6CCS3PRJ Final Year
ML in Software Engineering- Legacy Programming Language Translation using ML approach

Final Project Report

Author: Ting-Chen Chen in BSc Computer Science with Management

Supervisor: Kevin Lano

Student ID: 20076342

April 12, 2024

Abstract

This report presents a comprehensive exploration of the innovative integration of Machine Learning (ML) techniques in the field of software engineering, with a specific focus on the translation of legacy programming languages. The burgeoning demand to maintain and update legacy systems, many of which are written in outdated programming languages, has created significant challenges in the software engineering industry. This study aims to address these challenges by developing a machine learning-based approach for translating legacy code into modern programming languages.

The report delves into the adaptation of Machine Learning models to understand and translate legacy code, highlighting the novel use of neural networks, particularly convolutional and tree-based neural networks, for understanding programming language structures.

Significant findings of this research include the successful demonstration of ML models' ability to accurately translate legacy code with high efficiency and minimal human intervention. The report presents a comparative analysis of different ML models, evaluating their effectiveness in various scenarios of legacy code translation. This project developed perfectly aligned parallel standalone functions in both Pascal and Java for training, validation, and testing purposes. This project employs a corpus to develop a Pascal to Java Tree2Tree translation model, evaluating its effectiveness using chosen metrics in comparison to a commercial standard. The final result were all calculated in mathematically and statistically.

The implications of this study are profound. By automating the translation of legacy code, organizations can save considerable time and resources while mitigating the risks associated with manual code translation. Additionally, this approach fosters a smoother transition to modern programming environments, ensuring the longevity and sustainability of valuable legacy systems in the rapidly evolving technological landscape.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Ting-Chen Chen in BSc Computer Science with Management

April 12, 2024

Acknowledgements

I would like to thank my project supervisor Dr. Kevin Lano for his support during every meeting and provided lots of online resources and papers for me throughout this project. I would also like to appreciate to all of the previous works' and papers' authors. Without the previous research papers, this project wouldn't be completed. At the end, I would like to thank to my family and friends for all of the love and support throughout my whole life. Especially, thanks to my parents.

Contents

Chapter 1

Introduction

This project focuses on the innovative application of Machine Learning (ML) in the realm of software engineering, with a specialized emphasis on translating legacy programming languages using ML methodologies. The objective is to address a critical gap in current software maintenance and development practices: the efficient translation and modernization of legacy code, which is often written in outdated programming languages, into contemporary programming languages.

1.1 Motivation

The motivation for this project is deeply rooted in the two objectives of technological advancement and educational enrichment. While the primary aim is to update and sustain legacy systems by transcoding from Pascal to Java—thereby tapping into the enhanced capabilities of modern programming languages such as improved performance, security, and maintainability-the project also presents a unique opportunity for student learning and development. Embarcadero company is the company behind Delphi, which is the object pascal IDE. By reading through their company website, there are many case studies and successful stories from companies that utilise Delphi for company's main software development. [14] Recognising Pascal as a legacy programming language, its continued use in education is attributed to its simple syntax and ease of learning, making it an ideal starting point for programming students.[49] Integrating the motivation for student learning into the broader context of updating and sustaining legacy systems through transcoding from Pascal to Java adds an important educational dimension to the project. This approach not only addresses the technical aspects of modernization but also

leverages the process as a valuable learning opportunity for students.

1.2 Definition

- Legacy Systems: Older computer systems or applications that continue to be used, despite the availability of more modern technologies. [48]
- Transcoder: A tool or system that converts code from one programming language to another. [25]
- Neural Machine Translation: An approach using neural network models for improving translation accuracy and efficiency.[34]

1.3 Aims and Objectives

This project mainly aims to analyse the design and utilisation of program translation model, tree-to-tree model, by translating from Pascal to Java, which is an unexplored translation pair. The aim of this project is to modernize legacy software systems, enhancing their maintainability and scalability while reducing maintenance costs associated with outdated technologies. This involves creating a transcoder capable of efficiently translating legacy code into modern programming languages.

The objectives include developing potential usable models for the transcoder, exploring advanced approaches such as Tree-to-Sequence Neural Machine Translation Models, Encoder Neural Machine Translation based on Gumbel Tree-LSTM, Tree-Based Convolutional Neural Networks (TBCNN), and Tree2Tree Neural Network approaches. These models are critical in ensuring accurate and efficient code translation. Additionally, the project aims to create a user-friendly web interface, making the transcoder accessible and easy to use.

1.4 Achievements of the Project

1.4.1 Datasets Creation

As one of the impact part of my project, I successfully created 18 Pascal-Java Pair datasets for tree-to-tree model training. The datasets can be all shown in the github repostitory: .

1.4.2 AST Grammar Analysis

Another minor accomplishment within my project has been the thorough analysis and comparison of basic programs in Pascal and Java, focusing specifically on their AST (Abstract Syntax Tree) grammar. This examination is a fundamental component of my primary effort to effectively transcoding Pascal into Java. Additionally, the detailed tables and findings from this analysis will be included in the appendix section of the document.

Chapter 2

Background

This chapter aims to equip readers with foundational knowledge on machine learning modelsin program translation. It begins by exploring neural networks, inspired by human brain functionality, and introduces the first neural network model by McCulloh and Pitts. The chapter then gets into machine translation, comparing various models including Rule-Based, Statistical Machine Translation, and innovative approaches like Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) models. It highlights the evolution from rule-based systems to more dynamic models capable of handling complex translations, such as tree-to-tree neural networks, chosen for their ability to manage structured data effectively. This selection is justified by examining their architecture, which mirrors the encoding and decoding processes inherent in language translation, and their superiority in understanding and reorganizing data structures.

2.1 Neural Networks

The concept of neural networks was motivated by the functionality of the human brain. Moreover,the first neural network was found by McCulloh and Pitts in the use of biological neuron. [37]

Below is the mathematical formula of the first neural network:

$$\mathbb{R}_+ \left(\sum_{i=1}^{d} w_i x_i - \theta \right),$$

where the domain dimension $d \in \mathbb{N}$ represents the number of input signals to the neuron. The activation function $\mathbb{R}_+ : \mathbb{R} \to \mathbb{R}$ is defined such that $\mathbb{R}_+(x) = 0$ for $x < 0$ and transitions to

$\mathbb{R}_+(x) = x$ otherwise. Within this framework, w_i are the weights assigned to each input signal x_i, and θ signifies the neuron's threshold.

Currently, the inspiration behind neural network theory is largely derived from the application of deep neural networks in a method known as **deep learning.** [19] Within deep learning, the focus is on algorithmically determining the most appropriate deep neural network for a given task. Consequently, it makes sense to explore strictly mathematical reasons for the effectiveness and suitability of a Multilayer Perceptron (MLP) architecture in practical scenarios, rather than basing the rationale solely on the successful performance of biological neural networks. (Figure 2.1)

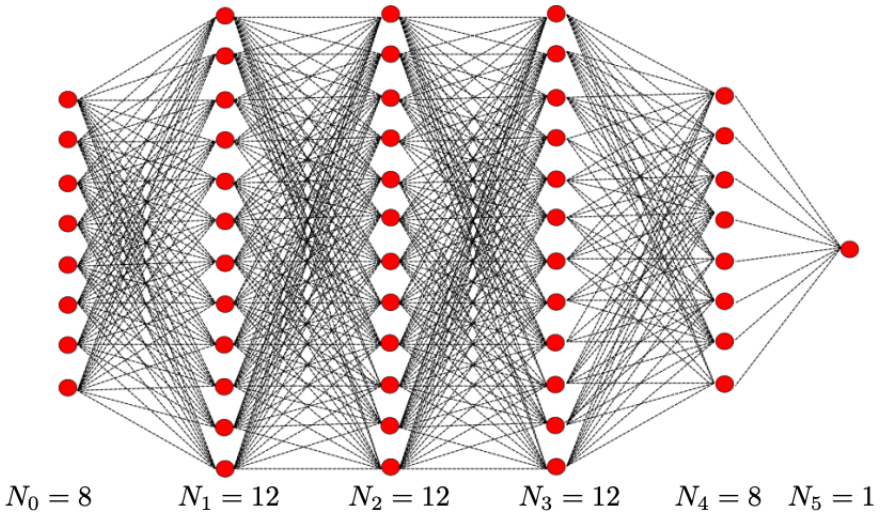


Figure 2.1: Multilayer Perceptron Architecture with Five Layers, Represented by Neurons as Red Dots[37]

2.2 Machine Translation

In the context of machine translation, the task involves taking a sequence of symbols, or text, in one language and having a computer program transform it into a corresponding sequence of symbols in a different language. This process is frequently used with natural languages, for example, converting text from English into French.[19]

2.2.1 Ruled-Based Machine Translation (RMBT)

Rules-Based Machine Translation (RBMT) systems, the pioneers of commercial machine translation, operate on linguistic rules to manipulate words based on context, undergoing analysis, transfer, and generation phases. Developed through the meticulous efforts of linguists and

programmers, RBMT uses manually created lexicons that users can refine, offering control over translation output. Despite its predictability, RBMT can be labor-intensive to refine and might produce translations that, while understandable, lack fluency and require significant post-editing for specific audiences. To overcome these drawbacks, recent RBMT developments have incorporated techniques from **Statistical Machine Translation (SMT)**, evolving into Hybrid MT models, which aim to balance RBMT's rule-based precision with SMT's adaptability.[33]

2.2.2 Statistical Machine Translation (SMT)

SMT is one of the type of machine translation which uses statistical approach and statistical models to translate text from one language to another. SMT is also a corpus-based machine translation method which the idea came out by International Business Machine (IBM) Corporation during 1990s. During the training phase in SMT, it learns from a large dataset of sentences that are translations of each other in two languages. It employs two main models: a translation model to predict how words and phrases translate from the source to the target language, and a language model to ensure the output is fluent and grammatically correct in the target language. During translation, SMT selects the most probable translation based on these models. Despite its effectiveness, SMT can struggle with nuances and context due to its reliance on direct statistical correlations, leading to the rise of **Neural Machine Translation** (NMT) for more context-aware translations.[38]

Word-based SMT and Phrase-based SMT

As the name of the machine translation, it's a simple unit word translation. As a result, words in an input sentence are translated individually, phrase by phrase, and are then subsequently arranged in a specific manner to form the target sentence. [4] Researchers relied on word-based SMT model in early approach. Yet, word-based SMT failed frequently to capture word-to-word translation. Therefore, lots of researchers started to use phrase-based SMT model instead of word-based SMT model. Phrase-based SMT model has more advantages compared to word-based SMT model. This approach inherently understands the context of words and the local rearrangement within phrases, providing a more nuanced translation.[32] As the result, phrase-based SMT is the most popular model in SMT approach.[39] A comparison of word-based SMT and phrase-based SMT diagram is shown in figure 2.2.

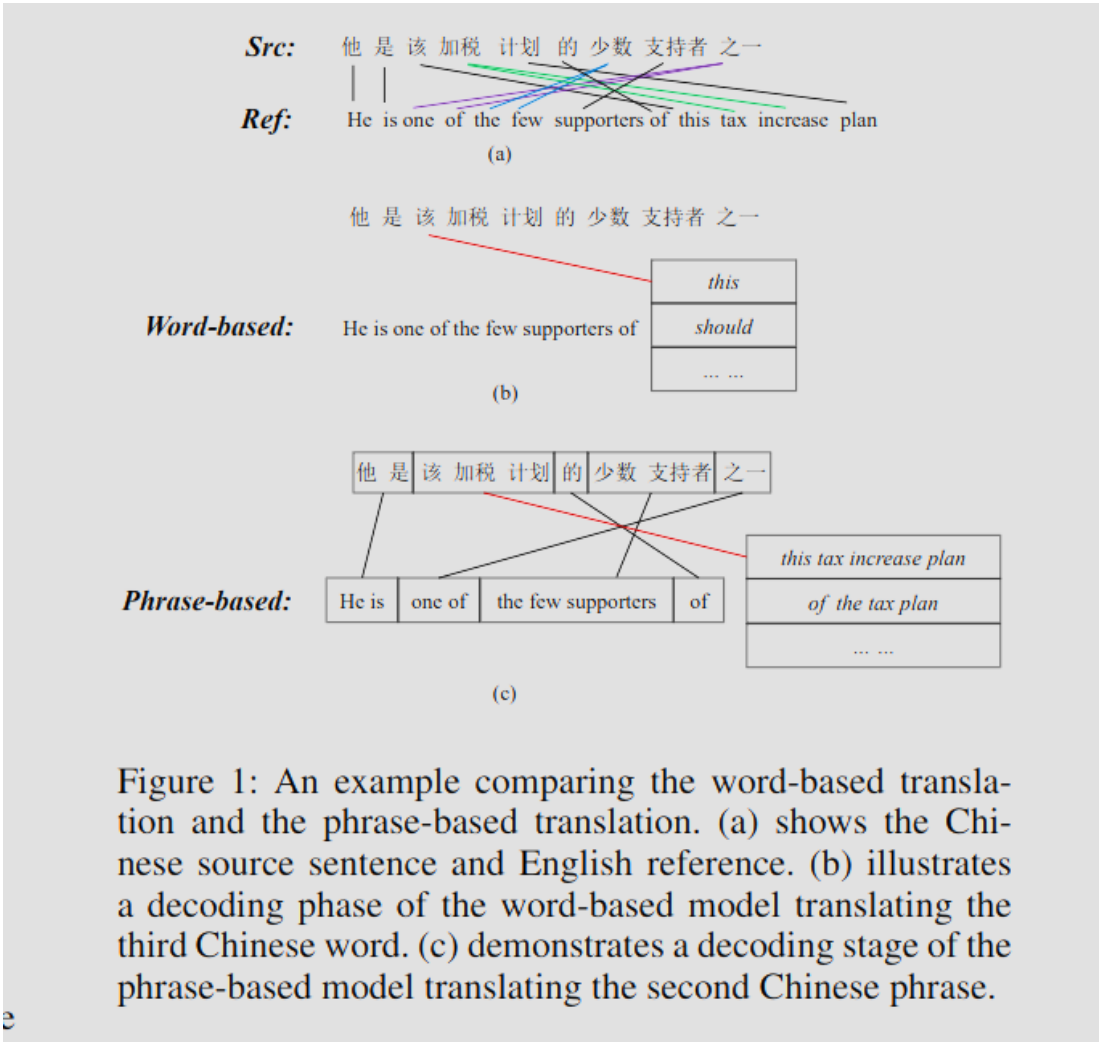


Figure 1: An example comparing the word-based translation and the phrase-based translation. (a) shows the Chinese source sentence and English reference. (b) illustrates a decoding phase of the word-based model translating the third Chinese word. (c) demonstrates a decoding stage of the phrase-based model translating the second Chinese phrase.

Figure 2.2: Comparison of Word-Based SMT and Phrase-Based SMT in Chinese and English Language[53]

2.2.3 Example-Based Machine Translation (EMBT)

Example-Based Machine Translation (EBMT) works by having a database of pre-translated sentences or phrases to construct new translations. When faced with a sentence to translate, EBMT searches this database for similar examples. It then adapts these found segments to match the context of the new sentence, possibly combining parts from multiple examples to produce a coherent translation. This method excels in handling nuanced and idiomatic expressions by directly utilizing real examples of language use, though it requires a comprehensive and well-maintained database of translations to be effective. EBMT's performance is particularly strong in specialized domains where there are abundant examples of previous translations to draw upon.[40]

2.2.4 Neural Machine Translation(NMT)

Neural Machine Translation (NMT) heralds a transformative approach in machine translation, employing deep neural networks to predict word sequences with remarkable accuracy. Surpassing traditional Statistical Machine Translation (SMT) methods within just three years, NMT has become the leading technology in the field, known for producing translations of superior quality, fluency, and adequacy. Unlike its predecessors, NMT trains a single, extensive neural network end-to-end, optimizing translation performance while minimizing memory usage. However, recognizing the occasional superiority of SMT in specific contexts, some, like Omniscien, have adopted Hybrid Machine Translation strategies, merging the best of NMT and SMT to achieve even higher quality translations.[34]

2.3 Forward Propagation and Back Propagation

Forward propagation in neural networks is the process where input data travels from the input layer, through the hidden layers where it is processed by **activation functions**, and ultimately reaches the output layer to produce a result. This progression ensures data flows linearly and prevents any circular movement that could hinder output generation. Each node in the hidden and output layers undergoes pre-activation — calculating a weighted sum — and activation, where this sum is adjusted by an activation function and bias to introduce non-linearity into the network's processing.

On the other hand, **backpropagation** is the method used to refine the neural network's accuracy, operating in reverse from the output layer back to the input layer. It involves recal-

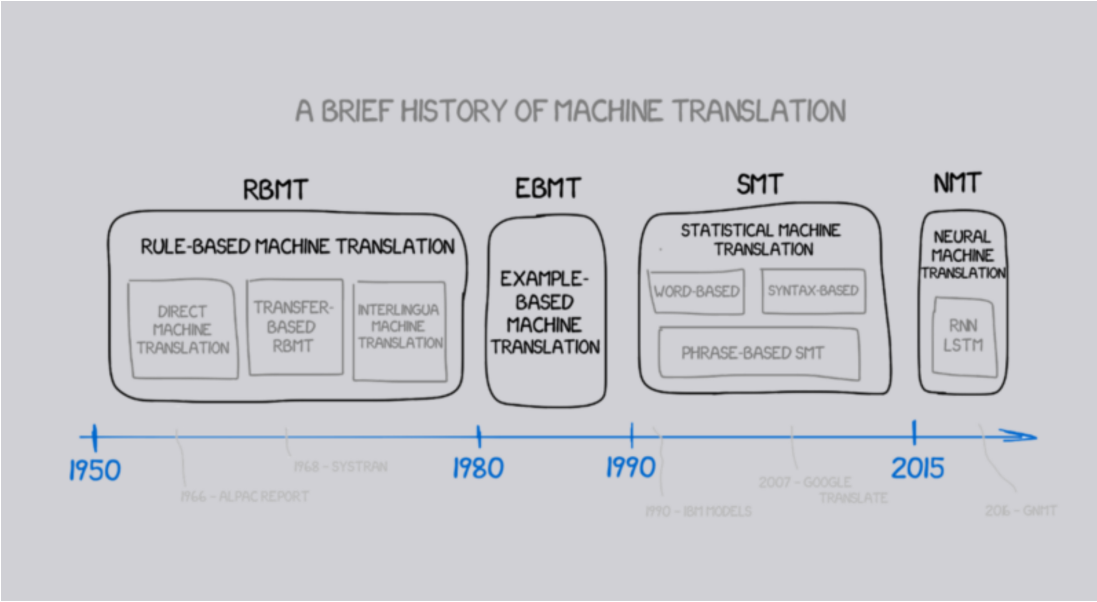


Figure 2.3: Machine Translation History[16]

culating and adjusting the network's weights based on the gradient of the loss function, which measures the network's prediction accuracy. Together, forward and backpropagation form the core mechanisms by which neural networks are trained, enabling them to learn from data and improve predictions over time. These processes allow the neural network to function as an interconnected system of input and output nodes, optimizing itself to reduce error rates and enhance performance.

2.4 Recurrent Neural Network (RNN)

Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed for processing sequential or time-series data, making them ideal for applications like language translation, natural language processing, speech recognition, and image captioning. Unlike traditional neural networks, RNNs possess a form of "memory" that allows them to use information from previous inputs to inform current and future outputs, effectively capturing dependencies within the input sequence. This characteristic enables RNNs to handle tasks where the order of elements is crucial, such as understanding the context in idioms or predicting the next word in a sentence.

RNNs are unique not only in their ability to process sequences but also in their architecture. They share parameters across different parts of the model, a contrast to feedforward networks where each node has distinct weights. This parameter sharing helps in learning sequences

efficiently but also introduces challenges during training, particularly through the backpropagation through time (BPTT) algorithm. BPTT adapts the traditional backpropagation process to sequence data, updating model parameters by aggregating errors across each timestep.

