C to Python translator using LEX & YACC

B Sai Abhishek, Balam Ruchith Balaji, Chillakuru Hari, Meena Belwal
Department of Computer Science and Engineering, Amrita School of Computing,
Bengaluru, Amrita Vishwa Vidyapeetham, India.
abhishek.busetty@gmail.com, bl.ruchith@gmail.com,
Hari291010reddy@gmail.com, b_meena@blr.amrita.edu .

Abstract— Our goal is to use Yacc and Lex in our project that is geared towards establishing a C to a Python translator. The mission of such translation assignment is to change C codes to Python ones smoothly, ensuring that all functions and logic remain intact. In order to minimize the problems of lexical divergence between both languages, our approach defines the rules for adequate tokenization and parsing through utilizing of Lex and Yacc. We make our effort is to derive the same exact code, in the Pythonic syntax and style, of the original C logic. As a team, we strive to provide a high-quality translator tool that builds in switching between code written in C and Python, which will be made easier through mutual effort and thorough testing.

Keywords— Translation, Lex, Yacc, C, Python, Parsing, Conversion

I. Introduction

With a huge increase in the need for efficient language translation technologies during the past years mostly due to the demand from software developers it has become one of the key fields. The main topic of study is the transition from C programming language to Python. There are scholars who as well as programming practitioners are implicating the number of ways of doing the conversion. Among the noteworthy studies by Freda Shi1 et al. [2] that investigates several existing techniques for C translation to Python and takes a closer look into positive or negative sides of them. Last but not least Jones and Brown presented offer a linear approach for the programmers to use Lex coupled with Yacc in regard to automating the translation process, thus providing new insights into the aspects associated with the applicability and expediency of such tools.

Audio and video records have a huge role in the world of historiography because they provide a more accurate picture of historical events. The handling of syntax deviation between C and Python poses the obstacle that has been indicated as one of the main challenges that have been summarized in research titled "Prospects and Challenges of Converting C Code to Python" (2019) by Lee and Kim. This paper stresses that finding a perfect reconciliation between languages becomes hard because the striking disparities in syntax are revealed and attempts to find the solution are made.

This research project aims to find the solution of how C code is converted to another Python code in the reliable and effective way. Our desired outcome is to construct a much potent translator that will generate the exact same code as its original C code but maintaining its logic and performance, by exploiting the powers of Lex and Yacc. The intention of this study is to evaluate how competently this technique functions toward bridging syntactical gaps and also enables smooth conversion from Python to C.

The format of the paper is as follows: An overview of the field and the difficulties in translating from C to Python are given in Section 1. In Section 2, a thorough review of the literature is presented, with publications already published categorized according to the techniques used. In Section 3, we go over the problem statement and context while emphasizing areas that still need work and highlighting current models. Section 4 presents our suggested approach for translating C to Python with Yacc and Lex. The experimental specifications, analysis, dataset selection, and evaluation parameters are described in Section 5. The experimental data are finally shown in Section 6, along with comparative tables and graphs..

II. Literature Review

In this research Marie-Anne et al. [1] deals with unsupervised machine translation paradigms for development of complete cross-language program translators. Model strives to utilize public Github repositories to perform such translation operations as conversion C++, Java or Python with accuracy. The model is more accurate and perform better than the existing commercial tools, it was more cost effective and require less time to perform the task.

Freda Shi1 et al. [2] constructed a Python-to-C translator with Lex and Yacc entailed developing an output terminology, completing transcripts, semantic examination, and syntax processing. It should be noted that this is a very delicate part of the writing process that requires developers' focus to get the needed result-the planned accuracy and readability.

Walid Hariri et al. [3] wrote an article looks at ChatGPT, a massively efficient language AI model used in the development of chatbots, content generation and medical diagnosis, and also the drawbacks that the model has such as biased replies, select indefinite statements, slips of phrasings and the unclear sentence structure, often leading to confusion and misunderstanding.

Marie-Anne et al. [4] presented a study unveils a new pre-training goal called DOBF, which implicitly uses native language constructs to advance the development of a technique targeting obfuscated scanning in an unassisted atmosphere, outperforming the exiting ones.

Syed Abdul Basit Andrabi et al. [5] written an essay is on the machine translation mechanisms that are used in resource-poor languages, clarifying problems and needs, and then review the current systems that are available for such languages.

