On–Device Translation of Dynamically Typed and Interpreted Languages

Pranav H, Muppavarapu Sriharshini, Rachuri Tarun, Meena Belwal,
Dept of Computer Science Engineering, Amrita school of computing, Amrita Viswa Vidyapeetham, Bengaluru, India.
pranav.03.h@gmail.com, harshini.bannu2004@gmail.com, tarunrachuri0303@gmail.com, b_meena@blr.amrita.edu

Abstract- On-Device translation of expression form languages with dynamic nature and using interpreted languages like MATLAB and Python requires a precise and automated form of processing the error-free conversion. With the help of the automation, the code in MATLAB is well-defined to Python so that translation is precise and the originality as well as functionality is ensured. This paper describes a robust and redundant methodology for converting maintaining the functionality and originality of the MATLAB code helping in smooth transition of code from MATLAB to python. The procedure being entirely local also ensures to protect privacy of the code.

Keywords- MATLAB, Python, Tokenizer, Token, Parser, Code Generator, Dynamical Type, Interpreting language, On–device Code Translation

I. INTRODUCTION

The field of programming is broad and dynamic with a vast array of programming languages accessible [1], each possessing distinct syntax and semantics, developers frequently have to collaborate across many languages [2]. This requirement may occur for a number of reasons, including the need to maintain old code, the need for a specific language for a given activity, or even just personal preference. But picking up and mastering several languages can be a difficult endeavor. This is where the idea of a converter and translator for programming languages is useful.

A tool called the Programming Language Translator and Converter [3,4] is intended to help create connections between various programming languages. Its objective is to transfer the logic and structure of the original source code written in one programming language is translated functionally to another programming language [5,6]. This tool can greatly increase productivity and efficiency by cutting down on the time and effort needed to rewrite code in a foreign language.

The goal of this project is to create a reliable and effective on-device converter and translator for programming languages. The major goal is to support translation of the popular language MATLAB to Python and with scope for additional extension. In order to guarantee appropriate translation, the project also attempts to manage the subtleties and complexities of each language, to ensure the translated code maintains it's originality and is functionally equivalent.

Hereafter, in Section 2 of the paper, the literature review surveys seven scholarly papers covering 10 different and modern methods of Language Translators. Section 3 of this Paper will be going through in detail of our proposed methodology.

II. LITERATURE SURVEY

Modern developments in summarization and source code translation are examined in this review of the literature. To ensure accurate code conversion, researchers have proposed unique and novel approaches that make use of various techniques explored further in this section.

Jana et. al [6] introduces CoTran, a technology that converts code while using feedback from compiler outputs together with symbolic execution for training massive language models. Through this method, the translated code is guaranteed to compile and have the same functionality as its source code thus being more effective than any other tool available since it is accurate in terms of compilation as well as functional equivalence.

Szafraniec et al. [7] suggests using lower-level compiler intermediate representations like LLVM IR to beef up to better neural machine translation results in JavaScript source codes. Such advancements aimed at eliminating common semantic unoriginality during unsupervised programming translation significantly improve unsupervised programming translation including multiple languages.

Brauckmann et al. [8] introduces ComPy-Learn as a flexible toolbox which is intended for the examination of divergent program code machine learning representations with the aim of optimizing compiler heuristics as well as other software engineering tasks. It allows conducting empirical research to uncover the best representations and models that combine higher-order syntax together with low-level compiler information.

Gourdin et al. [9] combines formally verified transformations integrated within CompCert, with a focus on Lazy Code Motion and Lazy Strength Reduction, to enhance performance optimizations provided by a compiler. The improvements result from introducing a Coq-verified validator that verifies correctness of these optimizations leading to their higher reliability.

Zugner et al. [10] introduces a new model that leverages both the context and structure of the source code, utilizing language-agnostic features derived from the abstract syntax tree and the code itself, achieving exemplary results in code summarization across five unique monolingual programming languages but also pioneers a multilingual code summarization model demonstrates significant improvements, especially in low-resource languages, underscoring the advantage of integrating structure and context in learning representations of code. Murali et al. [22] introduced a Utilizing visual problem-solving tools to empower novice programmers in learning and mastering coding concepts.

