Date available online: 23-07-2025
Vol. 3, Issue 1 (January – June 2025)

DOI: https://doi.org/10.24312/ucp-jeit.03.01.591

RUPT: An Extension to Traditional Compilers in C++ to Support Programming in Native Language

Muhammad Ishtiaq[1,3*], **Maryam Gulzar**[2], **Muhammad Farhat Ullah.**[1]
[1]School of Software Technology, Dalian University of Technology, Dalian, Liaoning, China
[2]Independent Researcher, Dalian, Liaoning, China
[3]Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian 116620, China

* Corresponding Author: Muhammad Ishtiaq (E-mail: ishtiaqrai8@gmail.com).

ABSTRACT
The medium of instruction has a significant impact on effective communication and comprehension. Most literature is in English, but presenting information in persons´ native language improves comprehension. In computer science, source code of programming languages is written in the English language, whereas endemic language has its own impact. To address this gap, this study has rendered a framework, "Roman Urdu Programming Translator" (RUPT), that will be used to translate a program coded in Roman Urdu or Hindi into a proportionate C++ program. RUPT acts as a layer above the C++ compiler, allowing programmers to write code using Roman Urdu keywords, which it translates into standard C++ for compilation and execution. The special set of Roman Urdu keywords includes, e.g., "keyboardSay (ks)" instead of "cin", "screenKiTaraf (skt)" instead of "cout", "klea" instead of "for" etc. RUPT replaces added Roman Urdu and Hindi keywords with equivalent C++ keywords to produce valid C++ code. It is only composed of the lexical analysis phase. State of the art has increased the understanding and learning rate of novel users towards the field of computer science.

INDEX TERMS C++, Compiler, Lexical Analysis, Native Language, Roman Urdu, RUPT

I. INTRODUCTION

An accelerator is a piece of hardware or software that has as its primary objective to improve the overall performance of the computer. A variety of accelerators are available to aid in improving the efficiency of various parts of a computer's operation. Due to their high performance and energy economy, numerous specialized Deep Neural Network (DNN) accelerators have also recently become more and more popular [1]. They have been implemented in servers, data centers, and pervasive computing [2], [3]. These accelerators concentrate on particular customization for DNN computations. DNNs are typically represented as computation graphs, where nodes stand for fundamental operations (like operators, such as convolution, pooling, and activation), and edges stand for the data that these operators consume or produce. To accelerate computation, these operators can be offloaded to accelerators [1]. Accelerators facilitate making efficient use of computer resources, but we need to grease the wheels for human beings and motivate them to dive into the field of computer science. Learning the native language is very important, as it has a great impact on the education of children. It has proved its unique importance as a key factor in getting awareness of new developments in studies and success in future life. The relationship between conscious self-regulation and executive functions, two groups of regulatory predictors, and academic performance in the native language has been discovered by structural equation modeling [4]. By knowing the importance of native language in academics, in this paper, work has been done to provide a platform that will support writing source code of programming language in native language. Programming languages use English words to code. Roman Urdu is the name for writing the Urdu language in Roman characters [5]. A recently developed language in South Asia is called Roman Urdu. Romanized Urdu deviates from the rules of the Urdu language. However, many internet users utilize this language to communicate their views and ideas on a variety of topics [5], [6]. Daud [7] estimates that 300 million people use the Urdu language worldwide. Additionally, there are roughly 500 million native Hindi speakers, according to Kunchukuttan et al. [8]. The majority of them are literate in Roman Urdu. As a result, we can estimate that there are about 800 million Roman Urdu speakers [10]. Roman Urdu, which appears in the last column of Figure 1, is an example of three sentences that both Urdu and Hindi speakers may understand.

Sr. No.	English Phrase	Urdu Phrase	Hindi Phrase	Roman Urdu Phrase
1	I play	میں کھیلتا ہوں	मैं खेलता हूँ	main khelata hoon
2	I love to read book	مجھے کتاب پڑھنا اچھا لگتا ہے	मुझे किताब पढ़ना अच्छा लगता है	mujhe kitaab padhana achchha lagata hai
3	I am working	میں کام کر رہا ہوں	मैं काम कर रहा हूं	main kaam kar raha hoon

FIGURE 1: Three phrases are compared using several foreign languages

Zain et al. [19] introduced RU-OLD, a hate speech detection model for Roman Urdu that integrates deep learning, transfer learning, and hyperparameter optimization. Their study highlights the linguistic challenges associated with Roman Urdu, particularly the need for effective tokenization strategies—aspect also addressed in the present work. In this context, we propose RUPT (Roman Urdu Programming Translator), a platform designed to enable programmers to write C++ code using Roman Urdu keywords instead of standard English-based syntax. Specific keywords are

targeted in this work, and these keywords are replaced with some other keywords taken from “Urdu”. These keywords are written in “English Typeface” i.e., the keyword “cin” is replaced with “keyBoardSay” whereas “screenKiTaraf” has taken the place of the “cout” keyword. Here new libraries will be written that will be used to translate partially written programs into pure C++ programs.