Wasi Uddin Ahmad et al. [6] proved that translation back is a big help for neural machine translation, anyway, it seems that multilingual pre-trained sequence-to-sequence models cannot be trained further. The new method of back translation including code summarization and generation will be proposed, which had the performance like that of the best methods.

The article by Kusum et al. [7] is analyzing the popularity of efficient approaches for language translation services, which is because the ancient system needs to communicate with the recent ones and shows the subtlety matter of those techniques.

Mikel Artetxe et al. [8] developed a technique relies on a module-based approach that consists on the combination of a phrase table with an n-gram nuclear model and the fine-tuning of the parameters. Iterative backtranslation is yielding results, generating improvements of up to 7-10 BLEU points for the shown systems against previous unsupervised systems.

Baptiste Rozière et al. [9] developed a source code translation method that are unsupervised and then those produce a noisy and error-prone result. For dealing with this, a parser of a neural network for unit-testing looks through to delete wrongly translated fields from the corpus thus assembling a quality checked parallel corpus.

Fang Liu et al. [10] created a SDA-Trans is syntax and domain-specific model that boosts cross-lingual begeraidability surpassing large-scale of pre-trained models especially for easier multilingual translation programs that are written in Python and Java Momowski.

The work by Mikel Artetxe et al. [11] proposes an unsupervised approach to create an unsupervised NMT system trained solely on monolingual data, which makes it free from the requirement of parallel corpora. The device is able to perform to an equal level quality of former and has the capability of small volumes of parallel data.

Mikel Artetxe et al. [12] analyzed that Modern studies in unsupervised SMT methods uncover inadequacies by using subword knowledge, a theoretically sound unsupervised tuning method, and introduce multi-parametric alignments to improve translations.

Zuchao Li et al. [13] suggests a model for unsupervised neural machine translation (UNMT) that is reference language-based, called RUNMT, that leads to source-target language paradigm pass which has been extended. Experiments provide findings for better quality of UNMT than a baseline with just one significant auxiliary language which is not user interactive.

The research by Iftakhar Ahmad et al. [14] is looking to machine translation using neural networks to facilitate cross-architecture binary code analysis for research in the computer security field. At the end, the model UNSUPERBINTRANS is set into the workbench and the outcome proved to be of the highest level in the tasks of code similarity detection and vulnerabilities finding.

Wasi Uddin Ahmad et al. [15] proved that program translation is the key factor in software development, but unsupervised strategies are coming to the limit. AVATAR, a set of 9,515 JAVA and Python programming problems of 10 lines each, shows that pre-trained language models are not functionally accurate.

The authors Jesse Michael Han et al. [16] shows using generatively pre-trained language models, the unsupervised frameworks achieve an METEOR score of 97.3 on the WMT14 for the English-French datasets, which is state-of-the-art compared to supervised models in current SOTA.

The research by Aniketh Malyala et al. [17] deals unsupervised machine learning-based program translations failures and develops a rule-herminder base program mutation engine. It demonstrates the possibility that this model can be combined with current code processing tools and translators into a new hybrid translator which will provide the translation performance that exceeds the current state-of-the-art.

The thesis by Akila Loganathan et al. [18] does the work supervised learning techniques, particularly the clustering method K-Means, to pretreat and analyze source code before the target language translation through rules-based approach. The study is oriented towards the migration from C++ to Java exclusively. It implemented some practical translation rules that are evaluated by the basic accuracy metrics equal to 77.89% and 81.34% in comparison with other similar approaches. The research seeks to validate the effectiveness of individualized migration programs and gauge their achievements.

The article by Miller et al. [19] outlines CoST, a multilingual Code Snippet Translation corpus as well as a new multilingual program translations model based on the combination of multilingual snippet denoising auto encoding and multilingual translation pre-training. Our method enables better translation performance for program-related languages low on resources than translation methods with their basic level do.

Chandan K. Reddy et al. [20] suggested a strategy that is going to implement the original language's IRLLVM IR to increase accuracy of the translation process, for example languages such as C++, Java, Rust, Go. The method increases the F1 score of machine translation by 11% and 79% for unsupervised and supervised cases. It also extends the evaluation sets and models.

Likhith et al. [21] developed a tool which operates on Lex and YACC is designed so as to translate and perform mathematical operations as English phrases, thus making it possible for someone to do computation using only English phrases, and then converting such phrases into executable code.