However, RNNs face significant challenges, such as exploding and vanishing gradients, which affect the network's ability to learn. These issues occur due to the size of the gradients during backpropagation, either becoming too large (exploding) or too small (vanishing), making the model unstable or causing it to stop learning. Addressing these problems often involves simplifying the network's architecture by reducing the number of hidden layers, thus mitigating complexity and stabilizing the learning process. RNNs' unique properties and their solutions to these challenges underline their significance in advancing applications that require understanding and generating sequential data.[23]

2.5 Long Short-Term Memory(LSTM)

Long Short-Term Memory (LSTM) is a type of **RNN architecture** designed to enhance the model's retention of information over extended time series. Unlike standard RNNs, which are limited to retaining short-term dependencies, LSTMs are proficient in capturing long-term relationships within the data. This is crucial as RNNs often struggle with the vanishing gradient issue when processing lengthy sequences, a problem LSTMs are structured to avoid. The key to their performance lies in their ability to selectively retain or discard information through an intricate system of gates. Specifically, LSTMs incorporate an input gate to manage the entry of new data into the cell state, a forget gate to eliminate non-essential information, and an output gate to determine the information to be carried forward to the next hidden state. This gated mechanism empowers LSTMs to maintain or erase memories in their cell state, thus enabling more effective learning from long-duration time series data.[13](Figure 2.4 and 2.5)

2.6 Encoder and Decoder Architecture

The encoder and decoder has two stages. First, an encoder stage that produces a vector representation of the input sentence. Then this encoder stage is followed by a decoder stage that creates the sequence output. The internal mechanism can be a recurrent neural network, as shown in figure below. A RNN encoder takes each token in the input sequence one at a time, and produces a state representing this token as well as all the previously ingested tokens. Then the state is used in the next encoding step as input along with the next token to produce the

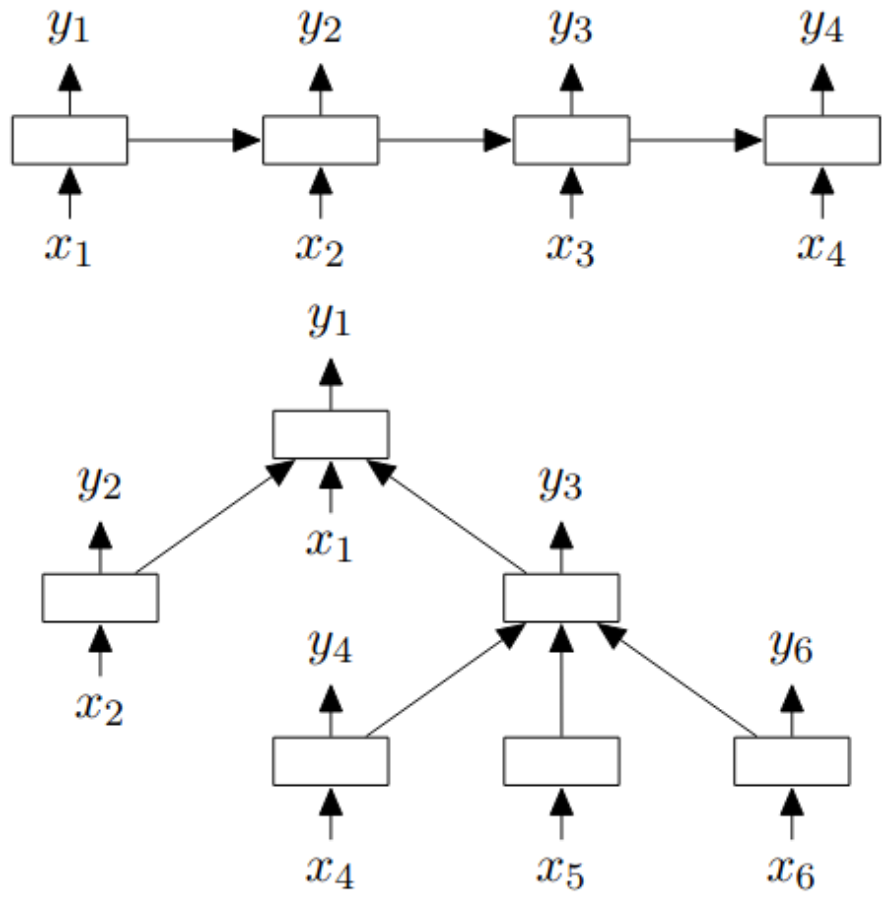


Figure 1: **Top:** A chain-structured LSTM network. **Bottom:** A tree-structured LSTM network with arbitrary branching factor.

Figure 2.4: LSTM, This project is mainly using tree-based structure LSTM network.[47]

next state. The decoder takes the vector representation of the input sentence and produces an output sentence. In the case of an RNN decoder it does it in steps, decoding the output one token at a time using the current state and what has been decoded so far.(Figure 2.6)

2.7 Attention Mechanism

Encoder-Decoder with attention mechanism was first brought out by Bahdanau et al., 2015. The research paper proposed the encoder-decoder model with an addictive attention mechanism.[5]

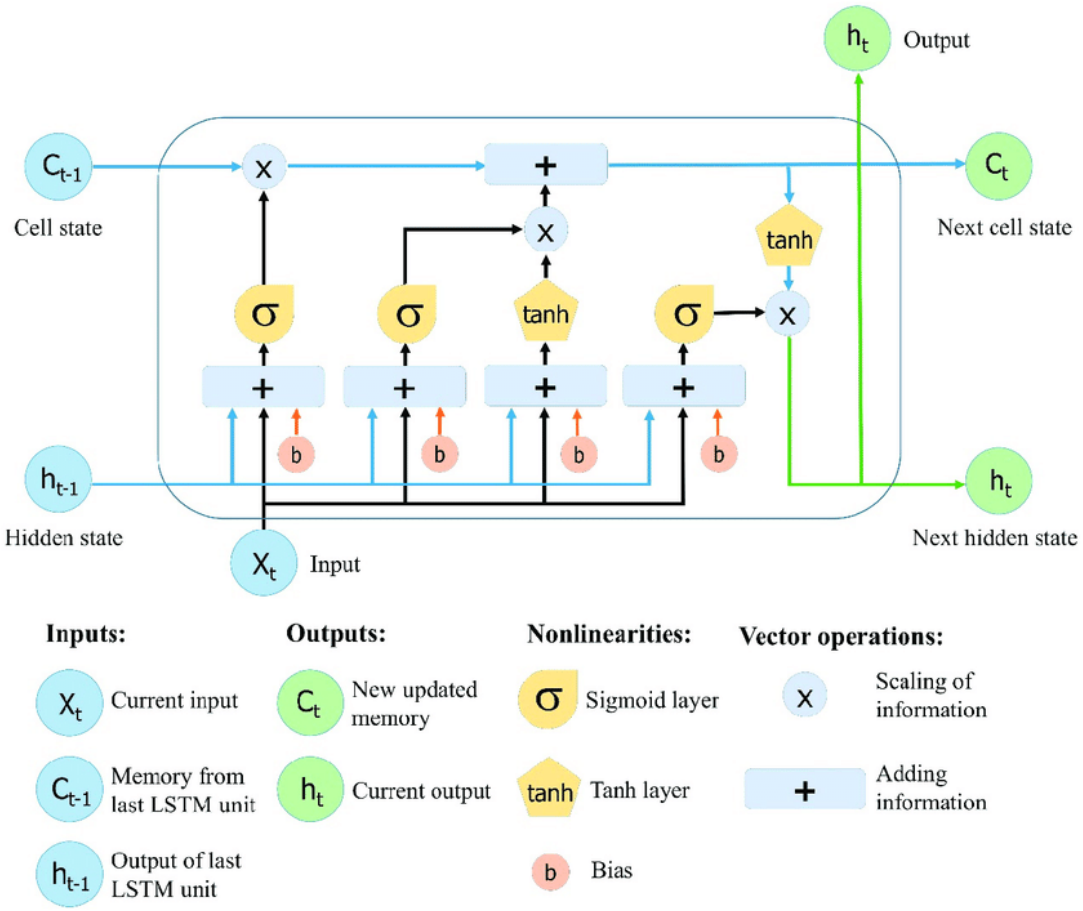


Figure 2.5: Structure of LSTM Network[27]

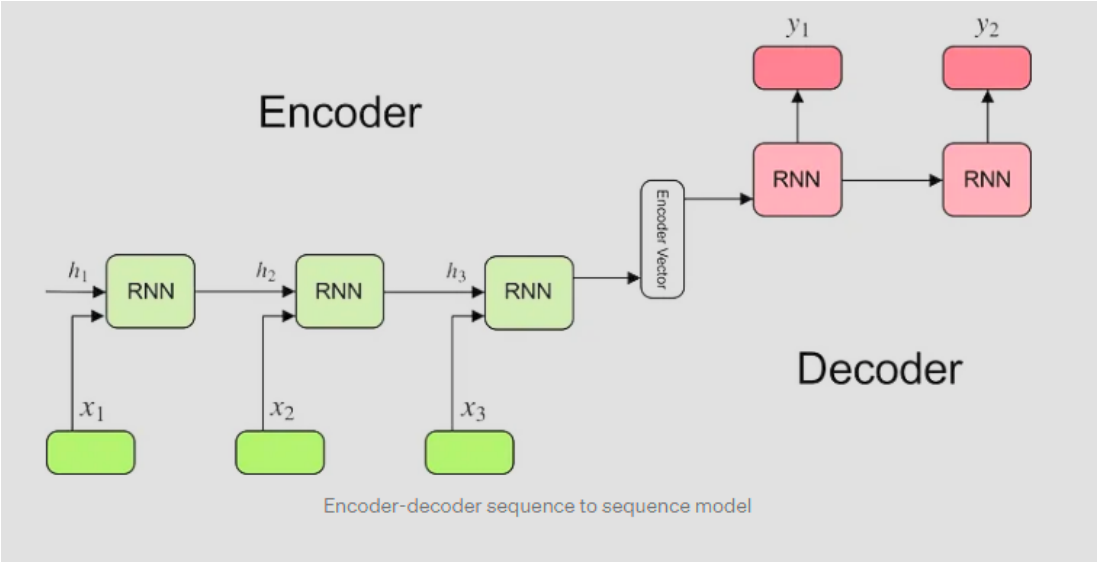


Figure 2.6: encoder and decoder sequence-to-sequence model[12]

In **Encoder and Decoder**, we can improve the translation by adding attention mechanism. Attention mechanism is a technique that allows neural network to focus on specific parts of an input sequence. This is done by assigning weights to different parts of the input sequence with the most important parts receiving the highest weights. By using the Attention model, the encoder passes a lot more data to the decoder. Instead of passing the final hidden state number to the decoder, the encoder passes all the hidden states from each time step. This therefore gives the decoder more context beyond just the final hidden state. The decoder then uses all the hidden state information to translate the sentence. The second change that this Attention model brings is adding an extra step to the attention decoder before producing its output. The extra step focuses only on the most relevant parts of the input. Firstly, it looks at the set of encoder states that it has received, each encoder Hidden State is associated with a certain word in the input sentence. Secondly, it gives each hidden state a score. Third, it multiplies each hidden states by its soft-maxed score. Thus, amplifying hidden states with the higher scores, and downsizing the ones with low scores. Putting all these together we get to see how the Attention Network operates. [41]

2.8 Word Embedding

The idea of 'Word Embedding' for machine translation plays a significant role in Natural Language Processing (NLP) and Machine Learning applications. Word embeddings are a method for capturing similarities among words in a text corpus by using models to predict the likelihood

of word co-occurrences within snippets of text. [6]These techniques became notably prominent in automated text analysis when it was shown they could discern analogies.Concrete example is shown in the figure. A noteworthy journey for Word embedding content is the **Word2Vec Model**, which is introduced by Tomad Mikolov and his colleagues. Word2Vec Model is consist of Continuous Bag of Words and Continuous Skip-gram, and this model learn word embedding in an efficient way. In 2013, Pennington et al. introduced G**lobal Vectors for Word Representation (GloVe)**, which contains the idea of global statistics. This model is used commonly nowadays for Natural Language Processing applications.(Figure 2.7)

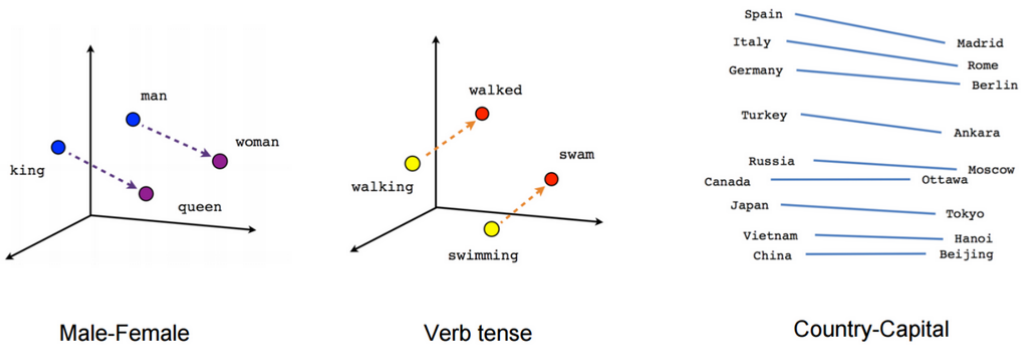


Figure 2.7: Word Embedding in 3-Dimension [8]

2.9 ML Models Comparison

2.9.1 Tree-to-Tree Neural Networks

The first part of the network, often called the ”encoder,” reads and interprets the original tree structure (like a sentence in English). It’s like learning the meaning and context of a whole sentence rather than just individual words. The second part, known as the ”decoder,” then uses this understanding to create a new tree structure in a different language or format (like translating that English sentence into French). This process is quite sophisticated because it’s not just about translating word for word but about understanding and reorganizing entire structures, maintaining the correct meaning and context in the new format. This capability makes tree-to-tree neural networks particularly useful in fields like natural language processing and automated code translation.[9]

2.9.2 Tree-Based Convolutional Neural Network (TBCNN)

This network, which is shown in figure 2.3 takes structured data in the form of trees and processes it in several stages to produce an output, often used for classification tasks. First, the tree's nodes are transformed into a vector representation, which numerically encodes the properties of each node. Then, a tree-based convolution operation is applied, which involves running filters over the tree structure to capture local features and hierarchical relationships between nodes. After convolution, a 3-way pooling method is used, which aggregates the features from different parts of the tree to create a fixed-size representation regardless of the tree's size. This pooled representation is then passed through a fully connected hidden layer that can learn complex patterns from the aggregated features. Finally, an output layer, often coupled with a softmax function, is used to classify the input tree into various categories based on the learned features. Each step is crucial in translating the structural information of a tree into a form that can be used for effective machine learning tasks.[28]

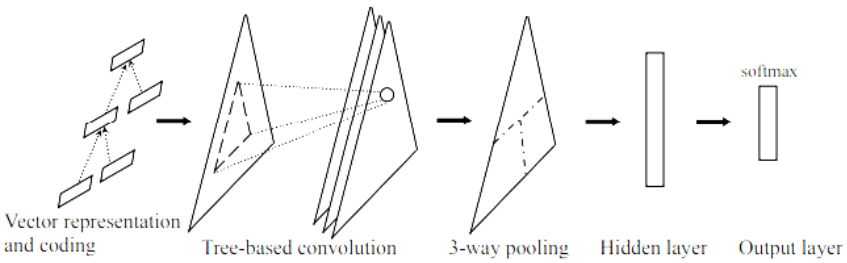


Figure 1: The architecture of the Tree-Based Convolutional Neural Network. The main components in our model include vector representation and coding, tree-based convolution and 3-way pooling. Then a fully-connected hidden layer and an output layer (softmax) are added.

Figure 2.8: Model Architecture shown from here[28]

2.9.3 Tree-to-Sequence Attentional Neural Machine Translation

The Tree-to-Sequence Attentional Neural Machine Translation architecture is a sophisticated framework designed for translating structured data, such as syntactic parse trees of sentences, into a sequential output, like text in another language. The architecture consists of an encoder-decoder mechanism, where the encoder processes the tree structure, capturing the syntactic nuances and hierarchical relationships of the input. Attention mechanisms are incorporated to focus on different parts of the tree when generating each word in the sequence, allowing the model to maintain context and handle long-range dependencies. The decoder then generates

the output sequence one element at a time, informed by the encoded representation and the attention distribution. This model excels in tasks where understanding the structural complexities of the input is crucial for producing accurate and coherent translations, as it can effectively map the intricate patterns of the source tree to the linear nature of the target sequence. [15]

2.9.4 Large-Scale Multi-Label Text Classification (LMTC) Method

The architecture for Large-Scale Multi-Label Text Classification (LMTC) involves several layers of processing to handle the complexity of assigning multiple labels to text documents. Starting with an input text, the data is first passed through an embedding layer, where words or phrases are transformed into numerical vectors that capture their semantic meaning. These embeddings are then fed into a series of neural network layers, which may include Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), or Transformer-based models with self-attention mechanisms, to capture the contextual relationships within the text. The advanced models might employ attention mechanisms to focus on relevant parts of the text for predicting each label. Afterward, the features extracted by the neural networks are used to predict the presence of each label, often through a sigmoid or softmax output layer that generates probabilities for each possible label. The final output is a set of labels that have been assigned to the input text, determined by thresholding the probabilities. This architecture is designed to scale to large datasets with a vast number of labels and can be enhanced with additional mechanisms like label-wise attention or hierarchical classification to improve performance. [43]

2.9.5 Sequence-to-Sequence Neural Networks

The core of a Seq2Seq model consists of two primary components: an encoder and a decoder, both of which are typically recurrent neural networks (RNNs), although modern variants might use LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Unit) cells to better capture long-range dependencies within the sequence. The encoder processes the input sequence, one element at a time, and generates a context vector, a condensed representation of the entire input sequence's information. This context vector is then fed into the decoder, which begins generating the target sequence, again one element at a time, using the context vector and the previous elements it has produced as inputs for each step. This process continues until a special end-of-sequence token is produced, signaling the completion of the sequence translation. Advanced Seq2Seq models may also incorporate attention mechanisms that allow the decoder to focus on different parts of the input sequence at each step, thereby improving the model's

ability to handle long and complex sequences by mitigating information loss and vanishing gradients issues common in basic RNNs.[10]

2.9.6 Sequence-to-Sequence v.s. Tree-to-Tree

The table below compares the performance of Tree2Tree and Seq2Seq neural network architectures in the context of translating between CoffeeScript and JavaScript, which are both programming languages. These architectures are evaluated based on program accuracy across different translation tasks, as indicated by the metrics CJ-AS, CJ-BS, CJ-AL, and CJ-BL, representing various test sets or translation aspects. [9] [10]

For Tree2Tree, the table shows two additional variants: one with parenthetical phrase (-PF) information and another with attention mechanisms (-Attn). The Seq2Seq also has a third variant, Seq2Tree, which suggests a sequence-to-tree adaptation, and a Tree2Seq indicating a tree-to-sequence adaptation.

The followings are the results' observations: Tree2Tree generally outperforms the standard Seq2Seq on most tasks, particularly in the CJ-AS and CJ-BS measures, which could be more syntax-focused, suggesting that the tree-based models capture the structural nuances of programming languages better than sequence-based models.[9]

The Tree2Tree with attention (-Attn) variant shows mixed results, indicating that attention mechanisms may offer benefits in certain contexts (such as CJ-BS for CoffeeScript to JavaScript translation). The Seq2Tree and Tree2Seq variants show that adapting the models to process input as a sequence and output as a tree, or vice versa, can also be effective, with some scores rivaling or exceeding the standard Tree2Tree and Seq2Seq.[9]

	Tree2tree			Seq2seq				Seq2tree		Tree2seq	
	T→T	T→T (-PF)	T→T (-Attn)	P→P	P→T	T→P	T→T	P→T	T→T	T→P	T→T
CoffeeScript to JavaScript translation											
CJ-AS	**99.57%**	98.80%	0.09%	90.51%	79.82%	92.73%	89.13%	86.52%	88.50%	96.96%	92.18%
CJ-BS	**99.75%**	99.67%	0%	97.44%	16.26%	98.05%	93.89%	91.97%	88.22%	96.83%	78.77%
CJ-AL	**97.15%**	71.52%	0%	21.04%	0%	0%	0%	80.82%	78.60%	82.55%	46.94%
CJ-BL	**95.60%**	78.61%	0%	19.26%	9.98%	25.35%	42.08%	76.12%	76.21%	83.61%	26.83%
JavaScript to CoffeeScript translation											
JC-AS	**87.75%**	85.11%	0.09%	83.07%	86.13%	73.88%	86.31%	86.86%	86.99%	71.61%	86.53%
JC-BS	**86.37%**	80.35%	0%	80.49%	85.94%	69.77%	85.28%	85.06%	84.25%	66.82%	85.31%
JC-AL	**78.59%**	54.93%	0%	77.10%	77.30%	65.52%	75.70%	77.11%	77.59%	60.75%	75.75%
JC-BL	**75.62%**	44.40%	0%	73.14%	73.96%	61.92%	74.51%	74.34%	71.56%	57.09%	73.86%

Table 1: Program accuracy for the translation between CoffeeScript and JavaScript.

Figure 2.9: Comparison of tree2tree and sequence2sequence models shown from here [9]

2.10 Website Development for Program Translation

Since the word limitations and this project is focusing more on the machine learning part, this section for background will generally introduce some of the tools that used in this project.