The core objective of this study is to reduce the linguistic barrier for novice programmers by enabling programming in native languages—specially Roman Urdu and Hindi—using a custom compiler extension named RUPT (Roman Urdu Programming Translator). RUPT serves as a preprocessor layer that translates Roman Urdu/Hindi code into syntactically valid C++ code, facilitating compilation through standard C++ compilers.

The significance of this problem lies in the widespread difficulty non-native English speakers face while learning programming, as most programming languages rely heavily on English-based syntax and semantics. This language-centric challenges often leads to cognitive overload, disengagement and slower learning curves.

The novelty of the proposed framework lies in its integration of native-language-inspired lexical tokens within a structured grammar system (RUPL), use of Finite State Machines (FSMs) for keyword recognition, and implementation of a custom tokenizer designed to handle non-standard script variations found in Roman Urdu. Key contributions of this work include:

- Designing and formalizing the RUPL grammar with mappings for Roman Urdu equivalents of common C++ keywords.
- Implementing a light-weight lexical analyzer that processes Roman Urdu-based source code and generates corresponding valid C++ code.
- Developing a fully functional GUI-based editor that supports writing and compiling programs in Roman Urdu.
- Conducting technical and opinion-based evaluations with students to validate ease of learning and system usability.

The rest of the article is organized in the following manner: In Section II, relevant work is elaborated. Section III presents the methodology used for state-of-the-art. Results and discussion, along with design and implementation, are covered in Section IV, while the conclusion is covered in Section V.

II. LITERATURE REVIEW

Over the last few years, several researchers carried out different studies for code conversion. David Unga et al. [9] proposed a computer-adaptive translator named the "University of Queensland Dynamic Binary Translator" (UQDBT) that followed a backward pass (decoding executable code) and forward pass (encoding the decoded executable code after required improvement). UQDBT converges faster than systems based on instruction anticipation because edge weight instrumentation converts frequently executed code to native code. It was a method that made it possible to run software on a machine and get acceptable output, whereas the software was designed for some other machine.

Tao Lei et al. [11] described a technique for creating input parsers automatically from English specifications of input file formats. The English specification was converted into a specification tree, which was then converted into a C++ input parser using a Bayesian generative model to capture pertinent natural language occurrences. A joint dependency parsing and semantic role labeling task was used to model the issue. Their approach is based on two different types of data: the first is the relationship between the text and the specification tree, and the second is noisy supervision, which is measured by how well the resulting C++ parser reads input examples. A state-of-the-art semantic parser obtained an F1-score of 66.7% using a dataset of input format specifications from the ACM (Association for Computing Machinery) International Collegiate Programming Contest, while this technique produced an F1-score of 80.0%.

Furqan et al. [18] released ERUPD, a parallel English-Roman Urdu corpus of 75,146 sentence pairs, created via synthetic prompt-engineering and human validation. This dataset could significantly enhance token mapping and keyword consistency in state-of-the-art related work. Besides, Ansarullah et al. [20] achieved 97.98% accuracy in segmenting mixed Roman Urdu and English text using dictionary based SVM and Bi-LSTM—highlighting effective strategies for script or code-switching detection application applicable to lexical analyzers.

RECCO, a REliable Code COmpiler that can automatically produce a reliable version of any C/C++ source code, was introduced by A. Benso et al. [12] The program uses a powerful algorithm for reordering the code and a flexible technique for variable duplication to build a trustworthy code that can recognize the appearance of important data defects. The tool makes changes that are entirely transparent to the programmer and have no impact on the targeted program’s original functionality. In order to keep overhead within the acceptable ranges, the tool also gives the user the option of selecting the percentage of duplicated variables. The approach’s efficacy and the low overhead that was added to the trustworthy code in terms of both memory occupancy and execution time were proved by experimental results.

Ben Gelman et al. [13] linked source code with three deliverables from 108,568 projects that were downloaded from GitHub and had at least 10 stars and a redistributable license. The first set of pairs links Doxygen-extracted comments with corresponding snippets of source code in C, C++, Java, and Python. The second group of pairings links the raw C and C++ source code repositories with the build artifacts that are

produced when the make command is used to create the code. The last set of pairs links unprocessed C and C++ source code repositories with probable code flaws, which are discovered by running the Infer static analyzer. The code and comment pairs can be utilized for tasks like comment prediction or code description in natural language. Reverse engineering and enhancing intermediate representations of code from decompiled binaries are two operations that can be accomplished using the code and build artifact pairs. It is possible to leverage the code and static analyzer pairs for machine learning approaches to vulnerability finding.

According to real-world applications and the drawbacks of existing programming software, Pan Duotao et al. [14] created the GIEPT, a type of cross-platform modelling programming software, on the open source Fedora 12/Linux platform. By utilizing its XML-based input programming approach and many programming kinds and scales, GIEPT offers a universal solution. The GIEPT now incorporates a number of features, such as the common solver interface, the unified standard using XML-based schema, which is used to input model data and automatically convert it into C++ source code, and the symbolic algebra system, which is used to produce the matrix of gradient, Jacobian, Hessian, etc. Since GIEPT is platform agnostic and open source, programmers are free to alter its characteristics and expand its functionality in response to the situation at hand. Additionally, development and application expenses have been significantly decreased.