Pecheti et al. [22] suggested that Abstract Syntax Trees (AST) could be done perfectly to assess the mathematical expressions. The method can be used for the professional and academic applications. It is perfect for solving equations and accommodate multiplication, division and etc., it is what we are talking. AST model hierarchy implementation results to a greater of assessment effectiveness. Interactive graph visualization may make things easier, breakdown complex computation and engineering tasks, and is crucial to mathematical expression recognition.

The translation of code is indeed an important factor that is determined by efficiency. Investigation could be carried out to find methods to make the translation process be made more effective and result to a code that is both efficient and idiomatic. The creation of the effective error handling methods and the debugging tools for the translation process is a must. Investigations into the ways of giving informative error messages and helping developers to debug translated code can be conducted through research.

III. METHODOLOGY

Lexical Analysis (Lex/Flex):

Using tools like Lex or Flex to tokenize the input C code. The fundamental elements of C language called “tokens” are separated into classification as the operator, keyword, and identifier. The step where you will precisely distinguish token type you will be to apply regular expressions.

.

Syntax Analysis (Yacc/Bison):

As for defining grammar rules for C language, use Bison or Yacc. Using the tokens produced with the help of Lex/Flex and the grammar rules given, build the parse tree. Help with aesthetics such as conditionals, loops, statements, expressions.

Semantic Analysis:

The processed C code parsing must be checked for its semantics. The operations to be done are checking function definitions, variable declarations, and type compatibility, whereas the mistakes that may arise are undefined variables and incompatible types that should be corrected.

Code Transformation:

Walk down the hierarchy and every C construct should be converted into its equivalent in Python. Store and use the mappings of C constructs to those expressions in Python language. To this, one may mention the indentation in Python, along with the fact that it uses dynamic typing, which is different from C.

Output Generation:

Translated constructs concatenate to form the source Python code. The bulk of the translation process is about writing a script or a module in Python.

Supporting Syntax:

a) Variable and Literal:

Variables: When translating selection using both c and python languages, keep in mind equivalent data types available in each of them. One might say int "in" C will equal int "in" Python, float "in" C will be somewhere close to float "in" Python, and so on.

Literals: Literal values (constants) like integers, float point numbers, characters, and strings which are directly converted from C syntax to relevant Python syntax need to be incorporated.

b) Arithmetic and Logical Expressions:

Arithmetic: Translate arithmetic operators such as +, -, *, /, % in form of python from C. Take good care that the logic of operator precedence and associativity is not violated during translation process.

Logical: Convert logical operators like && (and), || (or), from natural language to semantic language. A manifold switch (not) from C to Python programming languages. Python has its own operators for the same operations. They are and, or, not.

c) Conditional and Loop Statements:

Conditional Statements (if-else): Translate the if-else statements written in C and verify that the formatting is OK in Python.

Loop Statements (for, while): Translate in and while loops from C language to Python. Handle explicitly the syntax of the loop conditions, the loop bodies, and the loop control statements (the break, the continue).

***d) One-Dimensional Array*:**

Replace C style array declarations and access expressions by Python equivalents. In Python, you can use lists to represent arrays wherever you need them. Call for correctly change array indices needed due to the fact that Python utilizes zero-based indexing.

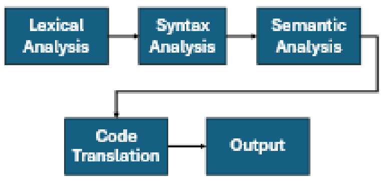


Fig-1 Flow Chart of code transformation.

The fig-1 shows how to determine the different stages of code interpretation: lexical recognizing, syntax parsing, semantic parsing, compile code, translation form human readable to machine readable.

Table – I
Token Table

Table I shows, In the given code the operations are returned as Tokens and Token are sent to next step to generate the python code.

