base glyphs are those glyphs that are physically located before the base glyph.

**Reph**: Reph is a typographical feature used to represent the half-form of the consonant RA, when it appears at the beginning of consonant cluster.

**Script**: Script denotes the set of written symbols that represent the sounds of the spoken language using certain shapes and symbols.

**Syllable** : A syllable unit of pronunciation that has a single vowel which may or may not contain preceding or succeeding consonant

**Viram or Halanta**: The normal form of a consonant character assumes the vowel ‘a’ (‘अ’) in Devanagari. To negate this and to form pure consonant, a non-base glyph is used called Viram/ Halanta.

**ZWJ** : ZWJ stands for Zero Width Joiner. It is a non-printing character that is used to suggest that the two adjacent characters should join to form a conjunct character or a ligature.

**ZWNJ** : ZWNJ stands for Zero Width No Joiner. It is a non-printing character that is used to suggest that the two adjacent characters should not join to form a conjunct character or a ligature and should be displayed as they are.

## Appendix I : Font Specification and Recommendations

# REFERENCES

[1] K. S. Lydon, “Designing devanagari type: the effect of technological restrictions on current practice,” Ph.D. dissertation, 2015.

[2] N. Kathmandu, “Nepali font standards.”

[3] P. Hall, “Problems with unicode for languages unsupported by computers,” in 17th Annual LRC Internationalisation & Localisation Conference. Limerick, Ireland: Language Technology Kendra, September 2012. [Online]. Available: https://ltk.org.np/downloads/unicode\_problems\_for\_languages\_lrc.pdf

[4] S. B. Basnet and S. Trishna, “Unification of fonts encoding system of devanagari writing in nepali,” Nepalese Linguistics, vol. 32, no. 2, pp. 130–136, 2017.

[5] M. Boualem, M. Leisher, and B. Ogden, “Encoding script-specific writing rules based on the unicode character set.”

[6] S. P. Mudur, N. Nayak, S. Shanbhag, and R. Joshi, “An architecture for the shaping of indic texts,” Computers & Graphics, vol. 23, no. 1, pp. 7–24, 1999.

[7] Microsoft, “Shaping engine for devanagari script,” https://learn.microsoft.com/ en-us/typography/script-development/devanagari#shaping-engine, 2024, accessed: 2024-08-06.

[8] J. Shimada, “The font wars,” 2006.

[9] U. Consortium, The Unicode Standard, Version 15.1 – Chapter 12: South and Central Asia-I, https://www.unicode.org/versions/Unicode15.1.0/ch12.pdf, 2023, accessed: 2024-08-06.

[10] Richard Ishida, "Indic Scripts and Text Layout: A Comprehensive Guide," W3C Internationalization Activity, 2014.

[11] Chandra, S. (2021). Indian Typefaces in Digital Platforms: Issues and Challenges. In: Martins, N., Brandão, D. (eds) Advances in Design and Digital Communication . Digicom 2020. Springer Series in Design and Innovation , vol 12. Springer, Cham.