2.10.1 Frontend

Node.js

Node.js is an open-source, cross platform JavaScript runtime environment that executes JavaScript code outside a web browser. Node.js is built on Chrome's V8 JavaScript Engine. It was first developed in 2009 by Ryan Dahl. Its latest version 15.14 was released in April 2021. Developers employ Node.js for building web applications on the server side, making it ideal for applications that require intensive data processing due to its asynchronous, event-driven architecture[44]

React

React is a JavaScript library crafted by Facebook and utilized in constructing Instagram.com, empowers developers to efficiently forge responsive user interfaces for both websites and applications. The cornerstone of React.js lies in the concept of the virtual DOM, a JavaScript-based component tree fashioned with React that serves as a proxy for the actual DOM tree. Its primary purpose is to minimize DOM manipulations, ensuring React components are updated in the most efficient way possible.[22]

2.10.2 Backend

SpringBoot

Spring Boot is a robust extension of the Java Spring framework. It simplifies the development process for stand-alone, production-grade applications. It streamlines the setup and deployment process, which allows developers to bypass the extensive configuration details that are typically required and jump straight into building their applications. Spring Boot is known for its ability to easily package applications into self-contained executables, which makes deployment and scaling straightforward in various environments.[24]

Flask

Flask is a nimble web framework that equips developers with the tools to swiftly construct web applications. Developed by Armin Ronacher of the Pocco international Python enthusi-

asts group[17], Flask leverages the WSGI toolkit and the Jinja2 engine for templating. It is positioning itself as a prime choice for building web applications that require a lightweight and efficient backend with minimal setup.[17]

2.10.3 DataBase

MongoDB

MongoDB is a leading NoSQL database favored for its flexibility and scalable approach to data storage. It's a document-oriented database, which means it stores information in JSON-like documents, making data integration for certain types of applications faster. Unlike relational databases that require predefined schemas, MongoDB's schema-less nature allows developers to work with more agile data models that can be adjusted on the fly. This feature with its powerful querying and indexing capabilities, makes MongoDB a versatile choice for businesses that deal with large volumes of diverse, complex data needing quick iteration.[18]

NoSQL (Not Only SQL)

NoSQL databases represent a broad class of database management systems that diverge from the traditional relational database model. They are designed to handle large volumes of unstructured or semi-structured data, accommodating a wide variety of data types and offering flexible schemas for data storage and retrieval. NoSQL databases are particularly suited for big data applications and real-time web applications, providing scalability, high performance, and ease of development. They often utilize non-relational mechanisms for storing and retrieving data, such as key-value pairs, wide-column stores, graphs, or document-oriented repositories. NoSQL allows for efficient data processing in distributed systems where horizontal scaling is essential.[2]

2.11 DockerHub

Docker Hub is a cloud-based registry service that provides a centralized resource for container image discovery, distribution, and change management. Every docker can be represented as one little Linux. It enables developers and DevOps teams to share and publish container images, facilitating collaboration and speeding up the deployment process. Docker Hub offers a vast collection of public and private repositories, hosting images for a wide range of applications, from web apps to databases and operating systems. Users can pull images from Docker Hub

to their local environment or push their custom images to Docker Hub for storage and sharing. This platform supports automated builds and workflows, allowing for seamless integration with GitHub and Bitbucket. Docker Hub simplifies the management and deployment of Docker containers across different environments. [11]

2.12 Postman

Postman is a popular software development tool that simplifies the process of building, testing, and modifying APIs (Application Programming Interfaces). It offers a user-friendly interface that allows developers to send HTTP requests to a server and view the responses. By facilitating the examination of API responses, documenting APIs, and automating tests, Postman streamlines the collaboration among developers in API development. Its wide range of features, including the ability to create collections of requests, define environments with variable sets, and write scripts for testing API responses, makes it an essential tool in modern software development workflows for both individual developers and teams.

2.13 Amazon Web Services (AWS)

Amazon Web Services (AWS) provided by Amazon, is a comprehensive cloud computing platform. It delivers a variety of services, including infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) for different organizational needs like computing power, database storage, and content delivery. Since its inception in 2006, AWS has become a leading cloud service provider globally, empowering businesses of various scales to enhance and expand their operations through affordable, flexible, and scalable computing solutions. AWS enables the deployment of diverse applications, ranging from web and mobile applications to data processing, warehousing, and archival solutions. [1]

Chapter 3

Literature Review: Program Translation Solution Research

In this chapter, we explore advanced neural network models that have proven effective in program translation, as validated by recent research findings. The important of these models is their relationship with the Tree2Tree model—a groundbreaking approach that has significantly advanced the field. The objective is to highlight the foundational principles of these models, illustrate their real-world applications, and demonstrate their connection to the Tree2Tree approach.

3.1 Transformer Model

Transformer model is first introduced and proposed by a landmark research *'Attention is All You Need'* by Google research.[52] It is the architecture that involves in self-attention to improve the performance of deep learning Natural Language Programming models. Self-attention mechanism is a model to weigh the importance of different parts of the input data differently. Self-attention mechanism boosts the efficiency of processing the entire sequence of data in parallel, which make it more efficient than RNNs and LSTMs for long sequences. For the weighting, the attention mechanism adjusts how much focus to put on each part of the input data. Therefore, it enables the model to be contextually aware. Moreover, attention weights can be analysed and visualised, offering insights into the model's decision-making process. The introduction of self-attention mechanism in transformer model has led to significant advances across a range of deep learning applications and modern AI models.[52](Figure 3.1)

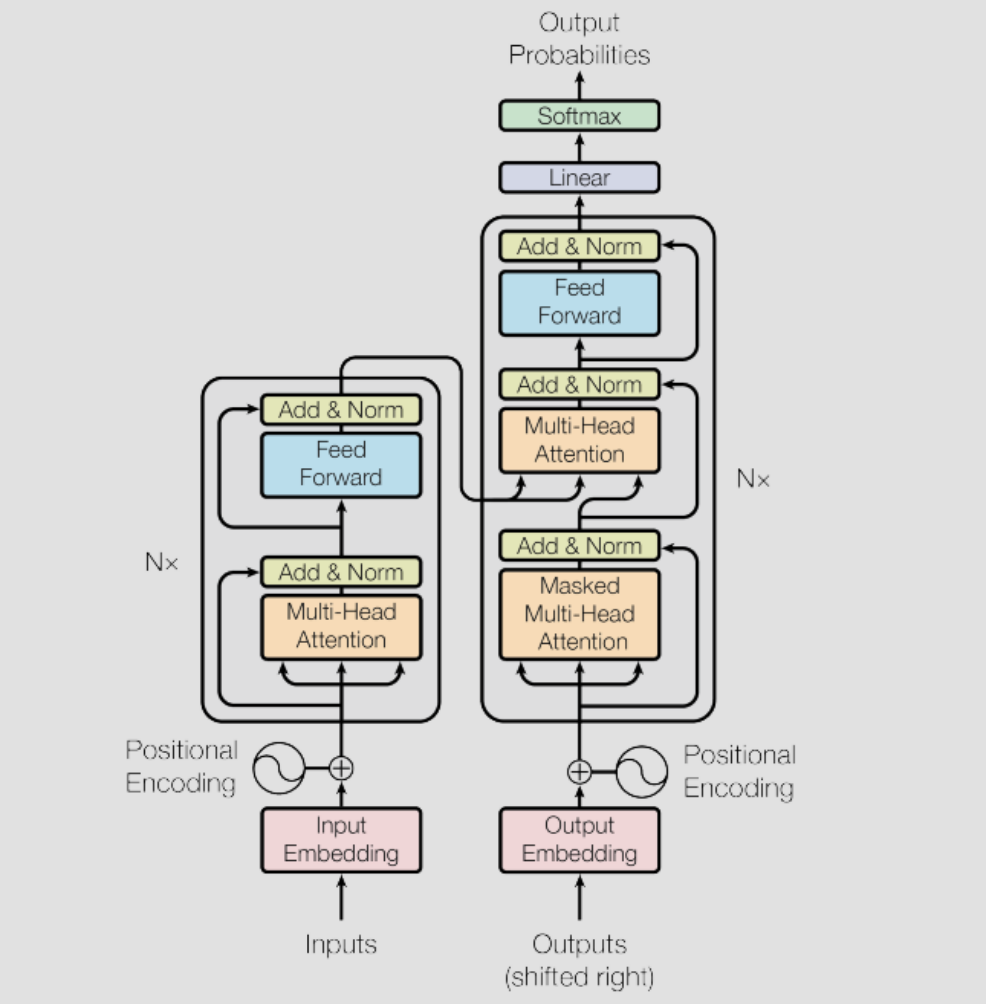


Figure 3.1: The Transformer Model Architecture[52]

3.1.1 Tree-based Transformer Model

Tree-based Transformer Model is indeed based on the Transformer model architecture. It is designed to incorporate the structural and hierarchical information in programming languages through the abstract syntax trees (ASTs).

Relative Research Paper-TreeGen Model

A research paper *TreeGen: A Tree-Based Transformer Architecture for Code Generation* proposed TreeGen for program generator. This innovative model addresses the challenges of long dependency issues and the modeling of code structures inherent in programming languages.TreeGen model [45]leverages the attention mechanism of Transformers to handle long dependencies effectively. Moreover, it enhances its ability to understand the relationships between different parts of the code that may be separated by several lines. Additionally, TreeGen introduces an AST (Abstract Syntax Tree) reader, or encoder, to integrate grammar rules and AST structures directly into the network. The model was evaluated on a Python benchmark and two semantic parsing benchmarks, showing significant improvements in accuracy and the ability to handle complex programming tasks.[45]

3.1.2 ChatGPT

Generative Pre-trained Transformer (GPT) from *OpenAI* is a popular language model which first introduced in November 2022. ChatGPT uses transformer model with self-attention. It does not consider as a 'Transcoder', since it's a large language model which predicts the next word or token in sequence based on the words or tokens that come before it. The architecture in this model are RNN, attention mechanism, Transformer architecture, pre-training and fine-tuning, language modeling, and generative pre-training. [20] Unlike the dedicated tools or compilers which are specifically trained for transcoding, ChatGPT can't execute code to verify its semantic equivalence between the source and target language. ChatGPT basically responses based on patterns and examples it has been training without understanding of programming logic.

3.1.3 GitHub Copilot

GitHub Copilot is an AI programming tool which was introduced in June 2021. This tool learns from all of the Github repositories and benefits the programmers with coding issues. This tool

generally implement the transformer model which demonstrates an advanced application of Transformer models in understanding and generating code.[36]

3.1.4 Seq2Seq Neural Network

As I stated in previous chapter for the introduction of sequence-to-sequence model, it is often implemented with RNNs and LSTMs. Moreover, RNNs and LSTMs were the first deep learning approaches to tackle the transcoding task. However, there's still some limitations of this model. While this model is effective for short sequences, Seq2Seq models can struggle with long dependencies and complex syntax structures in source code.

3.1.5 Seq2Seq and Tree2Tree Research paper for program translation

Chen et al.has done with the comparison of models between tree2tree, seq2seq, seq2tree, and tree2seq to transcode between Coffeescript and JavaScript. As the result, tree2tree model has an overall better performance in better program accuracy. The findings indicate that the tree-to-tree model excels at grasping the relationship between source and target programs, notably outperforming other baseline models, especially when dealing with longer inputs.

Chapter 4

Design

This chapter mainly focus on the Tree-to-Tree encoder-decoder architecture which is main focus of this machine learning program translation project. This design of model is fully inspired by the *Tree-to-Tree Program Translation* research paper by Chen et al.[9].

4.1 Tree-to-Tree Neural Network

4.1.1 Program Translation as a Tree-Based Problem

As the title suggested, in the research paper from Chen et al. considered the program translation problem as the translation between the source and the target tree in translating between CoffeeScript and JavaScript. The idea of translating between parse trees is demonstrated in figure 4.1. Similarly, translation between pascal and Java could possibly converting via utilising this idea.

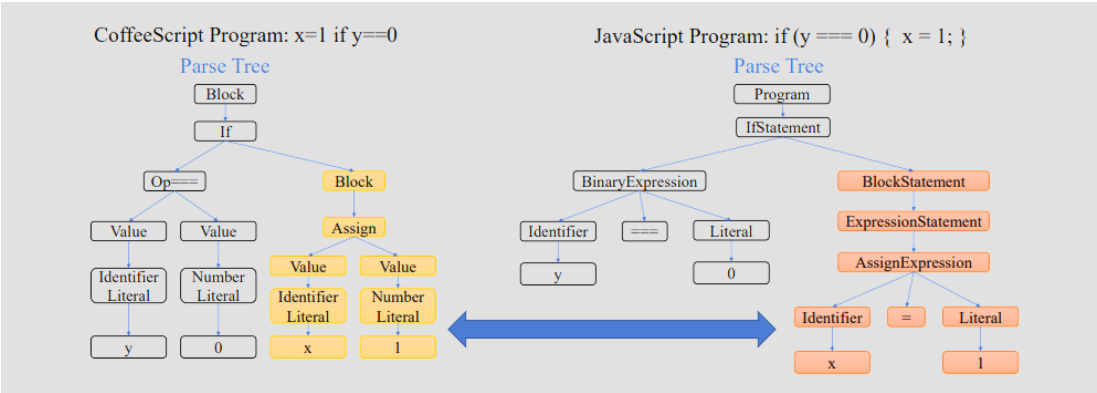


Figure 4.1: Translating CoffeeScript and JavaScipt between parse trees[9]

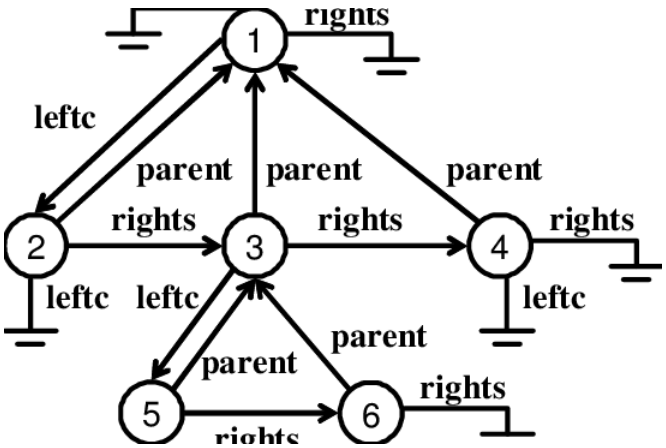


Figure 4.2: Left-Child-Right-Siblings example diagram[27]

4.1.2 Convert into Binary Trees

There will be many branches in the source and target trees. Chen et al. has noticed that applying trees with binary trees is a more effective way to work with structures that are inherently not binary. The first step from Chen et al. is that they firstly converted the source tree and the target tree (trees where nodes can have any number of children) into a binary tree. The method of *Left-Child-Sibling (LCRS)* is a decent method to convert a general tree into a binary tree. (Figure 4.2)

How Left-Child Sibling (LCRS) works

- **Left Child**: The left child of node in the binary tree represents the node's first child in the original tree.
- **Right Child**: The right child of a node in the binary tree represents the node's next sibling in the original tree.

After Conversion to Binary Trees

- **Left Child**: The left child, representing the first child of the original node in the multi-branch tree.
- **Right Child**: The right child, representing the next sibling of the original node in the multi-branch tree.

$$(h, c) = \text{LSTM}(([h_L; h_R], [c_L; c_R]), x)$$

Figure 4.3: Encoder Formula

4.1.3 Architecture

The hierarchical architecture is based on a supervised machine learning tree-based encoder and decoder model complementing with the idea of parent attention feeding mechanism.

Encoder

Chen et al. employed the specialised technique known as Tree-LSTM model in the encoder part. This model generates embedding for the whole tree and its each sub-trees.

Imagine we're focusing on a specific node within this tree, which we'll call Node N. This node is represented by a unique identifier called 'ts', and it has two offspring: a left child (NL) and a right child (NR). To figure out the embedding for Node N, we start from the very bottom of the tree and work our way up.

Let's think of Node N's children as having their own little information packets or 'LSTM states', denoted as (hL, cL) for the left child and (hR, cR) for the right child, where 'h' represents the hidden state and 'c' signifies the cell state of the LSTM. Additionally, Node N's identifier 'ts' is transformed into an embedding, which we'll refer to as 'x'.

To calculate the LSTM state (h, c) for Node N, we essentially merge the information from its children and its own identifier. We do this by combining the hidden states (hL and hR) and the cell states (cL and cR) of its children, along with the embedding 'x' of Node N's identifier. This combination is then fed into the LSTM, which gives us the new state (h, c) for Node N.

It is possible that node N might be missing a left child, a right child or both. If this happens, it is better to pretend the missing child(ren) has a state of zero, essentially filling in the gap, to ensure that our calculations can proceed without interruption (Figure4.3).[9]

Decoder

The decoder's job is to create the target tree, beginning with just the root node. It kick-starts this process by taking the LSTM state (which includes both the hidden state 'h' and the cell state 'c') from the root node of the source tree and applying it to the root node of the target tree. Following this initial step, the decoder keeps track of all the nodes that need further expansion in a queue and goes through them one by one, expanding each in turn.

During this step-by-step expansion, the decoder selects a node from the queue to work on,

$$\mathrm{t_t} = \mathbf{argmax}\ \mathbf{softmax}(We_t)$$

Figure 4.4: Decoder Formula

referred to as the "expanding node." The first task with the expanding node is to determine its value. To do this, the decoder calculates an embedding 'et' for the expanding node 'N'. This embedding 'et' is then used as input to a softmax regression network, which makes a prediction about the node's value.

The decision of what value the node should take is made through a process that involves a mathematical operation known as "softmax" applied to the product of 'et' and a trainable matrix 'W'. The matrix 'W' has dimensions that correlate with the output's vocabulary size 'Vt' (i.e., the total number of different values the node could potentially take) and the dimension 'd' of the embeddings. The operation aims to select the most likely value (denoted as 'tt') for the expanding node from the available vocabulary.

The embedding 'et' for the expanding node isn't just plucked from thin air; it's carefully computed using an "attention mechanism." Through this methodical process, the decoder gradually builds out the target tree, one node at a time, ensuring that each node's value is the best fit based on the context provided by the source tree and the structure of the target tree itself.[9]

In the decoding process for tree generation, each node is assessed to decide if it should expand further, with possibilities including non-terminal nodes, terminal nodes, or a special 'EOS' token indicating no further expansion. For nodes that expand, the decoder creates left and right children, each assigned unique LSTM states derived from specific parameters and a word embedding matrix 'B'. This process, which involves queuing new nodes for expansion and using the 'EOS' token to manage the tree structure efficiently, continues until no nodes remain to be expanded. Crucially, the decoder's design allows it to distinguish between terminal and non-terminal nodes without explicit grammar rules, learning over time to apply this knowledge accurately and ensuring the structural integrity of the generated tree. (Figure 4.4 and 4.5)[9]

Parent Attention Feeding Mechanism

Parent attention feeding is a specialized twist on the traditional attention mechanism, designed specifically for tasks that involve hierarchical data, like parsing programming languages or natural language sentences into tree structures. While the attention mechanism helps a model to focus on relevant parts of the input data, parent attention feeding takes this a step further by specifically incorporating information from parent nodes in a tree structure to enhance the

$$e_t = \tanh(W_1 e_s + W_2 h)$$

where W_1, W_2 are trainable matrices of size $d \times d$ respectively.

Figure 4.5: Embedding that includes the hidden state of the expanding node[9]

prediction accuracy for their child nodes.

The term "feeding" refers to how the model uses the information from parent nodes. It's like feeding the model a bit of extra helpful information. When the model is about to generate or translate a new piece of code (create a new child node), it doesn't do so blindly. Instead, it "feeds" off the context provided by the parent nodes, using that information to make a more informed decision about what the child node should be.

Parent Attention Feeding Mechanism in source subtrees

The tree-to-tree model uses a special technique called an attention mechanism to focus on specific parts of the source tree, especially when dealing with deeper nodes in the target tree. Normally, each node's embedding (et) could be based just on its own hidden state (h). But if we did that, the deeper we go into the target tree, the more we'd lose track of the source tree's details, leading to not-so-great outcomes. To solve this, the attention mechanism helps our model remember and utilize the important parts of the source tree by focusing on the specific sub-tree that matches the section of the target tree we're currently expanding. This way, we keep the valuable source tree information in mind, even for the deep nodes of the target tree, enhancing the model's overall performance.

How Parent Attention Feeding Mechanism works in the model

In the context of a model working with tree structures, like translating one programming language to another, each node in the output tree (representing a piece of code) is like a person in the family tree. When deciding how to translate a particular piece of code (a child node), the model doesn't only look at the direct translation but also considers the "parent" piece of code - essentially, the broader context or function within which the current line of code operates.