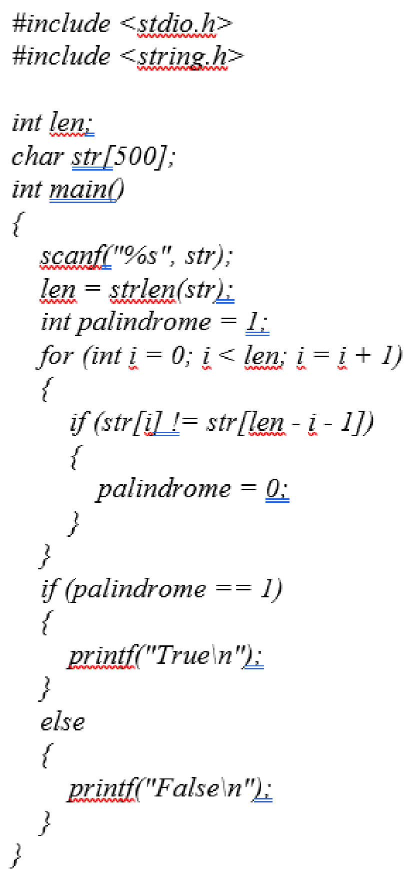
Operations	Return as
if	TIF
else	TELSE
for	TFOR
while	TWHILE
return	TRETURN
int	TINTTYPE
double	TDOUBLETYPE
char	TCHARTYPE
void	TVOIDTYPE
[a-zA-Z_][a-zA-Z0-9_]*	TIDENTIFIER
[0-9]+\.[0-9]*	TDOUBLE
[0-9]+	TINTEGER
['].[']	TCHAR
["].*["]	TSTRING
=	TEQUAL
==	TCEQ
!=	TCNE
<	TCLT
<=	TCLE
>	TCGT
>=	TGCE
&&	TAND
||	TOR
&	TAMPERSAND
(	TLPAREN
)	TRPAREN
{	TLBRACE
}	TRBRACE
[	TLBRACK
]	TRBRACK
.	TDOT
,	TCOMMA
;	TSEMICOLON
+	TPLUS
-	TMINUS
*	TMUL
/	TDIV
\n	Linenumber +=1
.	yyterminate()

IV. RESULTS

We tested out framework for various C codes. In this section we present three examples of different syntax of which each C code is coded, tested and generated the working output python code. The corresponding input and output are related and shown in the figures.

Example -1
a) Input.c

The code in C verifies whether a string is a palindrome by reading it, calculating its length, and iterating through half of the string. If a mish-match is found, the loop stops early, and the flag is set.



```
#include <stdio.h>
#include <string.h>

int len;
char str[500];
int main()
{
   scanf("%s", str);
   len = strlen(str);
   int palindrome = 1;
   for (int i = 0; i < len; i = i + 1)
   {
      if (str[i] != str[len - i - 1])
      {
         palindrome = 0;
      }
   }
   if (palindrome == 1)
   {
      printf("True\n");
   }
   else
   {
      printf("False\n");
   }
}
```

Fig-2 Palindrome C program

b) Out.py

The generated Python code uses a custom utility module, utils, to check if an input string is a palindrome. A corrected version uses built-in functions, comparing the string with its reverse and printing "True" or "False".

```
from utils import *

len = None
str = [None] * 500
def main():
        global len, str
        [str] = scanf("%s")
        len = strlen(str)
        palindrome = 1
        i = 0
        while (i < len):
                if (str[i] != str[((len - i) - 1)]):
                        palindrome = 0
                i = (i + 1)
        if (palindrome == 1):
                printf("True\n", ())
        else:
                printf("False\n", ())

if __name__ == '__main__':
        main()
```

Fig-3 Generated Output Palindrome Python file.

Example -2

a) Input.c

The C program creates the Fibonacci sequence up to the user's chosen number of terms, starting with the initialization of variables and the 'for' and 'if' loop, then updating the variables as needed.

```
#include <stdio.h>
int n;
int first;
int second;
int main()
{
  scanf("%d", &n);
  first = 0;
  second = 1;
  int next;

  for (int i = 0; i < n; i = i + 1)
  {
    if (i<=1)
    {
      next = i;
    }
    else
    {
      next = first + second;
      first = second;
      second = next;
    }
    printf("%d ", next);
  }
  printf("\n");
}
```

Fig-4 Fibonacci C program

b) Out.py

This generated Python script generates the Fibonacci sequence up to a user's specified number of terms, initializing variables and iterating through a 'while' loop, updating variables as needed.

```
from utils import *
n = None
first = None
second = None
def main():
        global n, first, second
        [n] = scanf("%d")
        first = 0
        second = 1
        next = None
        i = 0
        while (i < n):
                if (i <= 1):
                        next = i
                else:
                        next = (first + second)
                        first = second
                        second = next
                printf("%d ", (next))
                i = (i + 1)
        printf("\n", ())

if __name__ == '__main__':
        main()
```

Fig-5 Generated Output Fibonacci python file.