In a C++ implementation of a concordance program for texts in Old West Norse and Runic Swedish, Lars Engebretsen [15] discussed some of the author's experiences. It was only reasonable to use Unicode to represent data both inside the program and in external files because the input to the program employed a character repertoire that no typical onebyte character encoding supports. The input and output were represented in UTF-8, while each character within the program was represented using C++ "wide characters." During file I/O, the author created C++ code conversion aspects that translate data between those two formats. This allowed him to successfully construct and execute the concordance application on both Windows XP (using Visual C++.NET 2003) and Linux (Fedora Core 3 with gcc 3.4.2). When switching platforms, only a few lines of code—the ones deciding which code conversion facet to use—had to be modified in the source code; all other sections of code stayed the same. Even though the code conversion facets given by the library had been updated, the author could still use the standard C++ locale framework for collation and code conversion.

A method for automatically creating documentation summaries for C++ procedures was suggested by Nahla J. Abid et al. [16]. A summary template was made using method stereotypes, one for each individual method archetype. The primary parts of the approach are then extracted using static analysis. The generated documentation summaries are then used to update each method's documentation. The strategy may be applied to various object-oriented programming languages and is very scalable. These summaries can aid in maintaining comprehension. Undergraduate students that participated in the evaluation were the initial subjects. The findings show that the automated summaries adequately describe what the approach accomplishes and contain all necessary details. The outcomes also suggest that their approach to this issue—creating unique templates for each stereotype—is a workable and effective remedy. Despite the fact that the automated summaries were generally praised by the participants, some changes are still required, notably for the controller and collaborator, because they are rather complex and challenging to effectively summarize.

III. METHODOLOGY

Butt et al. [17] propose a transformer-based model using m2m100 with masked language modeling to transliterate between Roman-Urdu and Urdu, achieving character-level BLEU scores of 96.37 and 97.44. Their use of transfer learning and rigorous domain adaptation offers a strong precedent for lexical mapping approaches in Roman Urdu Programming Translator (RUPT). RUPT will translate a program containing Urdu Roman words as an alternative to C++ keywords into a pure C++ program. It is able to perform lexical analysis of the program containing a mixture of Urdu Roman words and C++ keywords according to the definition of Roman Urdu Programming Language (RUPL).

A. RUPL DEFINITION

Roman Urdu Programming Language (RUPL) contains the following alphabets as C++:

{A, B, C, D, E, F, G, H, I, J, K,
L, M, N, O, P, Q, R, S, T, U, V,
W, X, Y, Z, a, b, c, d, e, f, g,
h, i, j, k, l, m, n, o, p, q, r,
s, t, u, v, w, x, y, z, 0, 1, 2,
3, 4, 5, 6, 7, 8, 9}

Real numbers can be defined by the following alphabets:

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .}

TABLE 1: RUPL Keywords

C++ Keyword	RUPL Keyword	C++ Keyword	RUPL Keyword
cin	keyboardSay	cout	screenKiTaraf
if	agar	else	naheTw
for	klea	switch	badlo
case	imkan	break	roko
default	pehlySay	do	karo
exit	niklo	private	niji
public	awami	protected	mehfooz
while	jabK	return	wapis
continue	jari	string	doree

This table presents the mapping between C++ keywords and their equivalents in the proposed RUPL language.

The alphabets given above have been used to make grammar for our language, RUPL. Grammar has four

parts given below:

1) **N:** non-terminal alphabets
2) **T:** terminal alphabets
3) **P:** it defines production rules
4) **S:** it is start symbol belongs to N

Here is an example to define grammar for an identifier:

N = { <id>, <digit>, <letter>}
T = { 1 , 2 , 3 , a , b , c }
P = production rules to be used
1. <id> → <letter>
2. <id> → <id> <letter>
3. <id> → <id> <digit>
4. <digit> → 1
5. <digit> → 2
6. <digit> → 3
7. <letter> → a
8. <letter> → b
9. <letter> → c
S = <id>

B. RUPL KEYWORDS

Besides, a subset of C++ keywords has been selected. Among these, some keywords will be redefined into the Roman Urdu language. These Urdu Roman-redefined keywords will be replaced back into the original C++ by RUPT. Table 1 shows C++ keywords and their corresponding special RUPL keywords.