Benefits of the Parent Attention Feeding Mechanism

Overall, the research shows that using the Parent Attention Feeding Mechanism generally improve the performance of program translation. Firstly, this mechanism allows the model to maintain a richer contextual understanding as it processes each node. Secondly,the added

context from parent nodes leads to more precise embeddings and, subsequently, better overall model performance. Lastly, it also makes the efficiency of handling the long-range dependencies.

4.1.4 Challenges and Considerations

In my exploration of program translation between Pascal and Java, a critical consideration emerges from the fundamental differences in the programming structures of these languages. Unlike the commonly referenced research on translating between JavaScript and CoffeeScript, which share a similar grammar structure, Pascal and Java present a big contrast. Pascal, with its roots in educational and procedural programming paradigms. Java is a language designed for object-oriented applications and widely used in enterprise environments, embody distinct syntactical and conceptual frameworks. The grammar and program language structure poses unique challenges for the tree-to-tree model. Lastly, overall flow diagram for the Tree2Tree LSTM Encoder-Decoder Model with Parent Attention Feeding Mechanism is shown in figure 4.6.

4.2 DataSet

Since Pascal Programming Language is lack of dataset resources online nowadays, the creation of special Pascal Dataset is necessary. Moreover, since the Java datasets need to be exactly having the same outcome or similar meaning transcoded from Pascal datasets, the Java dataset is translated from Pascal by myself and some help from ChatGpt to validate the correctness of the programs.

4.2.1 Amount of Datasets Design

Since the time limitations for submitting the project, this project only provides 15 pairs of training datasets, 3 pairs of validation datasets, and 5 pairs of testing datasets. Machine learning are mostly trained by 80 percentages of training datasets and model performance is tested on the other 20 percentages of the datasets. Therefore, my datasets amount is decided by this indication of suggested percentage amount.[7]

4.2.2 Difference Grammar between Pascal & Java

Here's the listed main and obvious difference between Pascal and Java's data structure and syntax difference.

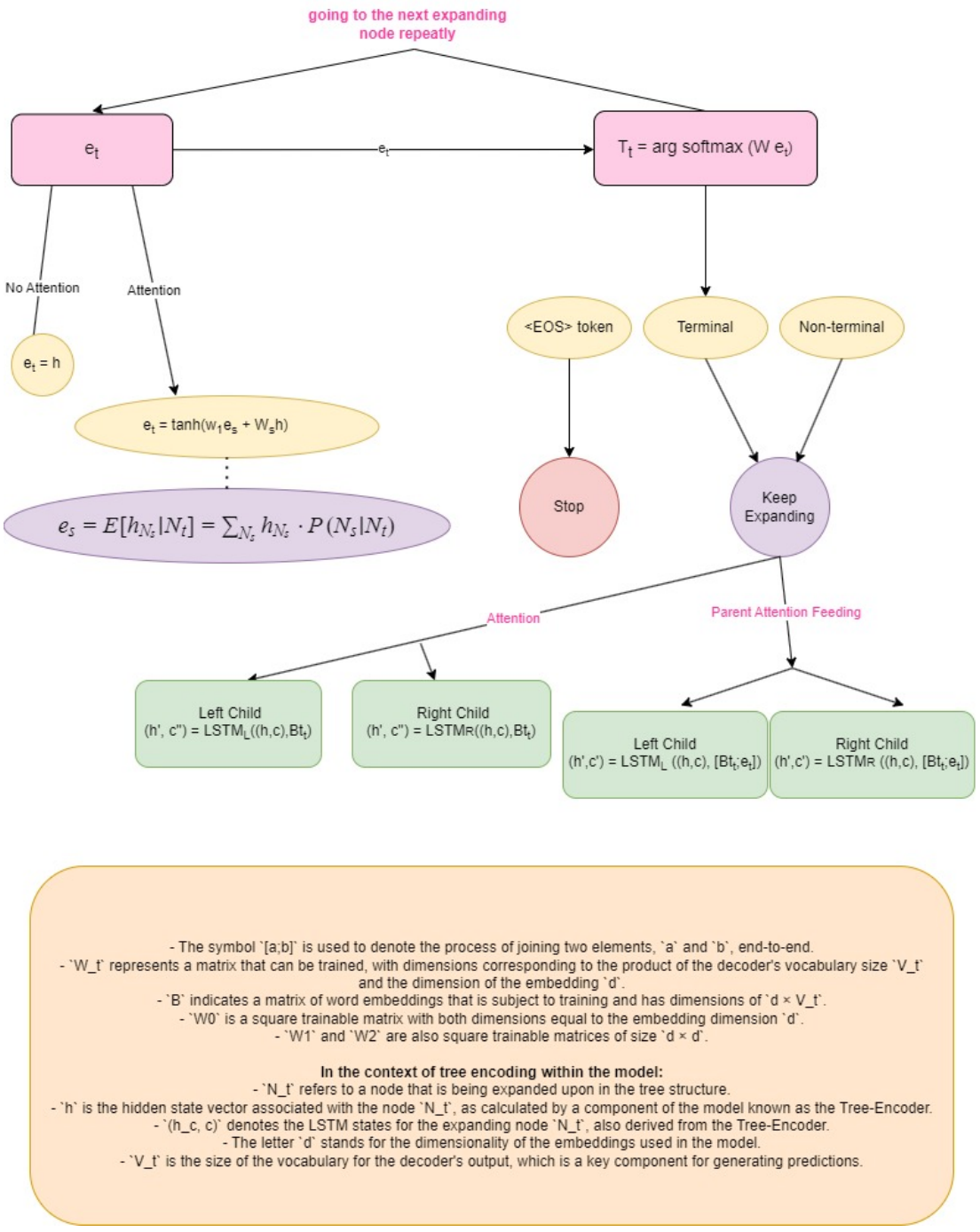


Figure 4.6: Overall Flow Diagram for the Tree2Tree LSTM Encoder-Decoder Model with Parent Attention Feeding Mechanism

Program Structure:

- Pascal: Traditionally uses a more rigid structure, starting with a program header, followed by a declaration section, and finally the program body within a begin ... end. block. It encourages structured programming with a strong emphasis on data structures.
- Java: Uses a class-based structure where the code is contained within classes and methods. The entry point of a Java program is the public static void main(String[] args) method within any class.

Typing Discipline:

- Pascal: Has a strong typing system but allows for more explicit control over data sizes via subrange types and packed structures.
- Java: Also strongly typed, but it abstracts away much of the memory management details. Java does not support unsigned types, unlike Pascal.

Memory Management:

- Pascal: Offers manual control over memory allocation and deallocation, with the programmer responsible for managing the lifecycle of objects and data structures.
- Java: Automates memory management through garbage collection, freeing the programmer from manual memory management tasks.

Procedure and Function Calls:

- Pascal: Distinguishes between procedures (which do not return a value) and functions (which do).
- Java: Uses methods for both procedures and functions. Methods in Java belong to a class and can return a value or be void.

Pointers:

- Pascal: Directly supports pointers, allowing manipulation of memory addresses explicitly.
- Java: Does not support explicit pointers. References are used to access objects, but direct memory address manipulation is prohibited to maintain security and integrity.

Concurrency:

- Pascal: Earlier versions did not have built-in support for concurrency, though modern Object Pascal (Delphi) includes support for multithreading.
- Java: Has built-in support for multithreading and concurrency, allowing for the creation and synchronization of threads directly in the language.

Error Handling:

- Pascal: Uses traditional error handling mechanisms, such as returning error codes from functions or using external libraries.
- Java: Implements exception handling, providing a structured way to catch and handle errors and exceptions through try-catch blocks.

Inheritance and Polymorphism:

- Pascal: Traditional Pascal does not support object-oriented programming. However, Object Pascal, an extension, supports classes, inheritance, and polymorphism.
- Java: Is inherently object-oriented, with support for classes, inheritance (single inheritance for classes and multiple inheritance for interfaces), and polymorphism.

4.2.3 Design for Datasets Pairs

Due to the difference between Java and Pascal, I have tried my best to translate a similar meaning and data structure in the pair of Pascal and Java Datasets.(figure 4.7)

4.3 Evaluation Metrics Design

4.3.1 Program Accuracy

In Chen et al.[9] research paper for Tree-Based LSTM Encoder-Decoder Model, it proposed Program Accuracy as the main metrics for validation. The formula is given below:

$$\text{Program Accuracy} = \frac{\text{Number of Accurate Output Programs}}{\text{Number of All Programs}}$$

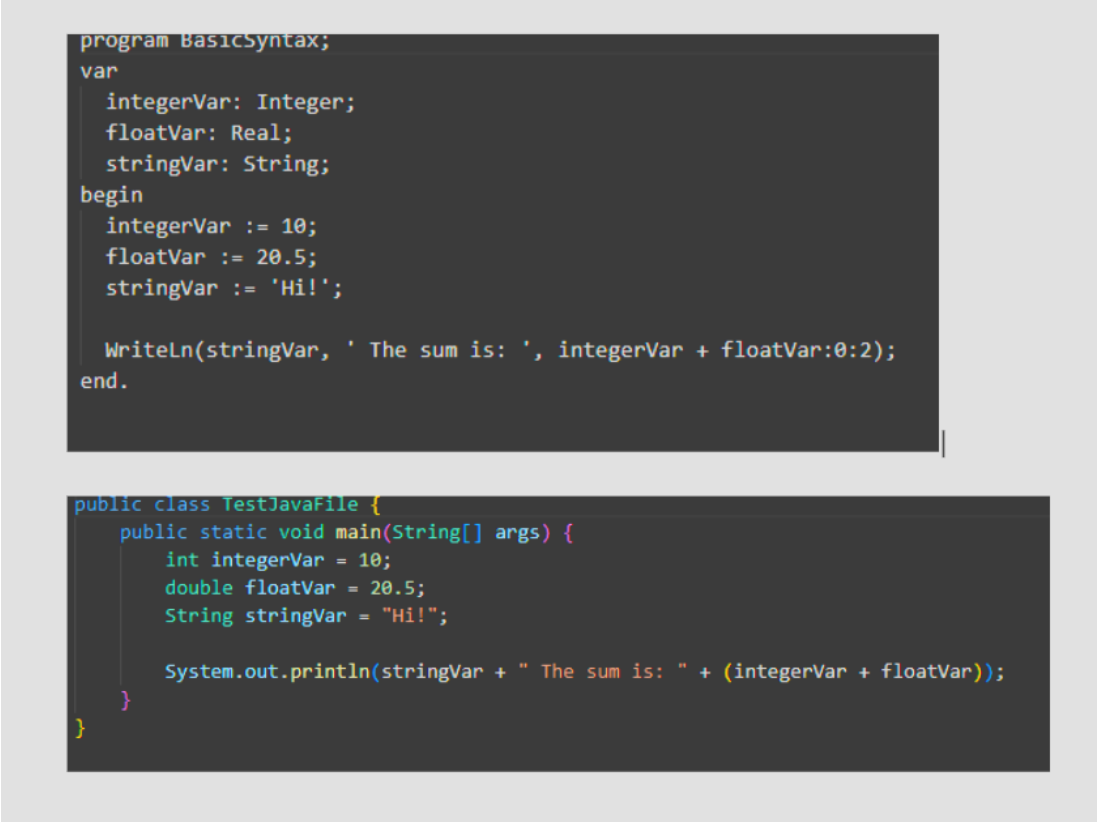


Figure 4.7: The comparison of pascal to Java translation dataset example is provided here. The datasets are mostly parallel to each other, except for the class and program name.

4.3.2 Token Accuracy

In Chen et al.[9] research paper, it also proposed Token Accuracy as another main metrics for validation. The formula is given below:

$$\text{Token Accuracy} = \frac{\text{Number of Accurate Tokens}}{\text{Number of All Tokens}}$$

4.3.3 Token Edit Distance Ratio (EDR)

In both research papers of 'Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code' and 'Lexical Statistical Machine Translation' by [30][29] used EDR for Language Migration. This metric quantifies the amount of work required to modify the generated code by adding or removing code tokens, in order to convert the output into its correct form. The formula is given below:

$$\text{EDR} = \frac{\sum_{\text{methods}} \text{EditDistance}(s_R, s_T)}{\sum_{\text{methods}} \text{length}(s_T)}$$

$\text{EditDistance}(s_R, s_T)$ represents the edit distance between the reference code snippet s_R and the translated code snippet s_T, indicating the minimum number of token edits (insertions or deletions) needed to transform s_T into s_R. The denominator, $\sum_{\text{methods}} \text{length}(s_T)$, aggregates the lengths of all translated code snippets, thus normalizing the edit distance by the total length of the translation output.

4.4 Software Architecture Design

In operation, when a translation request is initiated from the frontend, the backend receives this request and interacts with the database to determine if the required data is available. If it is, the translated information is swiftly sent back to the frontend. In cases where the data is not present in the database, the Tree2Tree model is invoked to perform the translation. The model, trained on abstract syntax trees (ASTs), computes the translation which is then stored in MongoDB to improve future query responsiveness. Notably, the Tree2Tree model is continuously trained and improved upon, ensuring that the system evolves and maintains high accuracy and efficiency in program translation tasks. This architectural setup not only streamlines the translation process but also facilitates the maintenance of a robust and scalable application.(figure 4.8)

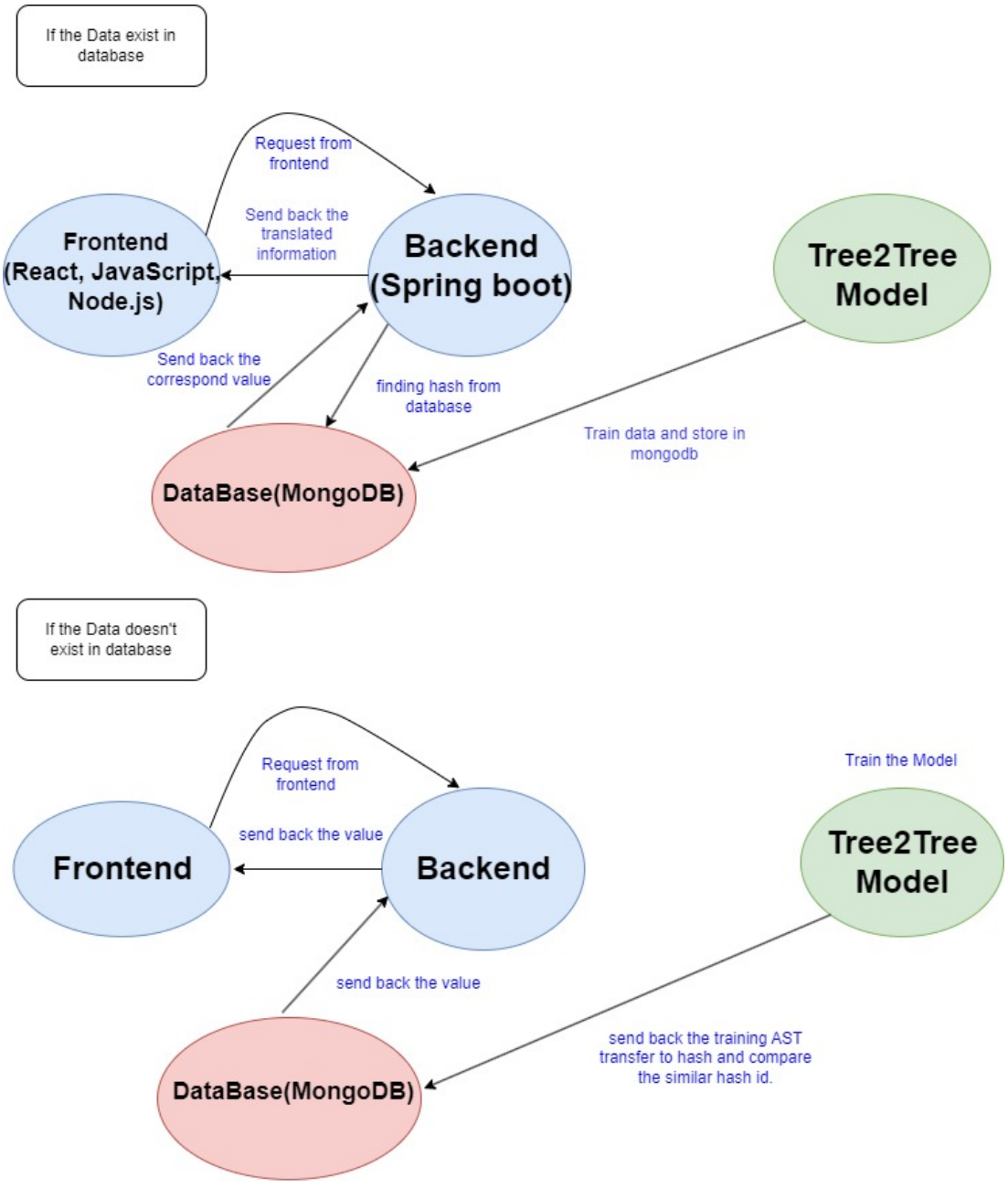


Figure 4.8: The software architecture flowchart design is shown from here

Chapter 5

Specification

5.1 Environment Requirement

This section mainly introduces the environment requirement of this project. The original Tree2Tree system code was firstly provided by my project supervisor - Kevin Lano, but the Tree2Tree model coding is from *Tree-to-Tree Neural Networks for Program Translation* research paper from Chen et al..

5.1.1 Python versions and Packages

The neural network execution requires Python3 environment. The software for this neural network will also required to install additional packages. The additional packages are provided in the requirement.txt file in this project. Lastly, this project requires Python 3.6 to 3.8 version to run the training and test for the Tree2Tree model.

5.1.2 ANTLR4

ANTLR4(ANother Tool for Language Recognition) is a powerful parser generator. It can be used as a tool for grammar translation or a runtime library for most of the programming languages. By inputting the program into the ANTLR4, it generates the AST compilation of the program. Since the translation from Pascal to Java is the topic of this project, generating AST trees for both languages program is necessary. An example of generating the AST from a pascal program can be shown in figure 5.1, figure 5.2, and figure 5.3.

Installation Steps

The steps for generating Java and Pascal ANTLR AST form are exactly the same.

1. Open IntelliJ IDEA.
2. Go to $File > Settings > (for macOS, go to "IntelliJ IDEA" > "Preferences")$.
3. In the "Plugins" section, search for "ANTLR v4" and install the plugin.

To configure the ANTLR v4 plugin:

1. After the plugin is installed, restart IntelliJ IDEA.
2. Create a new project or open an existing one.
3. Add a Java grammar file (such as Java.g4) to your project (available at [3]).
4. Right-click on the .g4 file and select "Configure ANTLR".
5. In the pop-up dialog, set up the location for the generated code and other options.

To generate the parser and lexer:

1. Right-click on the .g4 file and select "Generate ANTLR Recognizer".
2. This will automatically generate the code for the parser and lexer.

To write Java code to use the generated parser:

1. Create a new Java class file.
2. Write code within this file to instantiate and utilize the parser created from the .g4 file.
3. The structure of the code will be similar to the previously mentioned example, but you may need to adjust it according to the generated code and project structure.

To run parser:

1. Execute the Java class that just created to process Java source files and generate an AST (Abstract Syntax Tree).
2. Validate the structure of the AST by printing it or inspecting it in another manner.

To process and traverse the AST:

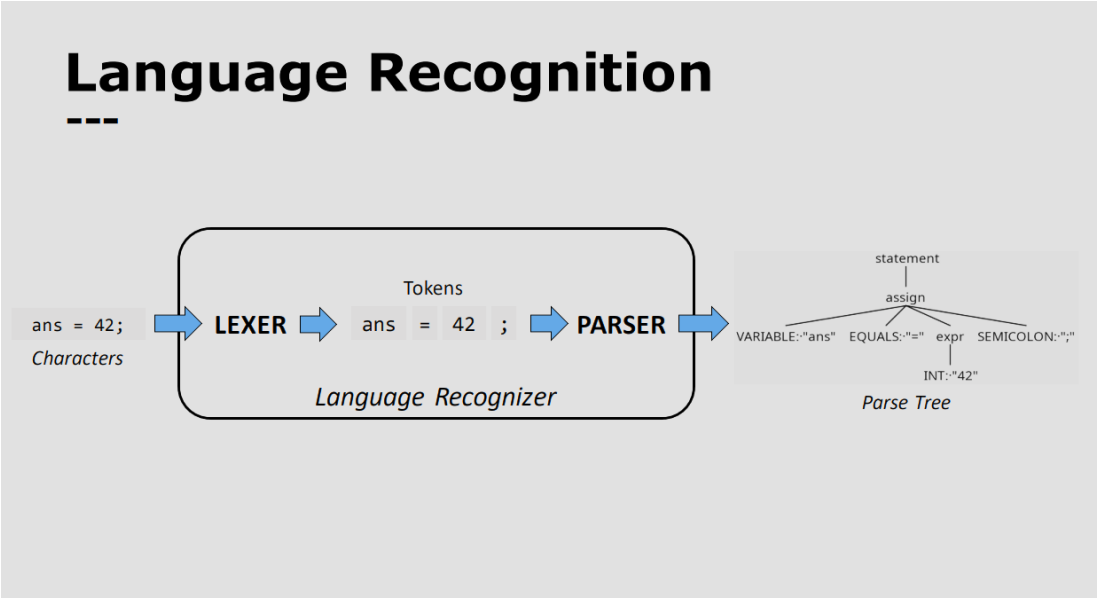


Figure 5.1: ANTLR4 Language Recognition[26]

- To process the generated AST in greater depth, you can create a visitor or listener class and override relevant methods to navigate and manipulate the AST.