Exploring advanced techniques for source code summarization using abstract syntax trees (ASTs) and transformers, Choi et al. [12] integrates graph convolution with transformers to capture both sequential and structural code features, enhancing summarization accuracy, Hou et al.[11] developed a tree structure-based transformer model, TreeXFMR, which utilizes hierarchical attention and bi-level positional encodings improving the representation and summarization of provided source code, showing significant performance improvements over existing methods. Varshini [23] introduced a syntactical parsing technique using the CYK algorithm to parse basic Telugu sentences, evaluating its performance in terms of accuracy, precision, and recall. Pecheti et al. [13] introduces a method for parsing and visualizing, expressions using Abstract Syntax Trees to enhance accuracy and adaptability.

Lachaux et al. [14] introduced an unsupervised neural translator that can translate among C++, Java as well as Python. It was trained by using monolingual source codes from GitHub. Liu [15] presents SDA-Trans, A model that is both Syntax and Domain Aware for unsupervised program translation, performing well on especially translation of functions between Python, Java, C++, especially with not specifically, using a smaller-scale corpus. Nguyen [16] proposes a novel refinement procedure for unsupervised machine translation models to focus on low-resource languages by disentangling them from high-resource languages in a multilingual environment, achieving state-of-the-art results in translating between multiple low-resource languages. Huang [17] developed Code Distillation (CoDist), model that uses a language-agnostic intermediate representation to overcome the lack of parallel corpora in program translation, improving performance on several program translation benchmarks.

Advancements in multi-language and multi-target compilers, focusing on efficiency and domain-specific needs have evolved as a field of considerable research in the recent years wherein Nandhini [18] discussed an online multi-language compiler system designed to streamline coding processes in educational settings, Boukham [19] explored a domain-specific language compiler for graph processing, emphasizing adaptable intermediate representations for various computing paradigms and Mullin [20] addresses the compilation of data parallel languages into an intermediate language, aiming for effective use across different multiprocessor topologies. Together, these studies underscore the evolving complexity and adaptability required in modern compiler design. Likhith et al. [21] developed a tool that interprets and executes mathematical operations expressed in English-like sentences using Lex and YACC.

III. PROPOSED METHODOLOGY

To convert MATLAB code to Python, there is a need to utilize a series of automated processes to ensure precise and efficient translation.

To ensure the translation of MATLAB code to Python code is smooth and does not affect the originality and functionality of the code, there is a need to use a direct and dynamic approach.This is accompanished with the help of the following modules :

1. Tokeniser
2. Lexer
3. Parser
4. Code Generator

Tokeniser

At first, the tokeniser is used to split the source (. m) files into tokens. This encompasses the static methods that are used for getting the list of token types and the mapping of reserved words to their corresponding types. The TokenType is the tool that classifies tokens by their function in the language, for instance, the keywords, the operators, or the punctuation. Through this tokeniser, source code can be divided into the meaningful units for the better interpretation and Translation.