Example -3

a)Input.c

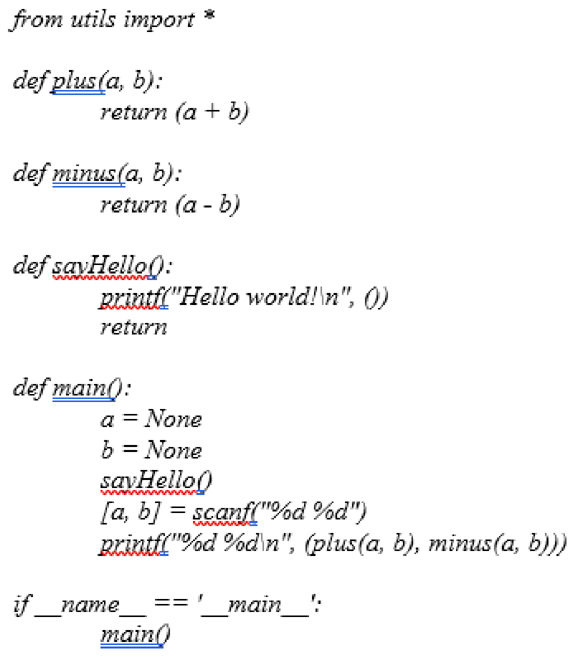
```
#include <stdio.h>

int plus(int a, int b)
{
  return a + b;
}
int minus(int a, int b)
{
  return a - b;
}
void sayHello()
{
  printf("Hello world!\n");
  return;
}
int main()
{
  int a;
  int b;
  sayHello();
  scanf("%d %d", &a, &b);
  printf("%d %d\n", plus(a, b), minus(a, b));
}
```

Fig-6 Functions C program

This C program uses plus and minus functions for addition and subtraction, and a void function sayHello for greeting messages. It reads user input, prints the results, and has no loops, including two int-returning functions.

b) Out.py



```
from utils import *

def plus(a, b):
        return (a + b)

def minus(a, b):
        return (a - b)

def sayHello():
        printf("Hello world!\n", ())
        return

def main():
        a = None
        b = None
        sayHello()
        [a, b] = scanf("%d %d")
        printf("%d %d\n", (plus(a, b), minus(a, b)))

if __name__ == '__main__':
        main()
```

Fig-7 Output generated python program.

.

The Python script makes use of the 'utils' module's functions for addition, subtraction and greeting messages. This way it scans user input, say Hello prints a message, and prints the resulting integers without any loops.

We tested our system for various constructs of C such as for loop, while loop, if-else loop, arrays to the code which was then tested to check if the translator successfully generated Python files as the output. These Python files were run without any errors, hence showing the good working of the translator in converting the C code to the Python code which runs Successfully.

V. Conclusion

From the observations it is seen that the C to Python translator successfully translates the C code into the Python code which is syntactically correct and functionally equivalent. The flawless operation of the Python code that was generated without any errors proves that the translation of the C code has not lost the logic and functionality of the original C. This indicates that the translator could be a useful source for the developers who want to migrate or work with C code in Python environments.

In the end, we will say that building this Python-to-C translator with Lex and Yacc was a a useful tool which involved creating an output, performing transformations, conducting the semantic analysis and the actual lex and syntax processing. As with the technologies and methods mentioned before, you will be able to build up a perfect translator that works both ways – from C to Python and the opposite.

Attention to detail is of crucial consequence all along this process, but we should not forget that it is more noticeable when managing the differences between the languages in terms of syntax and semantic meaning, providing accuracy and ready readability in the translated result. In addition, inspection and affixing of endorsement and practical instructions are needed in order to declare the interpreter fully functional and ready for use.

References

[1] Lachaux, Marie-Anne, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. "Unsupervised translation of programming languages." arXiv preprint arXiv:2006.03511 (2020).

[2] Shi, Freda, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. "Natural language to code translation with execution." arXiv preprint arXiv:2204.11454 (2022).

[3] Hariri, Walid. "Unlocking the potential of ChatGPT: A comprehensive exploration of its applications, advantages, limitations, and future directions in natural language processing." arXiv preprint arXiv:2304.02017 (2023).