One of the advantages of using IntelliJ IDEA is that it offers a user-friendly interface and streamlined workflow, especially for the configuration and code generation processes for ANTLR. With the integrated plugin, it can manage the generation process and outcomes of ANTLR 4 more conveniently.

5.1.3 Virtual Environment to Train Model

The original version of the tree2tree is from Chen et al., but the program was written in a old version of PyTorch. Therefore, I spent a lot of time on the code fixing part to match and run with the recent version of PyTorch. Using a virtual environment with pip for this project offers several benefits that significantly enhance the project's development, maintainability, and scalability.

- **Dependency Management**: A virtual environment can create an isolated space for the project. This allows installing specific versions of libraries and dependencies required by this project.

- **Experimentation and Testing:** Virtual environments make it easy to experiment with different versions of libraries or to test new features in isolation from the main project environment.

```
program Factorial;

function Factorial(n: LongInt): LongInt;
begin
    if n = 0 then
        Factorial := 1
    else
        Factorial := n * Factorial(n - 1);
end;

var
    number, result: LongInt;
begin
    Write('Enter a positive integer: ');
    ReadLn(number);

    if number < 0 then
        WriteLn('Please enter a positive integer or zero.')
    else
    begin
        result := Factorial(number);
        WriteLn('Factorial of ', number, ' is ', result);
    end;
end.
```

Figure 5.2: Pascal Program example

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Users\selin\IntelliJ IDEA 2023.3.3\lib\idea_rt.jar=60946:C:
(program (programHeading program (identifier Factorial) ;) (block (procedureAndFunctionDeclarationPart (procedureOrFun
```

Figure 5.3: A chunk of the Pascal program AST example

- **Cleanliness:** Keeping the project's dependencies separate from the global Python environment helps avoid clutter and potential conflicts with other projects.
- **Efficiency**: Managing dependencies through virtual environment can help optimising resource usage, as only the necessary packages are installed and maintained for the project.

Through the project, the environment can be installed by these command below.By typing in the command lines, the required version of package will be installed successfully.

For Windows:

```
$ cd tree2tree
$ python -m venv my_virtual_environment
$ virtual_environment_name\Scripts\activate
$ pip install -r requirements.txt
```

For Python3:

```
$ cd tree2tree
$ python3 -m venv my_virtual_environment
$ virtual_environment_name\Scripts\activate
$ pip3 install -r requirements.txt
```

For MacOs:

```
$ cd tree2tree
$ python -m venv my_virtual_environment
$ source virtual_environment_name/bin/activate
$ pip install -r requirements.txt
```

5.1.4 PyTorch

PyTorch is a significant package for most of the machine learning or neural network systems projects. PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR). It's widely used for applications in deep learning and artificial intelligence. PyTorch provides two high-level features:

- **Tensor Computing**: Powerful GPU acceleration, such as NumPy.
- **Deep Neural Networks**: Built on a tape-based auto-grad system enabling flexible model building and effortless optimisation methods.

Chapter 6

Implementation

6.1 Tree2Tree Model

The implementation of Tree2Tree model was the code from Chen et al[9]. In the original files from Chen et al. implemented Tree2Tree, Seq2Seq, Seq2Tree, and Tree2Seq models. We are focusing on the discussion of Tree2Tree model, and the deletion of the code of the unnecessary models are all deleted in the source code. Since the code from Chen et al. was used in deprecated PyTorch Version, and the code was not fully commented as well, this project fixed the problems to allow the model and software run successfully. Moreover, the ReadMe and requirement.txt files are all updated in this project to provide a successful runable software.

6.1.1 Tree2Tree Implementation

Main Tree2Tree model files are `translate.py`, `network.py`, `data_utils.py`, and `Tree.py`.This section will introduce each function in each files in the neural network.

`translate.py`

`translate.py` is the main and central execution file. This file serves as the main entry point for training and testing a Tree2Tree neural network model, specifically designed for program translation tasks. It begins by importing necessary libraries and defining a range of parameters and options that can be customized via command-line arguments. These include various training parameters such as learning rate, batch size, the architecture of the neural network, and file paths for data sets and pre-trained models.

The script is structured to support both training and testing phases of the Tree2Tree model.

In the training phase, the model is initialized, and its weights are updated iteratively using a given training dataset. The process involves feeding batches of data to the model, calculating loss, performing backpropagation, and applying gradient clipping. The model's performance is periodically evaluated on a validation set, and checkpoints are saved, especially when the model achieves a new best score based on evaluation loss.

For testing, the script loads a pre-trained model and evaluates its performance on a test dataset. The evaluation considers metrics like loss and token accuracy and calculates the average Edit Distance Rate (EDR) to measure how close the model's output is to the target sequences.

In addition to the main training and testing routines, the script includes functions to compute and log the depth of abstract syntax trees, save translation results to files, and handle vocabularies such as saving them to CSV files.

Lastly, this file determines whether to run the training or testing routine based on the command-line arguments provided, with the test data being loaded and used if the test flag is specified. Otherwise, the training routine is executed using the specified training and validation datasets.

`network.py`

This file represents a module for a Tree-to-Tree Neural Network for program translation. The code is split into two main components: ‘TreeEncoder‘ and ‘Tree2TreeModel‘, both important for understanding and translating the syntactic and semantic structure of source code via abstract syntax trees (ASTs).

The ‘TreeEncoder‘ is a subclass of ‘nn.Module‘ and is responsible for transforming the input code into a meaningful representation that captures the hierarchical structure of programming languages. It embeds the source vocabulary into a dense space and utilizes LSTM units to process the tree structure of the code, calculating hidden states for each node that are used in the subsequent translation process.

The ‘Tree2TreeModel‘ combines the functionalities of an encoder and a decoder. It initializes the encoder to process the input code and set up the framework for the decoder to generate the target AST. The decoder predicts each node's value in the target tree starting from the root, using the encoded source tree for context. This prediction process involves the attention mechanism if enabled, which helps the model focus on relevant parts of the input when generating each node in the target tree. This is particularly important in program translation, where

the correspondence between different parts of the source and target code is not always linear.

In addition to the forward prediction process, the script manages batches of data for training and prediction, maintaining states and ensuring that the gradient flow is properly managed by the repackage_state function. The attention mechanism within this model is crucial for aligning the nodes from the source to the target tree, essentially allowing the model to "look" at the input while generating the output.

The implementation highlights several deep learning techniques specialized for the domain of program translation. For instance, the handling of variable tree structures, the encoding of syntactic and semantic information, and the use of attention to maintain context throughout the translation process. This sophisticated architecture embodies the cutting-edge efforts to teach machines the complex task of understanding and transforming code between different programming languages.

`data_utils.py`

This file sets the stage for neural network-based program translation by preparing and organizing the data required for the model to learn effectively. It first generates vocabularies from training datasets for both the source and target programming languages. These vocabularies map unique tokens, found in the code, to distinct numerical IDs. Special symbols to denote the beginning and end of statements, as well as placeholders for unknown tokens, are integrated into the vocabularies to maintain structural integrity during translation.

For training involving ASTs, the file provides functionality to convert these trees into token IDs based on the generated vocabularies. This process occurs recursively to ensure every node in the tree is accounted for and accurately represented. When dealing with sequential formats, the program is converted into a sequence of token IDs, appending an end-of-sequence marker to signal completion.

The file also includes methods for serializing ASTs into a linear sequence of tokens, crucial for models that expect sequential input. Additionally, it comprises functions to calculate statistics such as the maximum number of children in a tree and average sequence lengths, offering valuable insights into the dataset's composition. These pre-processing steps culminate in the creation of a data structure pairing source and target elements, either as tokenized trees or sequences, setting the neural network up for a successful training phase.

`Tree.py`

`Tree.py` is a core component of a Tree2Tree neural network model, primarily responsible for defining and managing the data structures essential for representing programs as trees in program translation tasks. It includes the 'BinaryTree' and 'Tree' classes, which provide a framework for each node in the tree, holding essential attributes such as the root value, child nodes, parent node, and depth information.

The 'BinaryTree' class is specialized in representing nodes with a binary structure, allowing for the left and right children, which is common in abstract syntax tree (AST) representations of programming languages. This class also includes state information pertinent to LSTM computations within the neural network, such as hidden and cell states, attention details, and prediction outcomes for each node, encapsulating the necessary data for model training and inference.

'TreeManager' is a class that manages collections of trees ('BinaryTree' instances), providing methods to create new trees, build them from dictionary structures (often derived from serialized program representations), and manipulate or access tree nodes by their unique identifiers. It's equipped with methods to clear previous states, which is crucial for resetting the model's state between different data samples or epochs during training.

The file's functionality is vital for constructing the input and output representations required by a Tree2Tree neural network. It helps in converting raw program code into a hierarchical tree structure that the neural network can process, ensuring that the data is adequately prepared for learning the patterns needed for program translation tasks.

6.1.2 Encoder Implementation

The LSTM Encoder is implemented by the class of TreeEncoder in `network.py`. Here's a overview of the implementation of Tree Encoder.

- Initialization: The Tree Encoder is initialized with specific parameters including the size of the source vocabulary, embedding size, hidden layer size, and batch size. It also initializes various neural network components such as embeddings for the source vocabulary and several linear layers for processing the tree nodes.

- Embedding the Nodes: Each node in the input tree is associated with a token from the source vocabulary. The encoder uses an embedding layer to convert these tokens into dense vector representations. This step is crucial for capturing the semantic information

of each node in a continuous vector space.

- Recursive Tree Traversal: The encoding of the tree is performed in a bottom-up fashion. Starting from the leaf nodes, the encoder works its way up to the root node. At each step, it combines the information from a node's children (if any) using a specific LSTM-based calculation described next.

- LSTM-based Child Integration: For a given node, the encoder first retrieves the embeddings of its left and right children (if present). If a node lacks children, zero vectors are used instead. The encoder then applies several linear transformations and nonlinear activations (such as sigmoid and tanh) to these embeddings and combines them in a manner akin to the operations of an LSTM cell. This process involves computing gates (input, forget, output) and cell state updates to integrate the children's information effectively.

- Calculating the Node's Representation: The encoder calculates the final representation of the current node by combining its own embedding with the processed information from its children. This representation captures not only the semantic meaning of the node itself but also the structural information from its position in the tree.

- Encoding the Entire Tree: By recursively applying the process described above to each node in the tree, starting from the leaves and ending at the root, the encoder builds a comprehensive representation of the entire tree. The output is a set of vector representations for each node, including the final, aggregated representation at the root, which captures the entire tree's information.

- Preparing for Decoding: The encoded representations are then passed to the Tree Decoder component of the model. To facilitate attention mechanisms or further processing, the encoder also outputs additional structures, such as attention masks that indicate the valid parts of the tree.

6.1.3 Decoder Implementation

The LSTM Encoder is implemented particularly in TreeEncoder class' 'forward' function. The forward function orchestrates the encoding of the entire tree. It employs LSTM-like gates and states to aggregate information across the tree structure, ensuring that the encoded representation reflects the hierarchical nature of the data.

```python
class TreeEncoder(nn.Module):
    def __init__(self,
                 source_vocab_size,
                 embedding_size,
                 hidden_size,
                 batch_size
                 ):
        super(TreeEncoder, self).__init__()
        self.source_vocab_size = source_vocab_size
        self.embedding_size = embedding_size
        self.hidden_size = hidden_size
        self.batch_size = batch_size
        self.cuda_flag = cuda.is_available()

        self.encoder_embedding = nn.Embedding(
            self.source_vocab_size, self.embedding_size)

        self.ix = nn.Linear(self.hidden_size, self.embedding_size, bias=True)
        self.ih = nn.Linear(self.hidden_size, self.hidden_size)
        self.rh = nn.Linear(self.hidden_size, self.hidden_size)

        self.fx = nn.Linear(self.hidden_size, self.embedding_size, bias=True)
        self.fh = nn.Linear(self.hidden_size, self.hidden_size)
        self.rh = nn.Linear(self.hidden_size, self.hidden_size)

        self.ox = nn.Linear(self.hidden_size, self.embedding_size, bias=True)
        self.oh = nn.Linear(self.hidden_size, self.hidden_size)
        self.rh = nn.Linear(self.hidden_size, self.hidden_size)

        self.ux = nn.Linear(self.hidden_size, self.embedding_size, bias=True)
        self.uh = nn.Linear(self.hidden_size, self.hidden_size)
        self.rh = nn.Linear(self.hidden_size, self.hidden_size)

    def calc_root(self, inputs, child_h, child_c): ···

    def encode(self, encoder_inputs, children_h, children_c): ···

    def forward(self, encoder_managers): ···
```

Figure 6.1: The calc_root and encode functions form the backbone of the Tree Encoder module, enabling the transformation of complex tree-structured data into a format amenable to neural network processing.

The forward Function: A Detailed Process

The forward function operates in a bottom-up manner, starting from the leaf nodes and progressing to the root, which involves the following steps:

- Initialization and Node Grouping: The function begins by initializing a queue to group nodes for processing. It then iterates through each node in the input tree, starting from the leaves, and groups nodes based on whether their children's states have been computed. This grouping is essential for efficient batch processing.

- Computing Node Representations: For each group of nodes, the function computes their LSTM-like states using the calc_root method. This computation involves determining input, output, and forget gates, similar to a standard LSTM, but within the context of a tree structure. The states of a node's children are used in combination with the node's own value to compute its final representation.

- Aggregating Information to the Root: The computed states are then aggregated from the leaves towards the root. This step ensures that the representation at each node captures not just its own information but also that of its subtree.

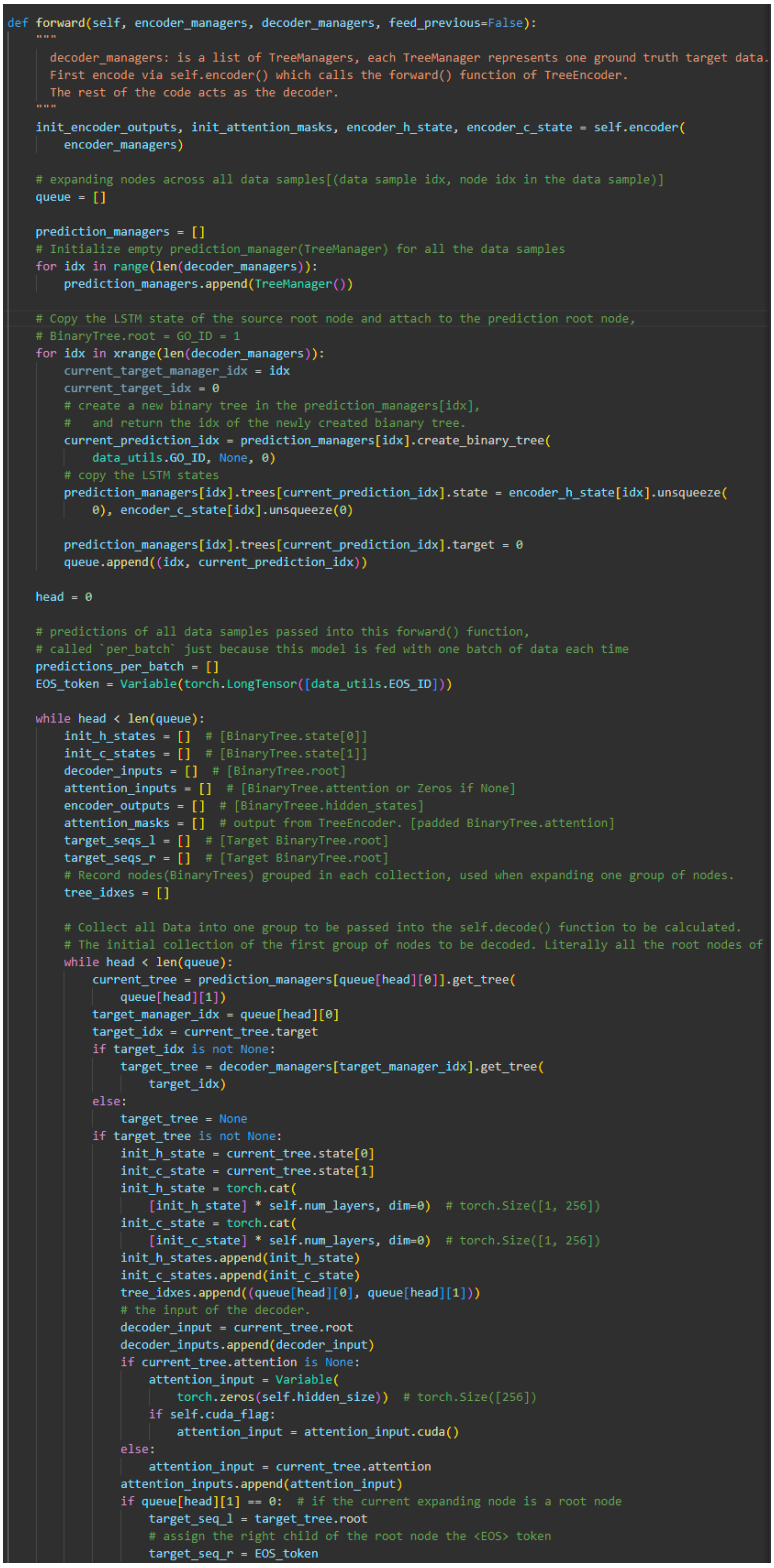
- Encoding the Entire Tree: The process repeats until the root node's representation is computed. This final representation encodes the entire tree's structural and semantic information, serving as the input for the model's decoding phase.

```
def decode(self, encoder_outputs, attention_masks, init_state, init_decoder_inputs, attention_inputs):
    """
    called from forward():
    predictions_logits_l, predictions_logits_r, states_l, states_r, attention_outputs_l, attention_outpu
    = self.decode(encoder_outputs, attention_masks, (init_h_states, init_c_states), decoder_inputs, atte
    @param: encoder_outputs: hidden states obtained from the encoder
    @param: init_decoder_inputs: the BinaryTree.root collection of the prediction manager.