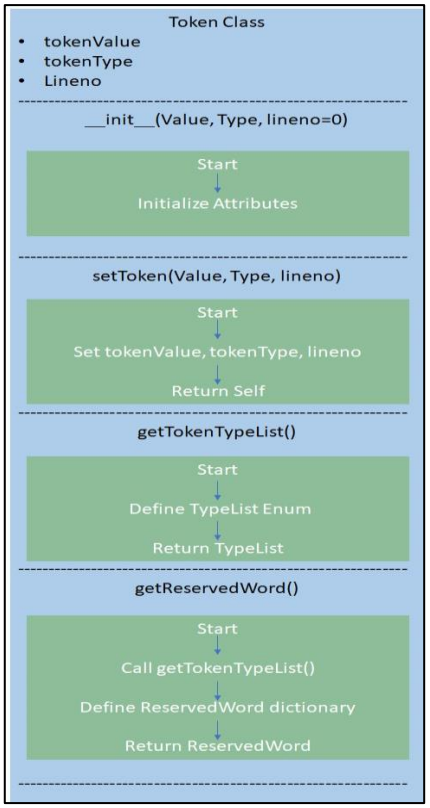

Fig.1.Creating a Token in Python

Lexer

The Lexical Analyser uses a state machine mechanism to sequentially process the characters. Every character met causes a switch to a certain state, like identification of numbers, identifiers, or operators. Such cases as comments, string literals, and logical operators are treated exactly. Tokens are divided into characters, which are then parsed and the reserved words are assigned the appropriate token types. This ordered approach guarantees the Lexical Analysis of the Source Code to be correct and thus the next steps of parsering will also be successful.

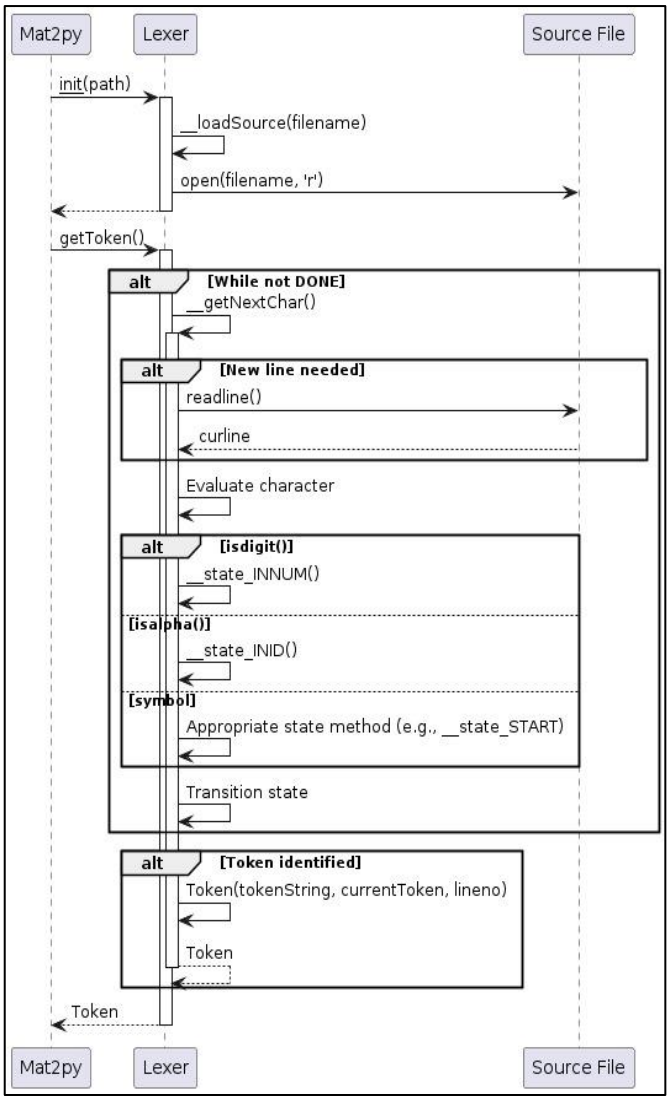

Fig.2. Sequence Diagram: Tokenization Process in Python Lexer Class

This diagram shows the process of how the Lexer class reads the source file character by character, finds tokens according to the rules of the programming language, and then creates Token objects that represent those tokens.

Parser

The Recursive decent Parser designed is a collection of methods that are specialized for different language elements like statements, expressions, loops, function declarations and the rest. It is the interaction between the lexer and the tokenizer that lets it to distinguish between the different types of nodes in the abstract syntax tree (AST) using the enumeration mechanism which, in turn, makes it possible for it to represent the hierarchical structure of the source code. It can also cope with the complicated construct as if-else blocks, for loops, while loops, function declarations, and expressions that contain operators, identifiers, literals, and function calls. Additionally, it can create a function table to note the functions that have been declared.