[4] Lachaux, Marie-Anne, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. "DOBF: A deobfuscation pre-training objective for programming languages." Advances in Neural Information Processing Systems 34 (2021): 14967-14979.

[5] Andrabi, Syed Abdul Basit. "A review of machine translation for south asian low resource languages." Turkish Journal of

Computer and Mathematics Education (TURCOMAT) 12, no. 5 (2021): 1134-1147.

[6] Ahmad, Wasi Uddin, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. "Summarize and generate to back-translate: Unsupervised translation of programming languages." arXiv preprint arXiv:2205.11116 (2022)..

[7] Kusum, Kusum, Abrar Ahmed, C. Bhuvana, and V. Vivek. "Unsupervised translation of programming language-a survey paper." In 2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N), pp. 384-388. IEEE, 2022.

[8] Artetxe, Mikel, Gorka Labaka, and Eneko Agirre. "Unsupervised statistical machine translation." arXiv preprint arXiv:1809.01272 (2018).

[9] Roziere, Baptiste, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. "Leveraging automated unit tests for unsupervised code translation." arXiv preprint arXiv:2110.06773 (2021).

[10] Liu, Fang, Jia Li, and Li Zhang. "Syntax and domain aware model for unsupervised program translation." In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 755-767. IEEE, 2023.

[11] Artetxe, Mikel, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. "Unsupervised neural machine translation." arXiv preprint arXiv:1710.11041 (2017).

[12] Artetxe, Mikel, Gorka Labaka, and Eneko Agirre. "An effective approach to unsupervised machine translation." arXiv preprint arXiv:1902.01313 (2019).

[13] Li, Zuchao, Hai Zhao, Rui Wang, Masao Utiyama, and Eiichiro Sumita. "Reference language based unsupervised neural machine translation." arXiv preprint arXiv:2004.02127 (2020).

[14] Ahmad, Iftakhar, and Lannan Luo. "Unsupervised Binary Code Translation with Application to Code Similarity Detection and Vulnerability Discovery." arXiv preprint arXiv:2404.19025 (2024).

[15] Ahmad, Wasi Uddin, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. "Avatar: A parallel corpus for java-python program translation." arXiv preprint arXiv:2108.11590 (2021).

[16] Han, Jesse Michael, Igor Babuschkin, Harrison Edwards, Arvind Neelakantan, Tao Xu, Stanislas Polu, Alex Ray et al. "Unsupervised neural machine translation with generative language models only." arXiv preprint arXiv:2110.05448 (2021).

[17] Malyala, Aniketh, Katelyn Zhou, Baishakhi Ray, and Saikat Chakraborty. "On ML-based program translation: perils and promises." In 2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), pp. 60-65. IEEE, 2023.

[18] Loganathan, A., 2021. Investigating the Effect of Cluster-Based Preprocessing on Source-to-Source Code Translation (Doctoral dissertation).

[19] Zhu, Ming, Karthik Suresh, and Chandan K. Reddy. "Multilingual code snippets training for program translation." In Proceedings of the AAAI conference on artificial intelligence, vol. 36, no. 10, pp. 11783-11790. 2022.

[20] Szafraniec, Marc, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. "Code translation with compiler representations." arXiv preprint arXiv:2207.03578 (2022).

[21] Likhith, Arlagadda Naga, Kothuru Gurunadh, Vimal Chinthapalli, and Meena Belwal. "Compiler For Mathematical Operations Using English Like Sentences." In 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 1-6. IEEE, 2023.

[22] Pecheti, Shiva Teja, H. M. Basavadeepthi, Nithin Kodurupaka, and Meena Belwal. "Recursive Descent Parser for Abstract Syntax Tree Visualization of Mathematical Expressions." In 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 1-6. IEEE, 2023.

[23] Jaswanth, Kunisetty, et al. "Data Encryption: A Compiler Based Approach." 2023 7th International Conference on Computation System and Information

Technology for Sustainable Solutions (CSITSS). IEEE, 2023.

[24] Balan, Anirudh, and K. Deepa. "Detection and Analysis of Faults in Transformer using Machine Learning." In 2023 International Conference on Intelligent Data Communication Technologies and Internet of Things (IDCIoT), pp. 477-482. IEEE, 2023.