    This function predicts the t_t(before argmax) value of the left child and right child of a node(Bina
    """
    embedding = self.decoder_embedding(init_decoder_inputs)
    state_l = init_state
    state_r = repackage_state(init_state)
    if self.no_pf:
        decoder_inputs = embedding
    else:  # parent feeding
        decoder_inputs = torch.cat([embedding, attention_inputs], 2)

    output_l, state_l = self.decoder_l(decoder_inputs, state_l)  # LSTM_l
    output_r, state_r = self.decoder_r(decoder_inputs, state_r)  # LSTM_r
    output_l = output_l.squeeze()
    if len(output_l.size()) == 1:
        output_l = output_l.unsqueeze(0)
    output_r = output_r.squeeze()
    if len(output_r.size()) == 1:
        output_r = output_r.unsqueeze(0)
    prediction_l, attention_output_l = self.predict(
        output_l, encoder_outputs, attention_masks)
    prediction_r, attention_output_r = self.predict(
        output_r, encoder_outputs, attention_masks)
    return prediction_l, prediction_r, state_l, state_r, attention_output_l, attention_output_r
```

Figure 6.2: Decode Function is shown from here



```
def forward(self, encoder_managers, decoder_managers, feed_previous=False):
    """
      decoder_managers: is a list of TreeManagers, each TreeManager represents one ground truth target data.
      First encode via self.encoder() which calls the forward() function of TreeEncoder.
      The rest of the code acts as the decoder.
    """
    init_encoder_outputs, init_attention_masks, encoder_h_state, encoder_c_state = self.encoder(
        encoder_managers)

    # expanding nodes across all data samples[(data sample idx, node idx in the data sample)]
    queue = []

    prediction_managers = []
    # Initialize empty prediction_manager(TreeManager) for all the data samples
    for idx in range(len(decoder_managers)):
        prediction_managers.append(TreeManager())

    # Copy the LSTM state of the source root node and attach to the prediction root node,
    # BinaryTree.root = GO_ID = 1
    for idx in xrange(len(decoder_managers)):
        current_target_manager_idx = idx
        current_target_idx = 0
        # create a new binary tree in the prediction_managers[idx],
        #   and return the idx of the newly created bianary tree.
        current_prediction_idx = prediction_managers[idx].create_binary_tree(
            data_utils.GO_ID, None, 0)
        # copy the LSTM states
        prediction_managers[idx].trees[current_prediction_idx].state = encoder_h_state[idx].unsqueeze(
            0), encoder_c_state[idx].unsqueeze(0)

        prediction_managers[idx].trees[current_prediction_idx].target = 0
        queue.append((idx, current_prediction_idx))

    head = 0

    # predictions of all data samples passed into this forward() function,
    # called `per_batch` just because this model is fed with one batch of data each time
    predictions_per_batch = []
    EOS_token = Variable(torch.LongTensor([data_utils.EOS_ID]))

    while head < len(queue):
        init_h_states = []  # [BinaryTree.state[0]]
        init_c_states = []  # [BinaryTree.state[1]]
        decoder_inputs = []  # [BinaryTree.root]
        attention_inputs = []  # [BinaryTree.attention or Zeros if None]
        encoder_outputs = []  # [BinaryTreee.hidden_states]
        attention_masks = []  # output from TreeEncoder. [padded BinaryTree.attention]
        target_seqs_l = []  # [Target BinaryTree.root]
        target_seqs_r = []  # [Target BinaryTree.root]
        # Record nodes(BinaryTrees) grouped in each collection, used when expanding one group of nodes.
        tree_idxes = []

        # Collect all Data into one group to be passed into the self.decode() function to be calculated.
        # The initial collection of the first group of nodes to be decoded. Literally all the root nodes of
        while head < len(queue):
            current_tree = prediction_managers[queue[head][0]].get_tree(
                queue[head][1])
            target_manager_idx = queue[head][0]
            target_idx = current_tree.target
            if target_idx is not None:
                target_tree = decoder_managers[target_manager_idx].get_tree(
                    target_idx)
            else:
                target_tree = None
            if target_tree is not None:
                init_h_state = current_tree.state[0]
                init_c_state = current_tree.state[1]
                init_h_state = torch.cat(
                    [init_h_state] * self.num_layers, dim=0)  # torch.Size([1, 256])
                init_c_state = torch.cat(
                    [init_c_state] * self.num_layers, dim=0)  # torch.Size([1, 256])
                init_h_states.append(init_h_state)
                init_c_states.append(init_c_state)
                tree_idxes.append((queue[head][0], queue[head][1]))
                # the input of the decoder.
                decoder_input = current_tree.root
                decoder_inputs.append(decoder_input)
                if current_tree.attention is None:
                    attention_input = Variable(
                        torch.zeros(self.hidden_size))  # torch.Size([256])
                    if self.cuda_flag:
                        attention_input = attention_input.cuda()
                else:
                    attention_input = current_tree.attention
                attention_inputs.append(attention_input)
                if queue[head][1] == 0:  # if the current expanding node is a root node
                    target_seq_l = target_tree.root
                    # assign the right child of the root node the <EOS> token
                    target_seq_r = EOS_token
```

Figure 6.3: The forward function of the TreeEncoder plays an important role in the Tree-to-Tree model's ability to process hierarchical, tree-structured data. The invocation of 'self.encoder()' initiates the 'forward()' method within the 'TreeEncoder' class.

6.2 Converting Parse Trees to Abstract Syntax Tree Representations

- Argument Parsing: The script begins by defining command-line arguments using argparse, allowing the user to specify details such as the input folder for parse tree files, the output file name for the JSON representation, and the source language. It also includes an option for processing a single example file instead of a batch of files.
- Building the AST: The build_ast function converts a list of tokens, derived from the parse tree string, into a dictionary representing the AST. This dictionary has a root key, indicating the current node's value, and a children key, listing its child nodes. The conversion is done using a stack to manage the tree's hierarchical structure, where opening parentheses indicate the start of a new tree node and closing parentheses indicate the end of the current node's definition.
- Tokenization: The get_token_list function preprocesses the input parse tree string by inserting spaces around parentheses, allowing for easy tokenization into a list of tokens. This list is then used to build the AST.
- JSON File Generation for an Example Parse Tree: The write_one_json function is a utility that generates a JSON file from a single parse tree, specified by the –example_file argument. It's intended for testing or demonstration purposes.
- Preparing Data:The prepare_data function orchestrates the process of converting multiple parse tree files into their JSON representations. It iterates over specified input directories (containing parse tree strings for either the source or target language), builds ASTs for each, and collects the results in a list. Each item in the list corresponds to one input file's AST, encapsulated within a dictionary alongside placeholder keys for the original program text (not used in the script). The final list is written to a JSON file, as specified by the –result_file_name argument.

Here's the Workflow: The script can be run in two modes: one for processing a single example file and another for batch processing files in a specified directory. The mode is determined by the –example flag. For batch processing, it reads parse tree strings from text files within a specified directory, converts each string to a list of tokens, and then builds an AST from each token list. The resulting ASTs are collected into a list, with each AST stored in a dictionary format that includes placeholders for the source and target program texts (though the script

```python
def build_ast(token_list):
    stack = []
    for token in token_list:
        if token == '(':
            stack.append(token)
        elif token == ')':
            inside_parenthese = []
            cur = stack.pop()
            while cur != '(':
                inside_parenthese.append(cur)
                cur = stack.pop()

            if not inside_parenthese:
                    # Special case: '(' and ')' are the tokens themselves with no content between them.
                left_parenthesis_dict = {'root': '(', 'children': []}
                right_parenthesis_dict = {'root': ')', 'children': []}
                # Create a special structure to hold both parentheses as siblings
                # This assumes your AST format allows for this kind of structure.
                # Alternatively, adjust this part according to your specific AST requirements.
                parentheses_dict = {'root': 'parentheses', 'children': [left_parenthesis_dict, right_parenthesis_dict]}
                stack.append(parentheses_dict)
                continue  # Continue to the next token

            inside_parenthese.reverse()
            root = inside_parenthese.pop(0)
            children = inside_parenthese # may be empty or many children
            root_dict = {'root': root, 'children':[]}
            for child in children:
                if isinstance(child, dict):
                    root_dict['children'].append(child)
                elif isinstance(child, str):
                    child_dict = {'root':child, 'children':[]}
                    root_dict['children'].append(child_dict)
            stack.append(root_dict)
        else:
            stack.append(token)
    ast_dict ={}
    if len(stack) > 0:
        ast_dict = stack[0]
    return ast_dict
```

Figure 6.4: buildAST function shown from here

does not populate these placeholders). Finally, the script outputs the list of AST dictionaries as a single JSON file, suitable for use in further processing or as input to tree-based machine learning models.

To build the json AST file from orginal datasets:

```
$ python build_json_from_parse_tree.py
--folder test_data
--result_file_name source_pascal_target_java_test

$ python3 build_json_from_parse_tree.py
--folder test_data
--result_file_name source_pascal_target_java_test
```

```
def write_one_json():
    """
    Generates a JSON file for one example string-formatted parse tree.
    """
    file_name = args.example_file
    file = open(file_name, 'r')
    file_str = file.read()
    token_list = get_token_list(file_str)
    dict_ast = build_dict_ast(token_list)
    result_file_dir = file_name[:-3]+'json'
    result_file = open(result_file_dir, 'w')
    result_file.write(json.dumps(dict_ast))
    result_file.close()
    print('Example JSON file is written.')
```

Figure 6.5: write one json function shown from here

```
def get_token_list(ast_string):
    """
    Processes a string-formatted parse tree into a list of tokens.

    param ast_string: The string representation of the AST.
    param file_name: The file name containing the AST string (unused in the function).
    return: A list of tokens.
    """
    ast_string = ast_string.replace('(',' ( ')
    ast_string = ast_string.replace(')',' ) ')
    token_list = ast_string.split()
    return token_list
```

Figure 6.6: get token list function shown from here

6.3 Metrics Implementation

6.3.1 Program Accuracy

In the evaluate function from Translate.py file, program accuracy assesses the correctness of entire output sequences compared to the target sequences. It is calculated by checking if an entire sequence from the output matches its corresponding target sequence exactly. Calculation Process is that for each sequence, if all tokens match the target sequence, acc_programs is incremented. Dividing acc_programs by the total number of sequences (tot_programs) yields the program accuracy. The formula is given below:

$$\text{Program Accuracy} = \frac{\text{Number of Accurate Output Programs}}{\text{Number of All Programs}}$$

6.3.2 Token Accuracy

Within the evaluate function in translate.py file, token accuracy is calculated by comparing each token in the output predictions (output_predictions) against the corresponding token in the target sequences (current_target). The implementation or calculation process is each token comparison that matches, acc_tokens is incremented. The total number of correct tokens (acc_tokens) divided by the total number of tokens in all target sequences (tot_tokens) gives the token accuracy. The formula is given below:

$$\text{Token Accuracy} = \frac{\text{Number of Accurate Tokens}}{\text{Number of All Tokens}}$$

6.3.3 Token Edit Distance Ratio (EDR)

The EDR metric is calculated using the edit_distance function, which computes the edit distance between each pair of output and target sequences. The evaluate function uses this to calculate the EDR across all sequences. The edit_distance function calculates the minimum number of operations (insertions, deletions, substitutions) required to transform the output sequence into the target sequence. The sum of these distances for all sequences divided by the total length of all target sequences (as seen in the evaluate function with average_EDR calculation) provides the EDR. The formula is given below:

$$\text{EDR} = \frac{\sum_{\text{methods}} \text{EditDistance}(s_R, s_T)}{\sum_{\text{methods}} \text{length}(s_T)}$$

EditDistance(s_R, s_T) represents the edit distance between the reference code snippet s_R and the translated code snippet s_T, indicating the minimum number of token edits (insertions or deletions) needed to transform s_T into s_R. The denominator, $\sum_{\text{methods}} \text{length}(s_T)$, aggregates the lengths of all translated code snippets, thus normalizing the edit distance by the total length of the translation output.

6.4 Command Lines for Training Model

```
cd tree2tree
```

Create a Python virtual machine if you do not have one:

```
python3 -m venv virtual_environment_name
```

Activate your Python virtual machine:

For macOS:

```
source virtual_environment_name/bin/activate
```

For Windows and Linux:

```
virtual_environment_name\Scripts\activate //in CMD
```

Install the packages with the desired version as recorded in the `'requirements.txt'` file:

```
pip3 install -r requirements.txt
```

or

```
pip install -r requirements.txt
```

To train the model in the Virtual Environment:

For Windows:

```
python translate.py
--network tree2tree
--train_dir ..\model_ckpts\tree2tree\
--input_format tree
--output_format tree
--num_epochs 100
--batch_size 5
```

```
evaluate(model, test_set, source_vocab, target_vocab, source_vocab_list, target_vocab_list):
"""
Evaluates the model on a test set and computes various metrics.

param model: The Tree2TreeModel instance to evaluate.
param test_set: The dataset to evaluate the model on.
param source_vocab: Dictionary mapping source vocabulary tokens to indices.
param target_vocab: Dictionary mapping target vocabulary tokens to indices.
param source_vocab_list: List of tokens in the source vocabulary.
param target_vocab_list: List of tokens in the target vocabulary.

Outputs evaluation metrics including loss, token accuracy, and program accuracy.
"""
test_loss = 0
acc_tokens = 0
tot_tokens = 0
tot_output_tokens = 0
acc_programs = 0
tot_programs = len(test_set)
res = []
average_EDR = 0

for idx in xrange(0, len(test_set), args.batch_size):
    # here so-called "decoder_inputs" is not actually the input of the decoder
    # it is the ground truth TreeManagers.
    # The data it represents will be called 'target_tree' in the model.forward()
    # and in the model.forward(),
    # decoder_input is the prediction_tree.root, which is t_t, the prediction result - a token id
    encoder_inputs, decoder_inputs = model.get_batch(test_set, start_idx=idx)
    eval_loss, raw_outputs = step_tree2tree(model, encoder_inputs, decoder_inputs, feed_previous=True)
    test_loss += len(encoder_inputs) * eval_loss
    for i in xrange(len(encoder_inputs)):
        if idx + i >= len(test_set):
            break
        current_output = []

        for j in xrange(len(raw_outputs[i])):
            current_output.append(raw_outputs[i][j])

        current_source, current_target, current_source_manager, current_target_manager = test_set[idx + i]

        current_source = data_utils.serialize_tree(current_source)
        res.append((current_source, current_target, current_output))

        tot_tokens += len(current_target)
        tot_output_tokens += len(current_output)
        all_correct = 1
        wrong_tokens = 0
        # Save Translation Result
        return save_translation_result(
            idx, current_output, source_vocab, target_vocab, source_vocab_list, target_vocab_list)
```

Figure 6.7: Evaluate Function in Translate.py

```python
def edit_distance(output, target):
    """
    Computes the edit distance between two sequences (output and target).

    param output: List of tokens representing the model's output.
    param target: List of tokens representing the target sequence.

    return: The edit distance (integer) between the output and target sequences.
    """
    output_len = len(output)
    target_len = len(target)
    OPT = [[0 for i in range(target_len+1)]
           for j in range(output_len + 1)]  # DP table

    # OPT[i][0] = i, assign values to first column
    for i in range(1, output_len+1):
        OPT[i][0] = i
    # OPT[0][j] = j, assign values to first row
    for j in range(1, target_len+1):
        OPT[0][j] = j
    single_insert_cost = 1
    single_delete_cost = 1
    single_align_cost = 1
    for i in range(1, output_len+1):  # row
        for j in range(1, target_len+1):  # column
            delta = single_align_cost if output[i-1] != target[j-1] else 0
            alignment_cost = OPT[i-1][j-1] + delta
            delete_cost = OPT[i-1][j] + single_delete_cost
            insertion_cost = OPT[i][j-1] + single_insert_cost
            OPT[i][j] = min(alignment_cost, delete_cost, insertion_cost)
    return OPT[output_len][target_len]
```

Figure 6.8: EDR function in Translate.py

```
--steps_per_checkpoint 5
--train_data ..\..\parser\data\source_pascal_target_java_train.json
--val_data ..\..\parser\data\source_pascal_target_java_validation.json
```

For Linux:

```
python translate.py
--network tree2tree
--train_dir ../model_ckpts/tree2tree/
--input_format tree
--output_format tree
--num_epochs 100
--batch_size 5
--steps_per_checkpoint 5
--train_data ../../parser/data/source_pascal_target_java_train.json
--val_data ../../parser/data/source_pascal_target_java_validation.json
```

To test the model in the Virtual Environment:

Best Eval Loss Model:

```
python translate.py
--network tree2tree
--test
--load_model ../model_ckpts/tree2tree/best_eval_loss_translate_200.ckpt
--train_data ../../parser/data/source_pascal_target_java_train.json
--test_data ../../parser/data/source_pascal_target_java_test.json
--input_format tree --output_format tree
```

Best Loss Model:

```
python translate.py
--network tree2tree
--test
--load_model ../model_ckpts/tree2tree/best_loss_translate_195.ckpt
--train_data ../../parser/data/source_pascal_target_java_train.json
--test_data ../../parser/data/source_pascal_target_java_test.json
--input_format tree --output_format tree
```

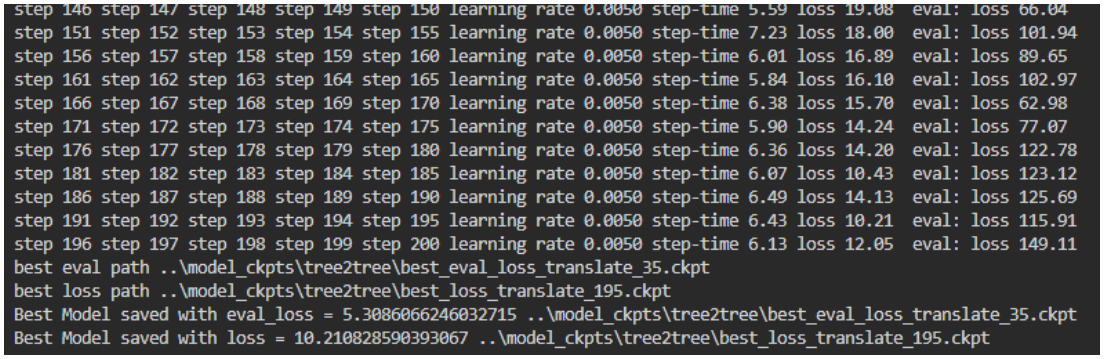


Figure 6.9: Training Screenshot shown from here

6.5 Integrating Frontend and Backend through RESTful API Services

This section discusses the architectural setup and data flow mechanisms implemented between the frontend, backend, and the Tree2Tree model's backend within our application. The connectivity framework is established leveraging HTTP RESTful APIs, utilizing Java Spring Boot for the frontend-backend interactions and Flask for communications between the backend and the Tree2Tree model. The overview of the software engineering part flowchart is shown in figure 6.4.

6.5.1 Frontend-Backend Interaction

The core interface facilitating the communication between the frontend and backend is encapsulated within the `PascalController.java` located in the backend directory (`backend/src/main/java/com/example/demo`). The important function managing this interconnection is `/processPascal`, which is designed to handle POST requests. In a standard setup, Spring Boot designates port 8080 for its operations.

The process flow is as follows:

1. **Extraction**: The method anticipates a JSON payload adhering to the structure defined by the `PascalRequest` class, predominantly containing a `String` attribute named `code`, representing the Pascal program intended for processing. The outcome will be turn out as shown in figure 6.5.

2. **Parsing**: The Pascal code extracted from the request via `request.getCode()` is fed into `parsePascal(code)`, resulting in a Pascal AST (Abstract Syntax Tree).

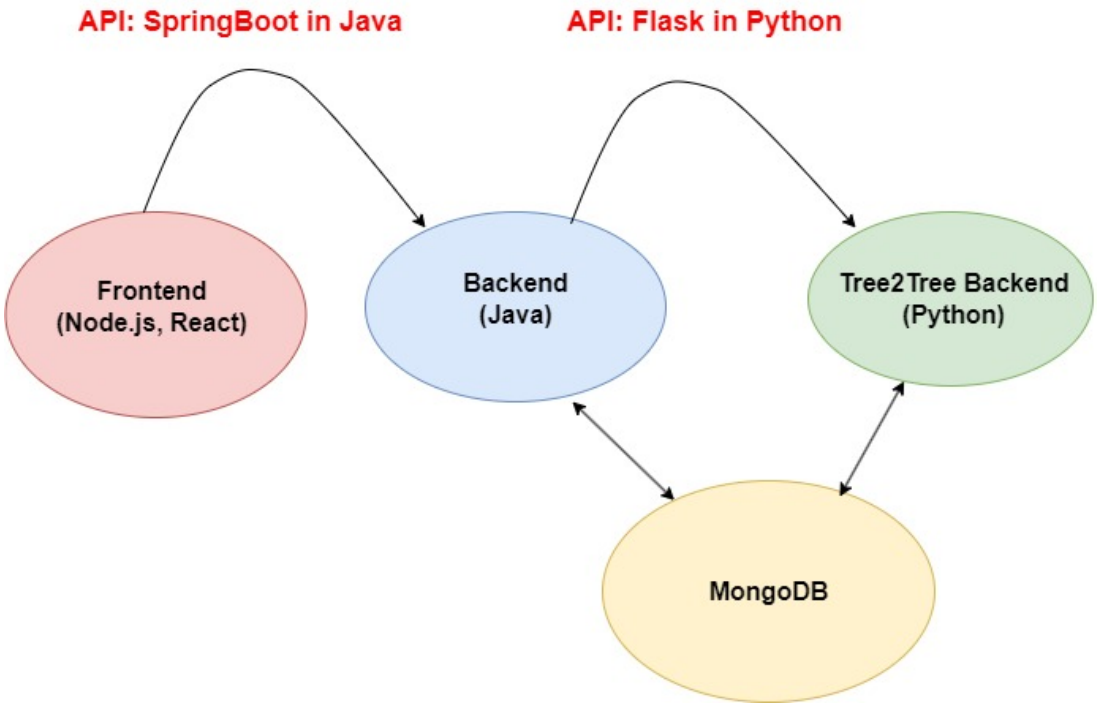


Figure 6.10: The overview flowchart diagram for software api

3. **Validation**: A non-empty Pascal AST signifies the Pascal code's validity. Conversely, an empty AST results in the backend returning an "Invalid Pascal" response, indicating a failure in the validation phase.

4. **Hashing and Database Lookup**: Valid Pascal programs undergo a hashing process. The generated hash is then cross-referenced with the database. If recognized, the corresponding Java program translation stored within the database is retrieved and sent back to the frontend.

5. **Tree2Tree Model Invocation**: Unmatched hashes trigger further processing. The Pascal AST, converted into JSON format, is dispatched to the Tree2Tree model's backend through the Flask endpoint `/getjavaast` at `http://localhost:5000/getjavaast`, initiating the conversion to a Java AST.

6.5.2 Backend-Model Backend Interaction

The conversion from Pascal AST to Java AST, facilitated by the Tree2Tree model's backend developed in Python Flask, forms a crucial part of the workflow designed to enable the automated transformation of Pascal code to its Java equivalent, employing machine learning methodologies for syntax tree transformations.

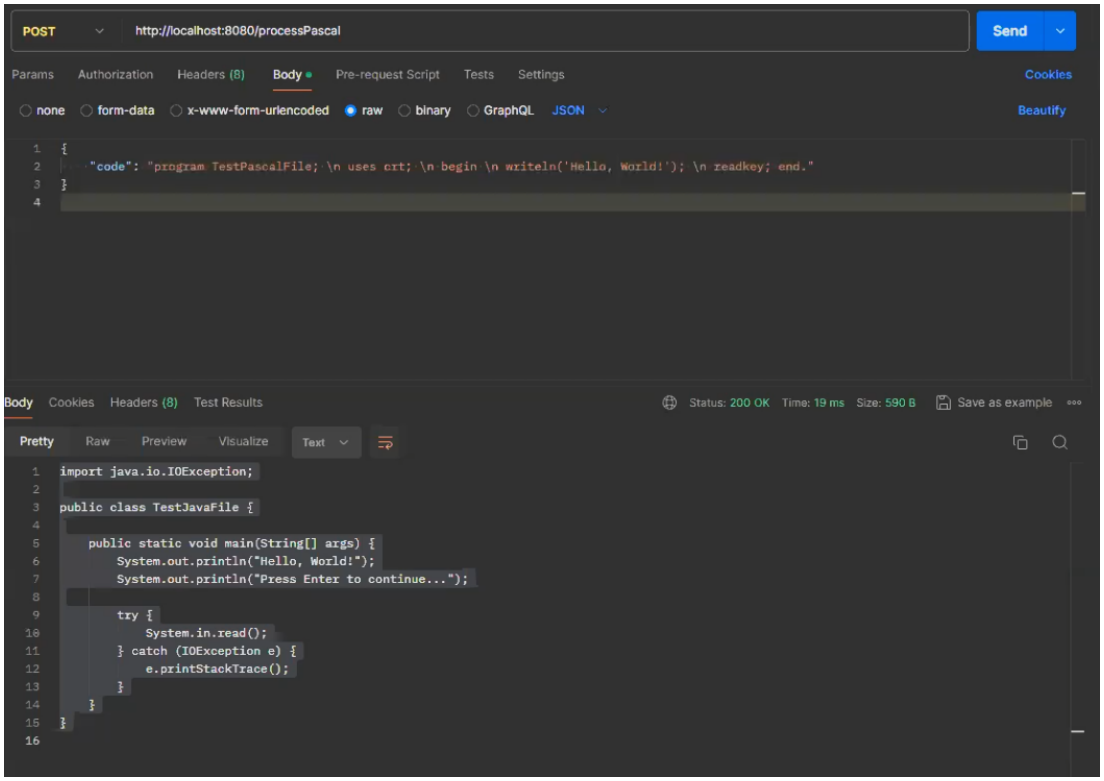


Figure 6.11: Json Pascal Code

6.5.3 Post-Translation Processing

Following the receipt of the Java AST from the model's backend, the subsequent steps involve:

1. **Similarity Assessment**: The Java AST is benchmarked against existing ASTs within the database utilizing Edit Distance Metrics to ascertain the most similar Java program. The code for the comparison of similarity is provided in figure 6.6

2. **Final Response**: The closest approximation, determined by the minimal edit distance, is identified as the translation outcome and relayed back to the frontend.

6.5.4 DataBase-MongoDB

The original translated result will all be stored in the database system as shown in figure 6.10.

6.5.5 Docker & AWS Cloud

The primary objective of leveraging DockerHub in this project is to ensure environmental isolation for the application. Each Docker container can be thought of as a miniature Linux system, providing a segregated runtime environment. This project is structured around five