C. RUPL TOKENS

RUPT tokens are the fundamental structures that obstruct the Roman Urdu Programming Language, which are developed together to compose a RUPL program. Every single littlest individual unit in a RUPL program is known as a RUPL token. A few types of RUPL tokens given below:

1) Keywords, e.g., string, klea
2) Identifiers, e.g., total, main
3) Strings, e.g., school, university
4) Constants, e.g., 1001, 1136, 1089
5) Operators, e.g., *, -, +, /
6) Special symbols, e.g., { }, ( )

D. RUPL TRANSLATION

There are two essential qualities of simulated dialects: syntax and semantics. Language structure is an arrangement of standards that must be taken after to announce a legitimate program, while semantics depicts consistent conduct of the substantial program. The way toward contrasting source code and the punctuation of dialect is finished by the parser, while the code generator allocates implications to the program. Strategies used to determine the linguistic structure of any dialect are grammar, finite state machines, and regular expressions.

As discussed earlier, RUPL grammar has four parts: *N* , *T* , *P* , and *S*. An element from the non-terminal letter set, *N* , speaks to a gathering of characters from the terminal letters in order, *T* . A non-terminal image is as often as possible encased in edge sections, <>. While the guidelines of generation utilize the non-terminal to portray the structure of the language. Notice that *N* is a set, yet *S* is not. *S* is one of the components of set *N* . The beginning image, alongside the tenets of creation, *P* , empowers you to choose whether a series of terminals is a substantial sentence in the dialect. In the case of beginning from *S*, a series of terminals is created by utilizing the principles of generation; at that point the string is a legitimate sentence.

1) Grammar for Identifier

Despite the fact that a RUPL identifier can utilize any capitalized or lowercase letter or digit, the same as C++, to keep the case little, this punctuation allows just the letters l, m, and n and the digits 7, 8, and 9. The principal character must be a letter, and the rest of the characters, assuming any, can be letters or digits in any mix. This grammar has three non-terminals, namely, < *id* >, < *letter* >, and < *digit* >. The start symbol is < *id* >, one of the elements from the set of non-terminals. The rules of production are of the form: **A** −−· **w**, where *A* is a non-terminal and w is a string of terminals and non-terminals. The symbol −−· means “produces” while the grammar specifies the language by a process called a derivation. To derive a valid sentence in the language, begin with the start symbol and substitute for non-terminals from the rules of production until you get a string of terminals. Here is a derivation of the identifier nlm9 from this grammar. The symbol means −−· “derives in one stage”. Grammar for RUPL identifier is given below:

N = { <id>, <letter>, <digit>} T = { l , m , n , 7 , 8 , 9 }
P = shows rules of production
1. <id> → <letter>
2. <id> → <id> <letter>
3. <id> → <id> <digit>
4. <digit> → 7
5. <digit> → 8
6. <digit> → 9
7. <letter> → l
8. <letter> → m
9. <letter> → n S = <id>

Besides, each deduction step serves as the generation administrator upon which substitutions are based. For example, consider Rule 2:

$$\langle id\rangle \text{ --·} \langle id\rangle\langle letter\rangle$$

This rule is applied to substitute ⟨*id*⟩during the derivation stage. For instance:

$$\langle id\rangle 9 \text{ --·} \langle id\rangle\langle letter\rangle 9$$

The conclusion of this inference operation corresponds to performing substitution on a letter in sequence. The symbol
−−·* denotes "derives in zero or more steps". The last eight inference steps can be summarized as:

$$\langle id\rangle \text{ --·*} \text{ nlm9}$$

This derivation confirms that nlm9is a valid identifier, as it

can be generated from the start symbol ⟨*id*⟩.

2) Finite State Machine (FSM) to Parse Identifier

A 2024 study [21] on Roman Urdu spelling variation (5˜244 words per variant) emphasizes the prevalence of orthographic inconsistency—supporting RUPT's FSM-based normalization to correctly map variant tokens to standardized C++ keywords. In Figure 2(a), the arrangement of states {*A, B, C*} is given. *A* is the beginning state and *B* is the last state, whereas *C* is the reject state. There is progress from *A* to *B* on a letter, from *A* to *C* on a digit, from *B* to *B* on a letter or a digit, and from *C* to *C* on a letter or a digit. To utilize the FSM, envision that the information string is composed on a bit of paper tape. Begin in the beginning state, and output the characters on the information tape from left to right. Each time you examine the following character on the tape, influence a change to another condition of the limited state machine. Utilize just the change that is permitted by the curve relating to the character you have recently checked. Subsequent to filtering all the info characters, on the off chance that you are in a last express, the characters are a legitimate identifier. Else they are most certainly not. Figure 2(b) shows the same process through a simplified finite state machine by removing the optional reject state. Table 2 and 3 show transition tables for FSM Identifier and Identifier through Simplified FSM, respectively.

TABLE 2: Transition Table for Identifier

Current State	New State (Letter)	New State (Digit)
A B C	B B C	C B C

This table defines state transitions when processing identifiers: a letter leads to one transition, while a digit leads to another, based on the current state.

TABLE 3: Transition Table for Identifier through Simplified FSM

Current State	New State (Letter)	New State (Digit)
A B	B B	– B

This table shows a simplified finite state machine (FSM) used to identify valid identifiers. State transitions depend on whether a letter or digit is encountered.