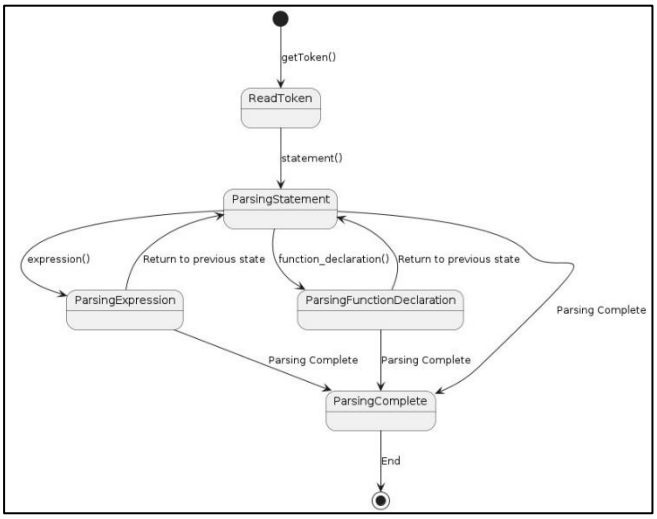

Fig.3. Sequence Diagram: Parsing a Python Program

This diagram shows the sequence of steps that the Lexer class follows when parsing a Python program. The lexer interacts with the Mat2py class and the source file to find tokens.

Code Generator

The Code generator goes on to output the Python code for the equivalent source matlab file, beginning with the necessary imports and headers. It is the procedure that defines the methods for dealing with various kinds of expressions and statements, recursively walking through the AST. When coming across expression nodes, it converts MATLAB operators and constructs into their Python counterparts which also covers function calls and special cases like ranges and assignments. It guarantees the indentation and the structure of the Python code that is produced. Moreover, it simultaneously loads library functions as required. The generator works systematically to transform MATLAB syntax and constructs into equivalent Python code, while preserving the formatting and structure of the code.