```java
public CodeData saveCodeData(CodeData codeData) {
    System.out.println(codeData.getPascal());
    String hash = hashGenerator.generateHash(codeData.getPascal());
    codeData.setPascal(hash);
    System.out.println(hash);
    return codeDataRepository.save(codeData);
}
```

Figure 6.12: The code for the comparison of similarity

```java
public class PascalController {

    private final PascalService pascalService;
    private final CodeDataService codeDataService;

    private final MLService msService;

    @Autowired
    public PascalController(PascalService pascalService, CodeDataService codeDataService, MLService msService) {
        this.pascalService = pascalService;
        this.codeDataService = codeDataService;
        this.msService = msService;
    }
```

Figure 6.13: class PascalController

```java
@PostMapping("/processPascal")
public String processPascalCode(@RequestBody PascalRequest request) {
    String code = request.getCode();

    String pascalAST = pascalService.parsePascal(code);

    if(pascalAST == null){
        return "Invalid Pascal";
    }
    System.out.println(pascalAST);

    String pascalASTHash = pascalService.toHash(pascalAST);

    if(codeDataService.isPascalASTHashInDatabase(pascalASTHash)){
        System.out.println("Bingo");
        return codeDataService.getJavaSourceCodeByPascalAstHash(pascalASTHash);
    }else{
        System.out.println("No bingo");
        String javaAst = msService.callGetJavaAstService(pascalAST);
        return codeDataService.findMostSimilarJavaSourceCode(javaAst);
    }
}
```

Figure 6.14: PostMapping route

```python
app = Flask(__name__)
CORS(app)

@app.route('/getjavaast', methods=['POST'])
def getJavaAst():
    pascalAst = request.json['pascalAst']
    print(pascalAst)
    data = build_json_from_parse_tree.prepare_data(pascalAst)
    java_ast = translate.main(data)

    return str(java_ast)
```

Figure 6.15: getjavaast route for connecting backend and machine learning backend.

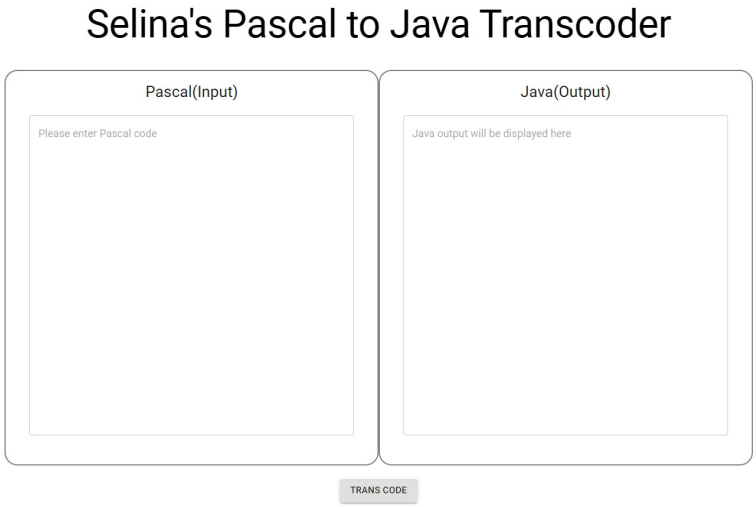


Figure 6.16: Simple User Interface in shown fromhere

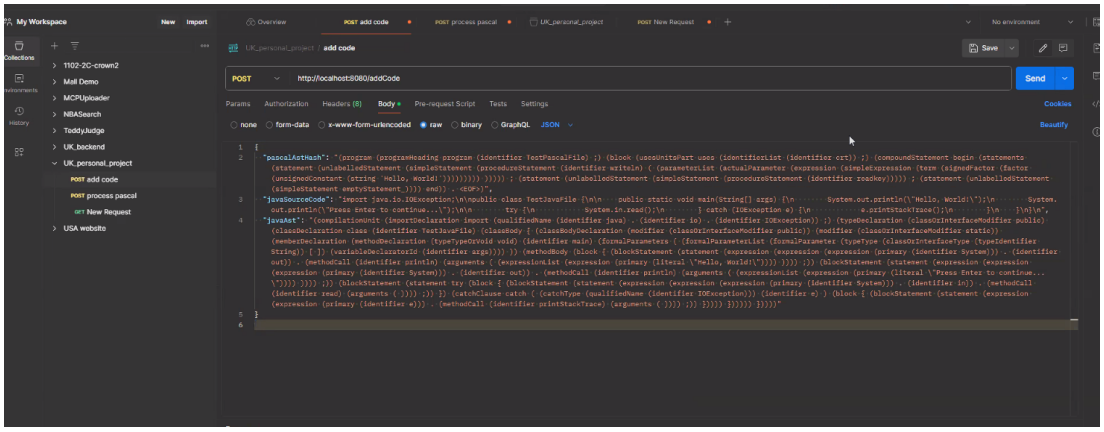


Figure 6.17: DataBase includes Pascal AST hash, Java source code, and Java AST.

core components: the frontend, backend, Tree2Tree backend, the database system, and the training backend. Accordingly, it will utilize four distinct Docker images hosted on DockerHub to manage these components efficiently.

Docker Compose plays an important role in unifying these four essential services on a local device, streamlining deployment by using fully built Docker images. This seamless integration facilitates the combination of the servers into a cohesive unit. Subsequently, this consolidated server infrastructure will be deployed to Amazon Web Services (AWS). This deployment step is instrumental in generating a publicly accessible URL, thereby making the project's website available to users worldwide. This approach underscores the project's commitment to robustness, scalability, and accessibility.

6.5.6 Launching the Software Using Docker

1. Follow these steps to seamlessly launch the software on any device using Docker:
2. Set Up Docker: Begin by installing Docker on your device and log in to your Docker account.
3. Download Project Source Code: Acquire the project's source code by downloading the provided zip file.
4. Initialize the Software: Navigate to the project's directory in your terminal and execute the command 'docker-compose up' to start the software.
5. Access the Software: Simply open your web browser and visit 'localhost:3000' to interact with the software on your device.

6.5.7 Interfacing with the Software through Postman

This section guides you through utilizing Postman to interact with the software's API, demonstrating how to send a request to the server. The example provided illustrates how to add Pascal and Java code snippets, along with their AST representations, to the system.

- Endpoint: `http://localhost:8080/addCode`
- HTTP Method: POST
- Request Body: The body of the request should be structured as a JSON object containing three key components: 'pascalAstHash', 'javaSourceCode', and 'javaAst'. Each key is associated with the corresponding data in string format.

```yaml
version: '3.8'

services:
  mongodb:
    image: selina0917/ptoj_trans_database
    volumes:
      - mongodb_data:/data/db
    networks:
      - my_network

  ast_backend:
    image: selina0917/ptoj_trans_ast_backend
    depends_on:
      - mongodb
    environment:
      SPRING_DATA_MONGODB_URI: mongodb://mongodb:27017/UKPersonalProject
    ports:
      - "8080:8080"
    networks:
      - my_network

  ml_backend:
    image: selina0917/ptoj_trans_ml_backend
    ports:
      - "5000:5000"
    networks:
      - my_network

  uk-frontend:
    image: selina0917/ptoj_trans_frontend # Ensure this matches your frontend image name
    ports:
      - "3000:80" # Map port 80 from the container to port 3000 on the host
    networks:
      - my_network
    depends_on:
      - ast_backend

networks:
  my_network:

volumes:
  mongodb_data:
```

Figure 6.18: Docker Compose File shown from here

```
PS C:\Users\selin\Downloads\uk-personal-project> cd software
PS C:\Users\selin\Downloads\uk-personal-project\software> docker-compose up
[+] Running 40/24
 ✓ ml_backend 12 layers [⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿]      0B/0B      Pulled
 ✓ mongodb 8 layers [⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿]      0B/0B      Pulled
 ✓ ast_backend 7 layers [⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿]      0B/0B      Pulled
 ✓ uk-frontend 9 layers [⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿]      0B/0B      Pulled
```

Figure 6.19: docker-compose up command line to access user interface

Figure 6.20: an example payload for the request

Figure 6.19 is an example payload for the request:

Sending the Request:

1. Open Postman and select the ‘POST‘ method from the dropdown menu.
2. Enter the endpoint URL ‘http://localhost:8080/addCode‘ into the request URL field.
3. Navigate to the ‘Body‘ tab, select the ‘raw‘ option, and ensure JSON is selected as the format.
4. Copy the JSON payload provided above and paste it into the body of the request.
5. Hit the ‘Send‘ button to dispatch the request to the server.

Upon successful request submission, the server processes the Pascal and Java code snippets, along with their AST representations, and adds them to the system. This procedure allows for efficient testing and integration of code snippets through the API, facilitating seamless interaction between the frontend, backend, and the Tree2Tree machine learning model backend.

6.6 Deployment with AWS

This flow outlines a pathway to leveraging AWS services for deploying a scalable and manageable application. The tools which use from AWS are mainly App Runner for application hosting and DocumentDB for database services.

6.6.1 Preliminary Steps

1. Prepare the Application: Ensure your application, including the frontend, backend, and any associated services like Tree2Tree model processing, is containerized. Docker is typically used for creating these containers.

2. Dockerize Your Application: Create Dockerfiles for each part of your application (frontend, backend, Tree2Tree backend, and DataBase). Define the environment, dependencies, and execution commands needed for each container.

3. Push Docker Images to Amazon ECR (Elastic Container Registry): After testing the Docker containers locally, push them to Amazon ECR. Create a repository for each container and use the AWS CLI or AWS Management Console to push your Docker images to these repositories.

6.6.2 Deploying with App Runner

1. Set Up AWS App Runner: Navigate to AWS App Runner in the AWS Management Console. App Runner simplifies deployment, managing infrastructure, scaling, and security.

2. Configure Service: Create a new service in App Runner. Select the source as "Container registry" and choose Amazon ECR as the provider. Then, select the appropriate ECR repository and image that you want to deploy. Repeat this process for each part of your application, including the frontend, backend, and Tree2Tree backend service.

3. Set Environment Variables: If your application requires environment variables (e.g., database connection details, API keys), configure them in the App Runner service settings.

6.6.3 Deploy and Test

Launch the service. AWS App Runner will handle deployment, load balancing, auto-scaling, and provide you with a URL to access your application. Test the application to ensure it's working as expected.

6.6.4 Setting Up Amazon DocumentDB

1. Create a DocumentDB Cluster: Go to the Amazon DocumentDB section in the AWS Management Console. Create a new cluster, configure its settings (e.g., instance size, number of instances), and initialize it.

2. Configure Security: Set up security groups and IAM roles to control access to your DocumentDB cluster. Ensure that your App Runner services can communicate with the DocumentDB cluster.

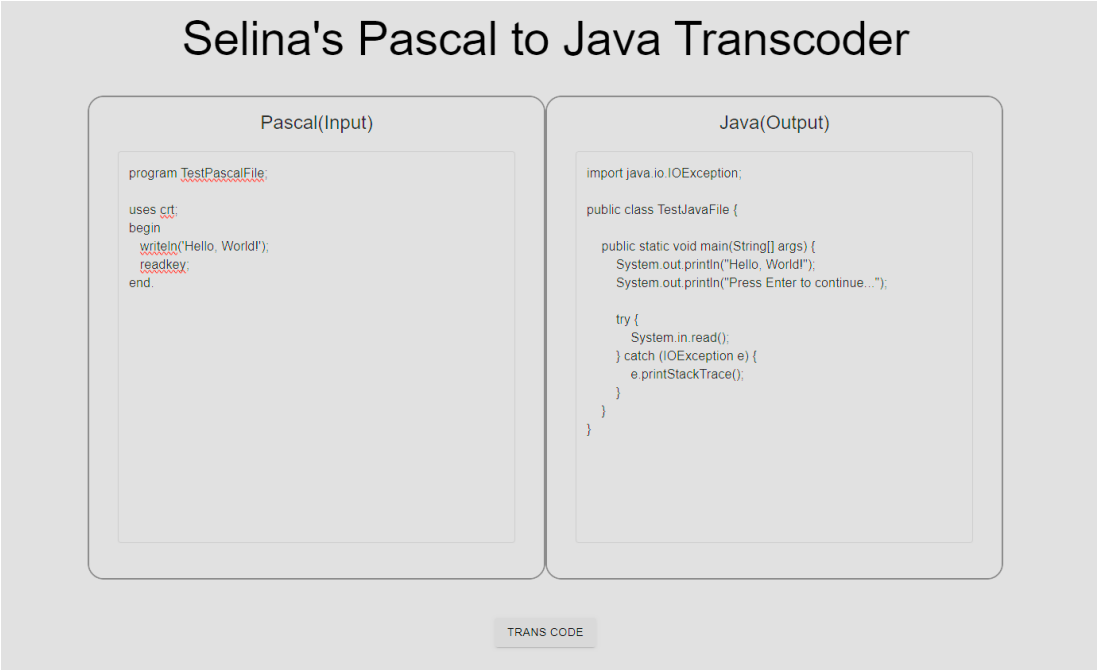


Figure 6.21: The deployed frontend version shown from here

3. Connect Your Application: Modify your application's backend to use the DocumentDB cluster as its database. Use the connection string provided by AWS for your DocumentDB instance.

4. Initialize the Database: If your application requires initial data or schemas to be set up in DocumentDB, execute those scripts or commands to prepare your database.

6.6.5 Final Steps

1. Test the Complete Deployment: With your services running in App Runner and your database set up in DocumentDB, perform thorough testing to ensure everything is working together seamlessly.

2. Monitor and Optimize: Utilize AWS CloudWatch and other monitoring tools to keep an eye on your application's performance and health. Based on insights, you might need to adjust configurations for better performance or cost efficiency.

Lastly, the website can be found in: `https://jpmfexmcd8.us-east1.awsapprunner.com/` (Figure 6.20)

6.7 Problems Encountered During Implementation

6.7.1 Handling Inconsistencies in Newline Character Handling

In the development of the Tree2Tree model for Pascal to Java translation , one significant challenge encountered was related to the process of updating the database using a Python script versus the actual behavior observed on the live website. When posting data to the backend via JSON on the website, newline characters
n were automatically inserted at the end of each line, altering the formatting of the input source code. This discrepancy was not replicated in the database update process executed through the Python script, where newline characters were not automatically added. As a consequence, the absence of consistent newline characters led to mismatches in the generation of hash values for the same source code input. Since the hash values are important for identifying and retrieving corresponding translations from the database, this inconsistency resulted in the inability to find matching hashes, thereby affecting the efficiency and accuracy of the translation service. Addressing this issue required careful consideration of input preprocessing and normalization strategies to ensure uniformity in the treatment of source code, both when directly interacting with the database and through user submissions on the website.

6.7.2 Challenges in Constructing AST JSON for Distinct Languages

Building Abstract Syntax Tree (AST) JSON representations for Pascal and Java posed significant challenges. The reason is due to the inherent differences between the two programming languages. Each language has unique syntactic and semantic structures , requiring a deep understanding of both to accurately parse and represent their respective ASTs. This task demanded a comprehensive analysis of language specifications to ensure the correct interpretation of programming constructs.

6.7.3 Python Version 3.6-3.8

Navigating the complexities of Python versions proved to be a surprisingly time-consuming aspect of implementing the Tree2Tree model by Chen et al. [9]. Compatibility issues across different devices necessitated amount of adjustments to the Python environment. This experience highlighted the often overlooked but critical role that software version control plays in the smooth execution of complex computational models.

Chapter 7

Results/Evaluation

7.1 Metrics Selection Reasoning

- Loss Function: The loss function is fundamental in training neural networks. It provides a quantitative measure of the model's predictions against the actual outcomes. For this project, we employed a loss function that captures the difference between the predicted token sequence and the target sequence. The significance of this metric lies in its direct impact on the gradient descent optimization algorithm—it guides the model to improve by minimizing the loss over successive training iterations. Lower loss values indicate better model performance, making it an essential metric for training efficacy.

- Program Accuracy: Program accuracy refers to the proportion of the translated code that is an exact match to the reference Java code. This metric is pivotal for program translation tasks because it evaluates whether the translated output is correct as a whole. Given the binary nature of program execution (a program either runs correctly or it doesn't), it is critical to measure how often the translated program is entirely correct. High program accuracy suggests that the model has learned to translate code with high fidelity to the intended functionality.

- Token Accuracy: While program accuracy evaluates the translated program as a whole, token accuracy drills down to the granularity of individual tokens. It assesses the percentage of correctly translated tokens in the output. This metric is crucial because it reflects the model's understanding of syntax and semantics at the token level. Token accuracy provides insights into the model's ability to translate each element correctly, which is especially important in programming languages where a single token can significantly alter

program behavior.

- Edit Distance Ratio (EDR): Edit Distance Ratio (EDR) offers a measure of the effort required to transform the model's output into the correct code by counting the number of insertions, deletions, and substitutions. EDR is particularly useful in scenarios where partial credit is warranted—when the translated program is close to the target but not perfect. This metric is beneficial in understanding the model's performance in terms of the magnitude of errors and is valuable for indicating the potential post-editing workload. This metric is often used in code migration evaluation tasks in previous papers.[30][29]

7.2 BLEU

BLEU stands for BiLingual Evaluation Understudy. It serves as an automatic evaluation metric for Language Translation and code migration.[35] In the code migration case, this metric quantifies translation quality through a calculated score based on n-gram precision and a penalty for discrepancies in length. Essentially, BLEU compares the n-gram overlap of the machine-generated translation with a set of reference translations, taking into account the number of words and the order in which they appear. It incorporates a brevity penalty to counteract the favoring of overly short translations, ensuring that the total length of the translated output is not disproportionately less than the reference. The BLEU score ranges from 0 to 1, where higher scores signify greater correspondence with the reference translation. Typically, BLEU considers up to 4-grams, providing a balance between the accuracy of individual words and longer phrases within a translated segment. The formula is shown below:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{7.1}$$

$$BP = \begin{cases} 1 & \text{if } c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \leq r \end{cases} \tag{7.2}$$

Where:

1. c is the length of the candidate translation.
2. r is the effective reference corpus length.
3. BP is the brevity penalty.

4. exp is the exponential function.

5. w_n is the weight for each n-gram.

6. p_n is the precision for n-grams.

7. N is typically 4, as BLEU considers up to 4-grams.

7.2.1 BLEU is not suitable for Program Translation

According to the paper "Does BLEU Score Work for Code Migration?" by Ngoc Tran et al., BLEU (BiLingual Evaluation Understudy), BLEU has been proved to be not suitable for Program Translation. [50]

The reasons are shown below:

- Lack of Semantic Accuracy Correlation: BLEU measures phrase-to-phrase lexical translation accuracy, but it has a weak correlation with the semantic correctness of the translated code. It focuses on lexical precision and fails to reflect the essential syntactic and semantic properties of source code, which are crucial for determining the correctness of program translation.[50]

- Semantic Correctness: In code migration, the semantic correctness of the migrated code is more critical than lexical matching. Programs need to maintain their functionality when translated, not just textually resemble the source code. BLEU does not account for functional or semantic equivalence, leading to situations where higher BLEU scores do not necessarily indicate semantically accurate or even compilable code.[50]

- Comparing Translation Models: BLEU scores are also not reliable for comparing the quality of translation between different SMT (Statistical Machine Translation) models. Empirical studies have shown that models with similar BLEU scores can produce functionally different and semantically incorrect code.[50]

- Complexity of Programming Languages: Source code has a well-defined syntax and dependencies, which natural language lacks. BLEU's method of evaluating translations is not suited to capture the complexities of programming languages and can result in high scores for syntactically incorrect code.[50]

7.3 Results

This section demonstrates the peak performance of the Tree2Tree model in translating Pascal to Java. This is achieved after an extensive period of training on the dataset. The training encompasses five test datasets, ten training datasets, and three validation datasets.

7.3.1 Loss

The model has an evaluation loss of 222.47, which suggests that there is still significant error between the model's predictions and the actual desired outputs. Lower loss values are generally better, indicating closer predictions to the target values.

7.3.2 Total Number of Programs

The model has been evaluated on 5 different programs. While this number helps establish the model's performance, it's quite a small sample size for generalizing the model's capabilities.

7.3.3 Total Number of Matched (Correct) Tokens

There were 922 tokens in the model's output that matched the expected output. This is out of a total of 1281 tokens that the model output, indicating the raw number of correct tokens.

7.3.4 Total Number of Target (Ground Truth) Tokens

There were 1920 tokens in the reference or target programs. This provides a sense of the overall size of the task the model was trying to accomplish.

7.3.5 Program Accuracy&Token Accuracy

The model's token accuracy rate at 48% is a moderate indication that nearly half of the individual code components were correctly translated. However, the 0% program accuracy sharply contrasts this figure. This reveals that despite some correct tokens, the model fails to compile these into a valid program. This stark deficiency suggests that while certain elements are recognized, the model lacks a deeper understanding necessary to reconstruct them into the functional structure of the source language. No program matched its reference exactly, showing that more work is needed to improve the model's understanding of programming concepts as a whole. (Refer to Table 7.1 for more details.)