By considering the following grammar, a few examples to parse RUPL keywords are represented.

X = { a, b, c, e, f, g, h, I, k, l, n, o, r, s, t, y }
T = { X, _ }
N = { <id>, <letter>, <symbol>} P = Production Rules
<id> → <id> <letter>
<id> → <id> <symbol>
<id> → <letter>
<letter> → X
<symbol> → _ S = <id>

Now, the key word "agar" will be parsed through the grammar defined above. As it is defined:

S = <id>
= <id><letter> (using Rule 1)
= <id><letter><letter> (using Rule 1)
= <id><letter><letter><letter>(using Rule 1)
= <letter><letter><letter><letter> (using Rule 3)
= agar

The keyword "agar" has been proved by RUPL grammar of keywords. Hence, "agar" belongs to RUPL. Figure 3 shows the FSM for the RUPL keyword "agar". The same as the keyword "klea" also being tested. So, the key word "klea" will be parsed through the grammar defined above. Parsing of the "klea" keyword is performed by applying the same rules as applied to parse "agar" because both keywords have 4 letters. Figure 4 shows FSM to parse RUPL keyword "klea".

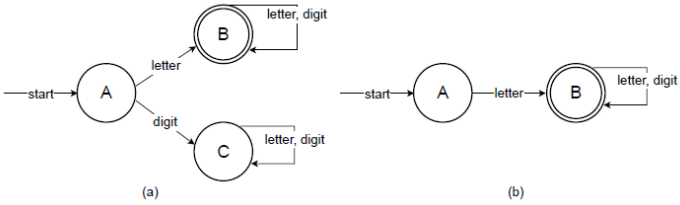

FIGURE 2: Finite State Machine (FSM) to Parse Identifier

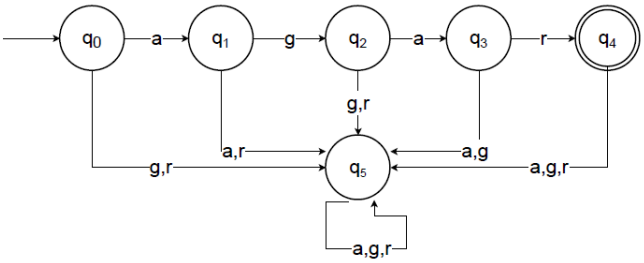

FIGURE 3: FSM to Parse RUPL keyword "agar"

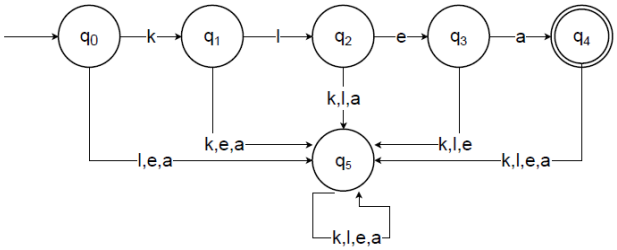

FIGURE 4: FSM to Parse RUPL keyword "klea"

The Algorithm 1 outlines how the RUPT lexical analyzer processes Roman Urdu keywords, identifieres, constants, and symbols to generate valid C++ tokens:

Algorithm 1 RUPT Lexical Analyzer with Spell Correction

Input: Source code written in RUPL (Roman Urdu Programming Language)
Output: Translated source code in C++ syntax
Load predefined `keywordMap` containing RUPL → C++ mappings
for all lines in RUPL source code **do**
  Tokenize each line into individual tokens $t_1, t_2, \ldots, t_n$
  for all token t_i **do**
    if $t_i \in$ keywordMap **then**
      Replace t_i with keywordMap[t_i]
    else
      Initialize $minDist \leftarrow 3$, $closest \leftarrow$ ""
      for all valid keyword k in keywordMap **do**
        $d \leftarrow \text{LevenshteinDistance}(t_i, k)$
        if $d < minDist$ **then**
          $minDist \leftarrow d$; $closest \leftarrow k$
        end if
      end for
      if $closest \neq$ "" **then**
        Replace $t_i \leftarrow$ keywordMap[$closest$]
        Display correction suggestion to user (optional)
      end if
    end if
    Append translated token to output
  end for
end for
Return the final translated C++ code

IV. RESULTS AND DISCUSSION

The benefit of the iterative model is that it facilitates the early development of a functional version of the product. As a result, implementing modifications is less expensive. This is the reason to follow the iterative development model for RUPT.

A. REQUIREMENTS

The requirement under which this framework is presented is to target those audiences facing problems in understanding the English language. It is known by everyone that the usage of compilers itself has all the keywords in the English language. That is the main reason RUPT is presented. To make the lives of those people easy by changing specific keywords. As a result, their programming skills won't be affected in any way. Other than that, during the process of the translation, natural language is also promoted, which results in better understandings and perceptions about the people that are doing this.