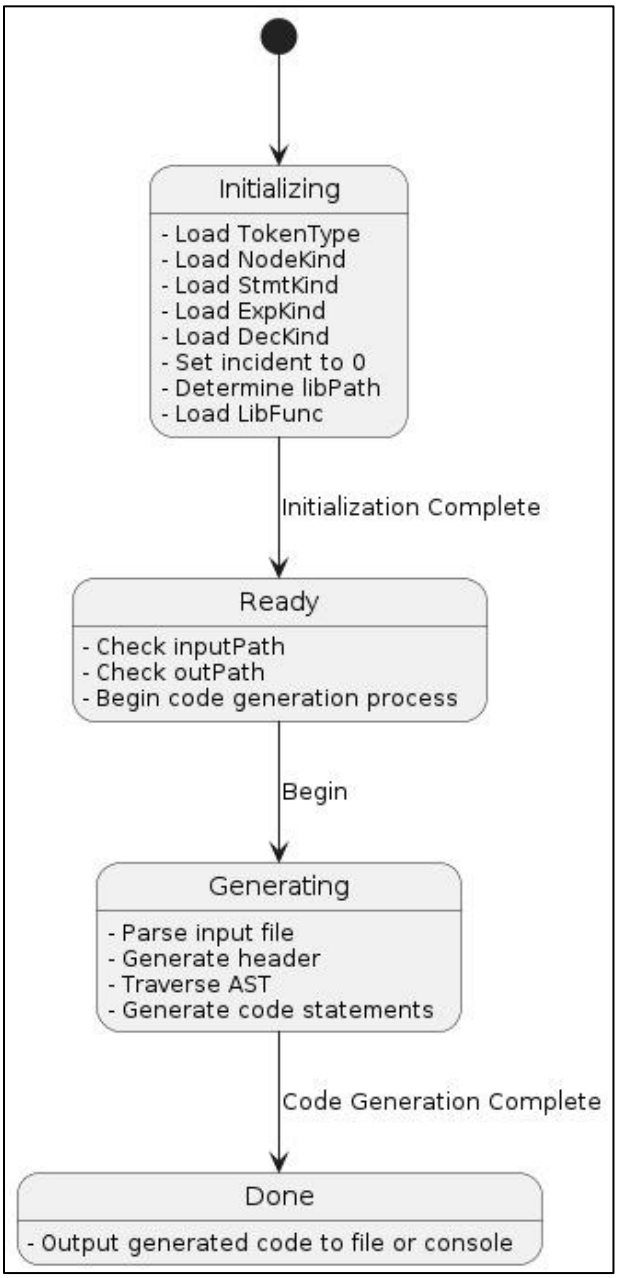

Fig.4. Code Generation Process

This flowchart shows the sequence of activities between a source file that is written in a programming language and executable code. The initialization phase is the phase in which the code generator sets up the required conditions, such as importing the necessary libraries or frameworks. After initialization, the generator verifies the source file and output location to ensure that the source file exists and is writable to the output location. Thus, the source file is analyzed and thus the structure and semantics of the file is understood, usually with the creation of an Abstract Syntax Tree (AST) that represents the syntax of the program. After the AST, the code generator closely follows the node, elements such as functions, variables and expressions are found. The process of this path is that the generator transforms each element into its programming language equivalent in the target language using syntax and logic unique to that language. Thus, the generated code is either inserted into the output file or displayed in the console, depending on the tool's capabilities.

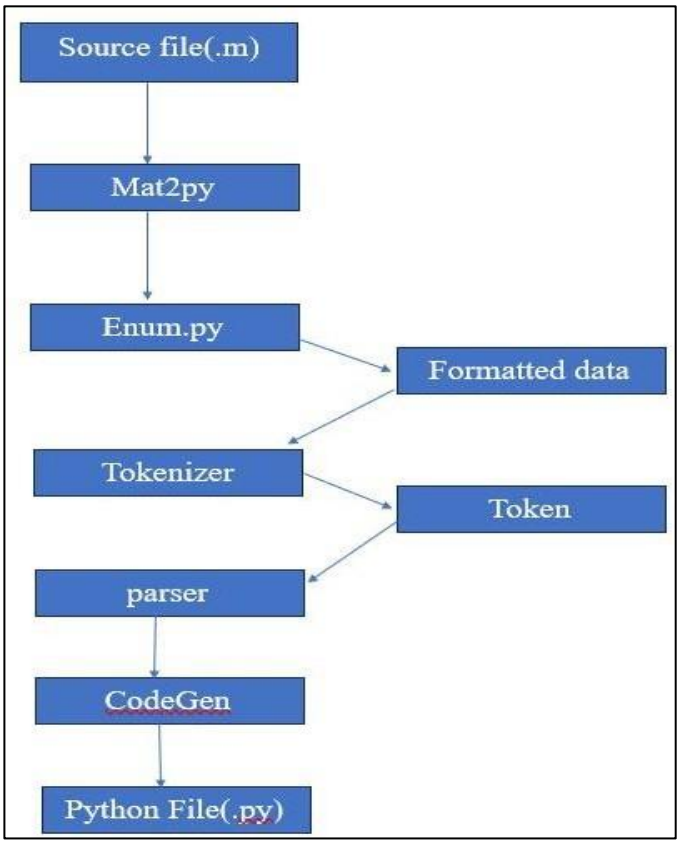

Fig.5.Flowchart outlining the conversion of a MATLAB file (.m) into a Python file (.py)

The flowchart depicts a program that facilitates the conversion of MATLAB code into Python code. Here's a simplified summary:

1. The MATLAB code is fed into the program.
2. Mat2py prepares the code for translation.
3. CodeGen.py, potentially aided by helper scripts, translates the code into Python.
4. The final Python code is generated as a new ".py" file.