B. DESIGN

As everyone knows, a translator or compiler requires some keywords and a list of tokens by which it recognizes the source code and compares it with the grammar defined for the programming language and translates that programming language to another. Similarly, a list of keywords, presented in Table 1, has been designed and targeted by RUPT. The user writes RUPL source code in the RUPT editor and gets the required output as pure C++ code, and later on this code is passed to the traditional compiler for further processing. Figure 5 shows an abstract view of RUPT, while Figure 6 helps to present a detailed understanding of the state-of-the art translator.

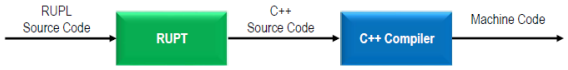

FIGURE 5: Abstract view of RUPT

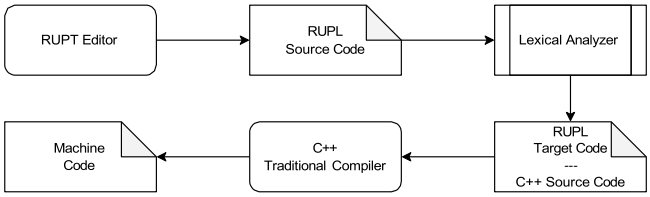

FIGURE 6: RUPT Detailed View

C. IMPLEMENTATION

After designing the RUPT, it was implemented into reality. For this purpose, C++ and C# have been used. DevCpp and Visual Studio are the main tools that have been used to implement the design. The RUPT editor has been developed as a user interface by writing instructions in C# as a programming language. As a tool, we have used Visual Studio to implement C#. Figure 7 shows the RUPT document window having RUPL source code and a RUPT interface. Source code to perform the process of converting a sequence of characters into a sequence of lexical tokens, called lexical analysis, is written in the C++ programming language. RUPT gives pure C++ code after performing lexing or tokenization on RUPL source code. Figure 8 elaborates on the working of the lexical analyzer.

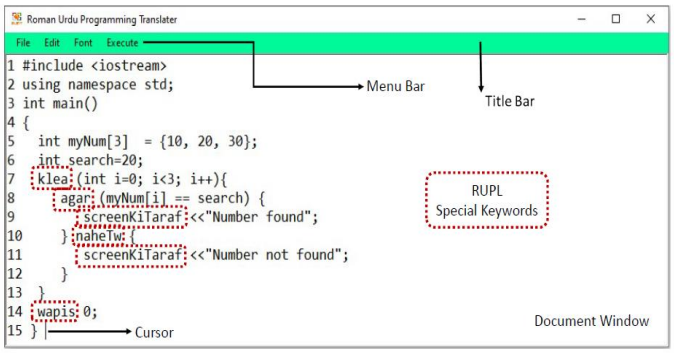

FIGURE 7: RUPT Interface

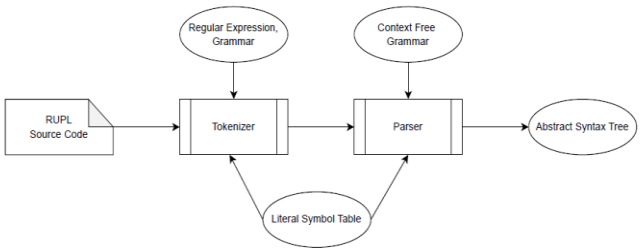

FIGURE 8: Lexical Analyzer Working

D. PROGRAM USED FOR EVALUATION

To verify the correctness of RUPT's translation output, a standard merge sort algorithm was used in Roman Urdu and then translated into C++ using the framework. The translated code was compiled and executed using a traditional C++ compiler. The Algorithm 2 shows the logic of the merge sort implementation used for evaluation:

Algorithm 2 Merge Sort Algorithm

```
Input: Array A, left index l, right index r
Output: Sorted array A
MERGESORT(A, l, r)
if l < r then
    m ← ⌊(l + r)/2⌋
    MERGESORT(A, l, m)
    MERGESORT(A, m + 1, r)
    MERGE(A, l, m, r)
end if

MERGE(A, l, m, r)
    n1 ← m − l + 1
    n2 ← r − m
    Create temporary arrays L[1 . . . n1] and R[1 . . . n2]
    for i = 1 to n1 do L[i] ← A[l + i − 1]
    for j = 1 to n2 do R[j] ← A[m + j]
    i ← 1, j ← 1, k ← l
    while i ≤ n1 and j ≤ n2 do
        if L[i] ≤ R[j] then A[k] ← L[i]; i ← i + 1
        else A[k] ← R[j]; j ← j + 1
        k ← k + 1
    end while
    while i ≤ n1 do A[k] ← L[i]; i ← i + 1; k ← k + 1
    while j ≤ n2 do A[k] ← R[j]; j ← j+1; k ← k+1
```