IV. RESULTS

The process of interpretation of MATLAB to Python operation begins with the involvement of Mat2py, a tool that is designed to support the translation of MATLAB codes into a compatible data format provided by Python form. After that, the Tokenize module finds these and divides the formatted data into tokens, containing words, symbols, and punctuation marks, which represent the lexical components of the code. Putting this idea into practice, we use Enum module that helps to create values in updates. Next a Parser module will get tokens as input. This module analyzes tokens and construct a structured representation (usually in the form of intuitive syntax tree) of the MATLAB code. Unlike unstructured codes that solely target the immediate problem, a predefined prototype is a smart and resourceful way to develop the subsequent stage of code generation. The turn over to CodeGen is now the responsibility of generating the Python code from the parsed MATLAB code such that the operations and semantics in the MATLAB language are converted into their Python equivalents to harmonize functionality between the two languages.

Table.1.For the given for .m file a for.py file is generated

For.m	For.py

function result = calculateFactorial(n)
result = 1;
for i = 1:n
result = result * i;
end
end
	#!/usr/bin/python
	# -*- coding: utf-8 -*-
	import numpy as np
	import copy
	import os
	import sys
	
	def calculateFactorial(n):
	
	result = 1
	for i = range(0,n):
	result = np.dot(result,i)
	
	return result

Table.2. For the given multiply.m file a multiply.py file is generated

Multiply.m	Multiply.py
% Define the dimensions of the rectangle length = 5; % Length of the rectangle width = 3; % Width of the rectangle % Calculate the area of the rectangle area = length * width;	#!/usr/bin/python # -*- coding: utf-8 -*- import numpy as np import copy import os import sys length = 5 width = 3 area = np.dot(length,width)

Table.3. For the given If-Else.m file a If-Else.py file is generated

If-Else.m	If-Else.py

function result = checkNumber(num)
if num > 0
result = 'The number is positive.';
elseif num < 0
result = 'The number is negative.';
else
result = 'The number is zero.';
end
end
	#!/usr/bin/python
	# -*- coding: utf-8 -*-
	import numpy as np
	import copy
	import os
	import sys
	
	def checkNumber(num):
	
	if num > 0:
	result = 'Thenumberispositive.'
	elif num < 0:
	result = 'Thenumberisnegative.'
	else:
	result = 'Thenumberiszero.'
	
	return result

As a result, the end result of the conversion process is a Python source code file (.py), which compiles the converted code into one file to be run or improved. This python file shows the functionality and the logic of the original MATLAB code which is making the porting and interoperability between the two languages fast and efficient.

V. CONCLUSION

Translation between MATLAB and Python, the two dynamic and interpreted languages, is a two-fold challenge; involving balancing between complexity and efficiency in terms of the techniques and tools in place. The function of the tool developed is to establish a systematic procedure using the Tokenizer, Lexer, Parser and Code Generator working collaboratively and translating MATLAB code into Python code whose meaning, originality and functionality is preserved. A point is made that an automated procedure helps the code's logic to remain unchanged and to maintain the same behavior as the original, so communication between MATLAB and Python domains is resolved without obstacles. The ability to translate these codes on-device efferently also ensures the privacy of the code. By the use of the proposed procedure, it is possible to do the research, practice and harness the capabilities of most of both languages, MATLAB and Python.

VI. REFERENCES

[1] Sharma, Mamillapaly Raghavender. “A Short Communication on Computer Programming Languages in Modern Era.” *International Journal of Computer Science and Mobile Computing* (2020): n. pag.

[2] Sahakyan, Davit and Gurgen Karapetyan. “The Language of Translation.” *Translation Studies: Theory and Practice* (2022): n. pag.

[3] Tonchev, Ognyan and Mohammed Elhafiz Ramadan Salih. “High-level programming languages translator.”

[4] Raj, Utkarsh, Navneet Kaur and Babita Rawat. “English Algorithm Translation to C Program using Syntax Directed Translation Schema.” *2022 International Conference on Cyber Resilience (ICCR)* (2022): 1-3.

[5] Mariano, Benjamin, Yanju Chen, Yu Feng, Greg Durrett and Işıl Dillig. “Automated transpilation of imperative to functional code using neural-guided program synthesis.” *Proceedings of the ACM on Programming Languages* 6 (2022): 1 - 27.