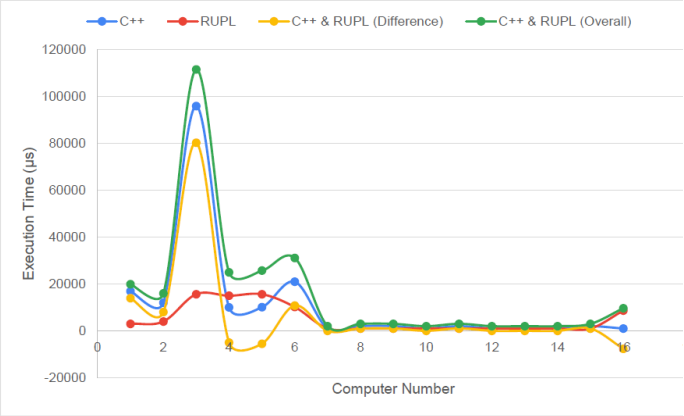

FIGURE 9: Sample View of Technical Servery

E. VERIFICATION

The community targeted to evaluate the state of the art was novel computer users towards programming, especially the intermediate students. The assessment process was based on two types of surveys. 1) Technical Survey and 2) Opinion Survey. In the technical survey, analysis was done by calculating the execution time of several programs, especially the merge sort program, by different users on many computers with variant aspects, e.g., CPU: Core, Frequency, and Generation. First of all, a pure C++ merge sort program was executed as aforesaid and calculated its execution time in microseconds. After that, a calculation was performed for the execution time taken by RUPT to yield a pure C++ program for merge sort from RUPL-based source code. A little bit of an increment in overall execution time was observed, but the understanding and learning rate increased as new learners found ease towards program scripting. A sample view of the technical servery is presented in Figure 9.

The opinion survey was conducted using both hard copy and digital forms, targeting intermediate-level and early undergraduate students from multiple educational institutions. Prior to participation, students were introduced to the use of RUPT in conjunction with RUPL for scripting, and its comparison with other programming languages. They were then asked to share their perspectives. The survey included students from both colleges and universities, representing a diverse academic background. The results based on responses from different institution types are illustrated in Figure 10.

In Figure 11, overall results present that 83.40% of participants appreciated RUPT as a great initiative to motivate individuals to the field of computer science, 8.30% said that in their views it does not affect them, while the same percentage remained neutral.

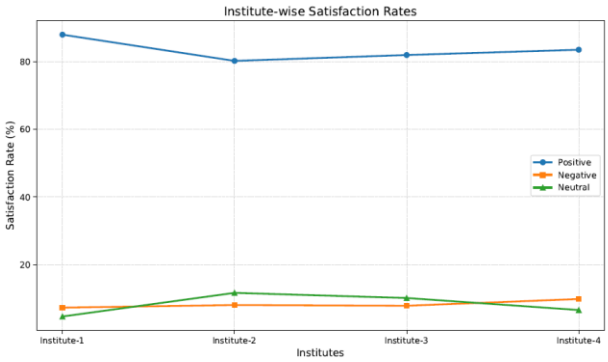

FIGURE 10: Views of Each Institute from Opinion Survey

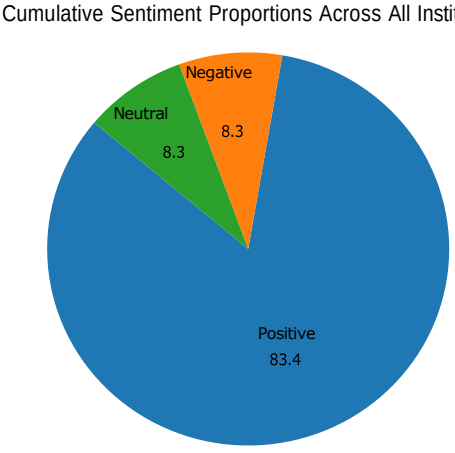

FIGURE 11: Overall Views from Opinion Survey

V. CONCLUSION

This work has proffered a framework, "Roman Urdu Programming Translator" (RUPT), as an additional layer to the original C++ compiler that translates a program coded in Roman Urdu or Hindi, known as "Roman Urdu Programming Language" (RUPL), into a proportionate C++ program. In this study, a special set of Roman Urdu keywords includes, e.g., “keyboardSay (ks)” instead of “cin”, “screenKiTaraf (skt)” instead of “cout”, “klea” instead of “for” etc., is focused on. In this work, additional Urdu and Hindi Roman keywords are added to the original set of C++ language and replaced by the C++ equivalent keywords through RUPT to convert it into a pure C++ program. RUPT is only composed of the lexical analysis phase. Keywords and tokens are defined and parsed by following the rules delineated in RUPL grammar. Evaluation is based on two types of surveys: 1) The technical survey has observed a minor increment in overall execution time, but the understanding and learning rate increased. 2) The opinion survey has presented that 83.40% of participants appreciated RUPT as a great initiative to motivate novel users to the field of computer science, 8.30% said that in their views it does not affect them, while the same number of participants remained neutral.

ACKNOWLEDGMENT

We would like to express our gratitude to the anonymous reviewers for their insightful feedback. This research work is not funded by anyone. Gramatical corrections are done by using AI at some places. Besides, authors do not have any conflict of interest.