[6] Jana, Prithwish, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan and Vijay Ganesh. “CoTran: An LLM-based Code Translator using Reinforcement Learning with Feedback from Compiler and Symbolic Execution.” (2023).

[7] Szafraniec, Marc, Baptiste Rozière, Hugh Leather Francois Charton, Patrick Labatut and Gabriel Synnaeve. “Code Translation with Compiler Representations.” *ArXiv* abs/2207.03578 (2022): n. pag.

[8] Brauckmann, Alexander, Andrés Goens and Jerónimo Castrillón. “ComPy-Learn: A toolbox for exploring machine learning representations for compilers.” *2020 Forum for Specification and Design Languages (FDL)* (2020): 1-4.

[9] Gourdin, Léo. “Lazy Code Transformations in a Formally Verified Compiler.” *Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems* (2023): n. pag.

[10] Zugner, Daniel, Tobias Kirschstein, Michele Catasta, Jure Leskovec and Stephan Gunnemann.“LanguageAgnostic Representation Learning of Source Code from Structure and Context.” ArXiv abs/2103.11318 (2021): n. pag.

[11] Hou, Shifu, Lingwei Chen and Yanfang Ye. “Summarizing Source Code from Structure and Context.” 2022 International Joint Conference on Neural Networks (IJCNN) (2022): 1-8.

[12] Choi, YunSeok, Jinyeong Bak, CheolWon Na and Jee-Hyong Lee. “Learning Sequential and Structural Information for Source Code Summarization.” Findings (2021).

[13] 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.

[14] Lachaux, Marie-Anne, Baptiste Rozière, Lowik Chanussot and Guillaume Lample. “Unsupervised Translation of Programming Languages.” ArXiv abs/2006.03511 (2020): n. pag.

[15] Liu, Fang, Jia Li and Li Zhang. “Syntax and Domain Aware Model for Unsupervised Program Translation.” 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (2023): 755-767.

[16] Nguyen, Xuan-Phi, Shafiq R. Joty, Wu Kui and Ai Ti Aw. “Refining Low-Resource Unsupervised Translation by Language Disentanglement of Multilingual Model.” ArXiv abs/2205.15544 (2022): n. pag.

[17] Huang, Yufan, Mengnan Qi, Yongqiang Yao, Maoquan Wang, Bin Gu, Colin B. Clement and Neel Sundaresan. “Program Translation via Code Distillation.” Conference on Empirical Methods in Natural Language Processing (2023).

[18] Nandhini, N., M. G. M. Prassanna, D. Tamilarasi, R. Varthini and S. Sadesh. “Implementation Of Multi Language Compiler for College Using Smart Lab System.” (2019).

[19] Boukham, Houda, Guido Wachsmuth, Martijn Dwars and Dalila Chiadmi. “A Multi-target, Multi-paradigm DSL Compiler for Algorithmic Graph Processing.” *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (2022): n. pag.

[20] Mullin, Lenore M. Restifo, C. Chang, S. Huang, Mikaël Mayer, Nader A. Nemer and C. Ramakrishna. “Intermediate Code Generation for Portable Scalable, Compilers. Intermediate Code Generation for Portable Scalable, Compilers. Architecture Independent Data Parallelism: The Preliminaries Architecture Independent Data Parallelism: The Preliminaries.” (2020).

[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] Murali, Ritwik, Rajkumar Sukumar, Mary Sanjana Gali, and Veeramanohar Avudaiappan. "Empowering Novice Programmers with Visual Problem Solving tools." In Proceedings of the 16th Annual ACM India Compute Conference, pp. 100-103. 2023.

[23] Varshini, Surisetty Hima, Gottimukkala Sarayu Varma, and M. Supriya. "A recognizer and parser for basic sentences in telugu using cyk algorithm." 2023 3rd International Conference on Intelligent Technologies (CONIT). IEEE, 2023.