REFERENCES

[1] J. Li, W. Cao, X. Dong, G. Li, X. Wang, P. Zhao, L. Liu, and X. Feng, “Compiler assisted Operator Template Library for DNN Accelerators,” *International Journal of Parallel Programming*, 2021, doi: 10.1007/s10766021-00701-6.

[2] N. P. Jouppi et al., “In Datacenter Performance Analysis of a Tensor Processing Unit,” *ISCA ’17*, pp. 1–12. Association for Computing Machinery, New York, NY, USA, 2017, doi: 10.1145/3079856.3080246.

[3] H. Liao, J. Tu, J. Xia, and X. Zhou, “DaVinci A Scalable Architecture for Neural Network Computing,” *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–44. IEEE Computer Society, Los Alamitos, CA, USA, 2019, doi: 10.1109/HOTCHIPS.2019.8875654.

[4] V. I. Morosanova, I. N. Bondarenko, T. G. Fomina, and B. B. Velichkovsky, “Executive Functions and onscious Self-Regulation as Predictors of Native Language Learning Success in Russian Middle School Children,” *Journal of Siberian Federal University. Humanities & Social Sciences*, 2021, doi: 10.17516/1997-1370-0824.

[5] M. Daud, R. Khan, Mohibullah, and A. Daud, “Roman Urdu Opinion Mining System (RUOMiS),” *Computer Science & Engineering: An International Journal (CSEIJ)*, 2015, doi: 10.48550/arXiv.1501.01386.

[6] K. Mehmood, D. Essam, and K. Shafi, “Sentiment Analysis System for Roman Urdu,” *Proceedings of the 2018 Computing Conference*, 2018.

[7] A. Daud, W. Khan, and D. Che, “Urdu language processing, a survey,” *Artificial Intelligence Review, An International Science and Engineering Journal*, 2016, doi: 10.1007/s10462-016-9482-x.

[8] A. Kunchukuttan, P. Mehta, and P. Bhattacharyya, “The IIT Bombay English-Hindi Parallel Corpus,” *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[9] D. Ung and C. Cifuentes, “Dynamic binary translation using runtime feedbacks,” *Science of Computer Programming*, 2005, doi: 10.1016/j.scico.2005.10.005.

[10] U. Hayat, A. Saeed, M. H. K. Vardag, M. F. Ullah, and N. Iqbal, “Roman Urdu Fake Reviews Detection Using Stacked LSTM Architecture,” *SN Computer Science*, 2022, doi: 10.1007/s42979-022-01385-6.

[11] T. Lei, F. Long, R. Barzilay, and M. Rinard, “From Natural Language Specifications to Program Input Parsers,” *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2013.

[12] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri, “A C/C++ sourceto-source compiler for dependable applications,” *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, 2000, doi: 10.1109/ICDSN.2000.857517.

[13] B. Gelman, B. Obayomi, J. Moore, and D. Slater, “Source code analysis dataset,” *Data in brief*, 2019, doi: 10.1016/j.dib.2019.104712.

[14] P. Duotao, H. Mingzhong, and Y. Decheng, “Development of large scale programming system based on Linux platform,” *2011 Chinese Control and Decision Conference (CCDC)*, 2011, doi: 10.1109/CCDC.2011.5968310.

[15] L. Engebretsen, “Platform-independent code conversion within the C++ locale framework,” *Software — Practice and Experience*, 2006, doi: 10.1002/spe.734.

[16] N. J. Abid, N. Dragan, M. L. Collard, and J. I. Maletic, “Using stereotypes in the automatic generation of natural language summaries for C++ methods,” *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, doi: 10.1109/ICSM.2015.7332514.

[17] U. Butt, S. Veranasi, and G. Neumann, “Low-Resource Transliteration for Roman-Urdu and Urdu Using Transformer-Based Models,” *arXiv preprint arXiv:2503.21530*, 2025.

[18] M. Furqan, R. B. Khaja, and R. Habeeb, “ERUPD–English to Roman Urdu Parallel Dataset,” *arXiv preprint arXiv:2412.17562*, 2024.

[19] N. Hussain, A. Qasim, G. Mehak, O. Kolesnikova, A. Gelbukh, and G. Sidorov, “ORUD-Detect: A Comprehensive Approach to Offensive Language Detection in Roman Urdu Using Hybrid Machine Learning– Deep Learning Models with Embedding Techniques,” *Information*, vol. 16, no. 2, p. 139, 2025, doi: 10.3390/info16020139.

[20] S. H. Kumhar, M. Kirmani, S. Alshmrany, *et al.*, “Language Tagging, Annotation and Segmentation of Multilingual Roman Urdu-English Text,” 2024. [Online]. Available: https://arxiv.org/abs/ (DOI or publisher not available).

[21] M. A. Soomro, R. N. Memon, A. A. Chandio, M. Leghari, and M. H. Soomro, “A dataset of Roman Urdu text with spelling variations for sentence level sentiment analysis,” *Data in Brief*, vol. 57, p. 111170, 2024, doi: 10.1016/j.dib.2023.11117