```
eval: loss 222.47
eval: total number of programs: 5
eval: total number of matched(correct) token: 922
eval: total number of output(translated) token: 1281
eval: total number of target(ground truth) token: 1920
eval: accuracy of tokens 0.48
eval: accuracy of programs 0.00
eval: average EDR of programs 0.57
```

Figure 7.1: Final Best Results' Screenshot

7.3.6 Edit Token Distance Ratio (EDR)

The average Edit Distance Ratio (EDR) stands at 0.57, indicating that over half the tokens in each program output require editing to achieve an exact match with the target. This significant EDR value demonstrates the model's translations are not only frequently incorrect but also differ substantially from the expected outcomes. An EDR over 0 suggests the model's outputs are not trivially misaligned and require considerable modifications, emphasizing the necessity for substantial improvements in the model's translation processes and its ability to predict more accurate tokens. (For more detailed analysis, see Figure 7.1.)

Num of Programs	**Num of Correct Tokens**	**Num of Translated Outputs**	**Program Accuracy**	**Token Accuracy**	**EDR**
5	922 tokens	1281 tokens	48%	0%	0.57

Table 7.1: Best Test Results

7.4 Translated AST Data Analysis

Due to the word limitation, it is not possible to analyse all of the AST output data. Therefore, in this project we will only provide one example for evaluation. The expected AST is shown in figure7.3 and the result AST is shown in figure 7.4.

7.4.1 Similarities Between the Expected AST & Output AST

- Structural Foundation: Both ASTs follow the compilationUnit structure that begins with class declarations, indicating that the model has learned the basic structure of a Java program.

- Use of Modifiers: Both trees include class-level modifiers (public, static) and method declarations (void main), suggesting that the trained model captures essential Java syntax for class and method definitions.

7.4.2 Difference Between the Expected AST & Output AST

- Variable Declarations and Types:The expected AST correctly declares three variables of different types (int, double, and String) and assigns them values.The Output AST seems to attempt variable declarations but shows inconsistencies and inaccuracies in the variable names (myInt, myBoolean) and types. Notably, it seems to confuse type declarations, evidenced by an attempt to declare a boolean but labeled as an int (primitiveType(int) for myBoolean), and lacks completeness in the declaration structure.

- Complexity and Completeness:The expected AST exhibits a complete and complex structure, incorporating a method call that prints out a concatenated message involving arithmetic operations and variable references.The output AST lacks this complexity, missing the critical statement block that demonstrates understanding of arithmetic operations, string concatenation, and method calls for output.

- Literal Values and Expressions:The accurate AST includes explicit assignments with literal values for each variable and combines these in a subsequent expression.The output AST fails to replicate this aspect accurately. While it attempts to assign a literal value to myInt, it diverges significantly in content and context for other variables and operations.

- Syntactic and Semantic Accuracy:The correct AST shows both syntactic and semantic accuracy, adhering closely to Java programming conventions.The output AST demonstrates syntactic attempts but falls short in semantic accuracy, particularly in variable types and operations, reflecting a superficial understanding of the language's constructs.

7.4.3 AST Analysis

The trained model seems to understand the foundational structure of a Java program. However, it struggles with variable declarations, types, and operations complexity. These are crucial for accurately translating or generating code. The inaccuracies and incompleteness in the output AST (figure7.4) suggest that the model may require further training, a more extensive dataset, or adjustments in its approach to better capture the depth and breadth of programming language constructs.

```java
public class TestJavaFile {
    public static void main(String[] args) {
        int integerVar = 10;
        double floatVar = 20.5;
        String stringVar = "Hi!";

        System.out.println(stringVar + " The sum is: " + (integerVar + floatVar));
    }
}
```

Figure 7.2: The original Java Code for testing

```
(compilationUnit (typeDeclaration (classOrInterfaceModifier public) (classDeclaration class
(identifier TestJavaFile) (classBody { (classBodyDeclaration (modifier
(classOrInterfaceModifier public)) (modifier (classOrInterfaceModifier static))
(memberDeclaration (methodDeclaration (typeTypeOrVoid void) (identifier main)
(formalParameters ( (formalParameterList (formalParameter (typeType
(classOrInterfaceType (typeIdentifier String)) [ ]) (variableDeclaratorId (identifier args)))) ))
(methodBody (block { (blockStatement (localVariableDeclaration (typeType (primitiveType
int)) (variableDeclarators (variableDeclarator (variableDeclaratorId (identifier integerVar)) =
(variableInitializer (expression (primary (literal (integerLiteral 10)))))))) ;) (blockStatement
(localVariableDeclaration (typeType (primitiveType double)) (variableDeclarators
(variableDeclarator (variableDeclaratorId (identifier floatVar)) = (variableInitializer
(expression (primary (literal (floatLiteral 20.5)))))))) ;) (blockStatement
(localVariableDeclaration (typeType (classOrInterfaceType (typeIdentifier String)))
(variableDeclarators (variableDeclarator (variableDeclaratorId (identifier stringVar)) =
(variableInitializer (expression (primary (literal "Hi!"))))))) ;) (blockStatement (statement
(expression (expression (expression (primary (identifier System))) . (identifier out)) .
(methodCall (identifier println) (arguments ( (expressionList (expression (expression
(expression (primary (identifier stringVar))) + (expression (primary (literal " The sum is: "))))) +
(expression (primary ( (expression (expression (primary (identifier integerVar))) +
(expression (primary (identifier floatVar)))) ))))) )))) ;)) }))))) }))))
```

Figure 7.3: The expected AST we should get

(compilationUnit(typeDeclaration(classOrInterfaceModifier(public))(classDeclaration(class)(id
entifier(TestJavaFile))(classBody({)(classBodyDeclaration(modifier(classOrInterfaceModifier(
public)))(modifier(classOrInterfaceModifier(static)))(memberDeclaration(methodDeclaration(t
ypeTypeOrVoid(void))(identifier(main))(formalParameters({'root': 'formalParameterList',
'children': [{'root': 'formalParameter', 'children': [{'root': 'typeType', 'children': [{'root':
'classOrInterfaceType', 'children': [{'root': 'typeIdentifier', 'children': [{'root': 'String', 'children':
[]}]}]}, {'root': '[', 'children': []}, {'root': ']', 'children': []}]}, {'root': 'variableDeclaratorId', 'children':
[{'root': 'identifier', 'children': [{'root': 'args', 'children':
[]}]}]}]}]}))(methodBody(block({)(blockStatement(localVariableDeclaration(typeType(primitiveT
ype(int)))(variableDeclarators(variableDeclarator(variableDeclaratorId(identifier(myInt)))(=)(v
ariableInitializer(expression(primary(literal(integerLiteral(10)))))))))(;))(blockStatement(statem
ent(expression(expression(expression(primary(identifier)))(.)(identifier))(=)(expression))(;)))(b
lockStatement(localVariableDeclaration(typeType(primitiveType(int)))(variableDeclarators(va
riableDeclarator(variableDeclaratorId(identifier(myBoolean)))(=)(variableInitializer(expression
(primary(literal(true)))))))))(;))(blockStatement(localVariableDeclaration(typeType(primitiveTyp
e(int)))(variableDeclarators(variableDeclarator(variableDeclaratorId))))(;))(blockStatement(st
atement(switch)))(blockStatement(statement))(blockStatement))))))(}))))))

Figure 7.4: The trained result

7.5 Overall Performance Evaluation

The overall performance evaluation of the Tree2Tree model for program translation from Pascal to Java indicates that there are significant areas for improvement. The program accuracy is reported at 0%, a clear indication that the model did not produce any fully correct program translations within the test set. While the token accuracy is 48%, suggesting that almost half of the tokens were correctly identified, this did not translate into correct programs.

The average Edit Distance Ratio (EDR) of 0.57 reveals that more than half of the tokens in each output would need to be edited to match the target code accurately. This high EDR is a quantitative reflection of the gap between the model's output and the expected code translations.

Such an evaluation points toward a need for deeper analysis into the model's learning mechanisms and its interpretative capabilities of source code. It is obvious that while individual tokens may be recognized, the model struggles to piece them together into the complex structures that form a valid program.

Future iterations of the model could benefit from a more expansive and varied dataset to improve its learning scope. Enhancements in the model's architecture to better handle the intricacies of code translation could also contribute to a better performance. Also, making careful tweaks to how the model works and studying its results closely could help it get better at translating and make fewer mistakes that need fixing. The Future Work chapter of this

report details potential directions for advancing this research.

7.6 Possible Reasons for Poor Performance

- Total Number of Programs: The dataset size is a relatively small sample size and may not provide comprehensive insights into the model's performance across a wide range of scenarios. In future work, it would be beneficial to test the model on a larger dataset for more robust evaluation.

- Big Difference Between Pascal and Java: The model may struggle with accurately capturing and translating the underlying logic and structure, especially for complex constructs. Also, Pascal and Java have specific idioms and patterns that are not always directly translatable to each other. A model may struggle if it cannot understand the context or the intended functionality behind the code snippets. The detailed difference listing is already stated in the previous page of this chapter.

- ANTLR (Another Tool for Language Recognition): If ANTLR generates incorrect or incomplete ASTs due to parsing errors, the model will be trained on faulty data. Moreover, ANTLR4 sometimes struggles with complex constructs. This could lead to erroneous ASTs. The possible solution could be using other generator tools for checking accuracy of the generated ASTs.

- Overfitting: The datasets might be too small and not diverse enough. The model might overfit to the training examples without learning the not provided data. Cross validation and increment of datasets could solve this issue.

7.7 Project Evaluation

7.7.1 Background and Literature

This work presents a comprehensive survey and critical assessment of various neural network models, with particular attention to the Tree2Tree model as a promising approach for translating legacy programming languages. The emphasis on the Tree2Tree model was established at the outset of this report, highlighting its potential advantages over other neural network architectures for this specific task. This preference stems from the Tree2Tree model's ability

to process and transform abstract syntax trees (ASTs), offering a more syntactically sensitive approach to program translation.

Additionally, we introduce essential software system tools, such as MongoDB or Docker employed in the software development phase of this project, providing an overview of their functionality and relevance to our work. These tools form the backbone of the development process, enabling the practical implementation of the theoretical concepts explored. Again, since this project focus on the machine learning part more, this project will not be investigate in sufficient details in the software system part.

However, it's important to acknowledge the constraints of time which limited the extent of our research. Consequently, there may be inaccuracies or oversights within the theoretical discourse presented. Despite these limitations, the background information and literature review compiled in this report reflect the depth of understanding and knowledge acquired through extensive literature review and research. This foundation of knowledge underpins the experimental and developmental aspects of the project, guiding the selection of methodologies and tools for the execution of the translation model.

7.7.2 Design& Implementation

This project takes a substantial step towards resolving the challenges encountered in the adaptation and application of the original Tree2Tree system for training on specific datasets. It notably tackles the critical issue of translating legacy programming languages into modern equivalents, achieving considerable success in this domain. The core of the project is the implementation of the Tree2Tree model, initially introduced by Chen et al., with a comprehensive detailing of its process and mechanisms presented within the practical segments of this work.

In terms of software engineering, the project exemplifies a methodical and thoroughly considered approach. It demonstrates a profound comprehension of both the theoretical underpinnings and the practical methodologies involved. This is evidenced by the inclusion of a comprehensive flowchart that outlines the entire software system's structure. Moreover, the project delves into the intricacies of API server implementation and the seamless integration between the frontend, backend, and machine learning backend components. Each of these elements is articulated with clarity and precision, reflecting a well-rounded understanding of the software development life cycle and the innovative solutions employed to bridge various components of the project.

7.7.3 Evaluation & Results

The evaluation segment of this project delves deeply into the analysis of metric evaluations, confronting the challenge posed by the scarcity of comprehensive Pascal datasets. This limitation, compounded by the project's time constraints, precluded the possibility of assembling a dataset that fully encapsulates the diverse structures characteristic of Pascal. The preliminary results, which highlight a performance that falls short of expectations, underscore the critical importance of leveraging larger datasets to mitigate the risk of overfitting and enhance model accuracy.

This evaluation explores other factors that might have contributed to the less-than-optimal performance, offering a thorough analysis within the evaluation section. It brings to light the intricate balance required between sufficient data representation and model training efficacy, particularly in the context of machine learning endeavors focused on program translation.

While the primary emphasis of this project was placed on the machine learning aspects, the software system's evaluation has not been extensively covered. Nevertheless, the section dedicated to future work presents an array of insightful suggestions and potential avenues for further development within the software engineering domain. These proposed enhancements and extensions reflect a forward-thinking approach to overcoming the current limitations and harnessing future opportunities to refine and advance the project's scope and impact.

7.7.4 Achievements

This project marks a significant achievement in program translation by successfully using the Tree2Tree model to translate legacy Pascal code into Java. To the best of my knowledge, this innovative application of the Tree2Tree model for Pascal to Java translation is unprecedented in existing research literature, marking an innovative step in the field.

Furthermore, the project undertook the ambitious task of curating a dataset comprising pairs of Pascal and Java code. This project not only provided the foundational data necessary for training the model but also contributes a valuable resource to the research community, facilitating future studies and advancements in program translation.

In addition to these achievements, the project stands out for its integration of multiple programming languages—specifically Java and Python—to architect a comprehensive software system. This interdisciplinary approach not only demonstrates technical versatility but also underscores the project's commitment to creating a robust and functional system capable of bridging the gap between legacy and modern programming languages.

Chapter 8

Conclusion

To sum up, this project embarked on the ambitious task of translating code from Pascal to Java using a novel Tree2Tree LSTM Encoder-Decoder model, enhanced with a parent attention feeding mechanism. Despite facing challenges and achieving varying levels of performance, the project successfully demonstrated the potential of neural network architectures in program translation. The development and deployment of a comprehensive software system and a dedicated website for Pascal to Java translation mark significant achievements. This project generally demonstrates the practical application of theoretical concepts in neural network program translation task. Moreover, the project utilized advanced tools such as PyTorch and ANTLR4, delving into the evaluation metrics of program translation, including BLEU, program accuracy, among others. This project not only contributed valuable insights into the neural network domain but also served as a solid foundation for further research and exploration.

Throughout this project, my journey was marked by significant personal and professional growth, spanning a wide range of technical skills and knowledge areas. I gained a deeper understanding of neural network concepts, delving into the intricacies of encoder-decoder architectures, Long Short-Term Memory (LSTM) networks, Recurrent Neural Networks (RNN), and various neural network models. This theoretical knowledge was put into practice as I adapted and applied the original Tree2Tree system to address the unique challenge of translating code from Pascal to Java. Furthermore, the problem solving skills has improved by completing this project.

The practical aspects of this project provided a valuable hands-on experience in software development, particularly in the use of two distinct APIs, Spring Boot and Flask, to create a robust software system. This experience was further enriched by implementing Docker and AWS

for the final deployment, which introduced me to the complexities of containerization and cloud services. Additionally, the use of MongoDB as a database solution deepened my understanding of database management and integration within a full-stack development environment.

This project also served as a platform for refining my coding skills across various programming languages, including Python, Java, JavaScript, and React. The inclusion of Docker, AWS, and MongoDB in the final implementation not only broadened my technical skill set but also provided insights into modern software development practices and tools.

As my university's course dissertation, this work not only signifies the culmination of my academic journey but also a stepping stone towards future attempt in the field of machine learning and program translation. The lessons learned, skills acquired, and knowledge gained throughout this project will undoubtedly serve as invaluable assets in my continued interest of excellence in computer science. Moving forward, it is inspiring to further explore the boundaries of neural networks in program translation, aiming to contribute to the advancement of this fascinating domain.

Chapter 9

Legal, Social, Ethical and Professional Issues

I am cognizant of and duly adhere to the regulations pertinent to my project domain, as well as the Code of Conduct and Code of Good Practice as promulgated by the British Computer Society. This project has been meticulously evaluated to ensure it is devoid of social and ethical issues. Furthermore, it is underscored by an academic discipline wherein all ideas, theories, and algorithms not originally conceived by me are duly acknowledged and cited in both the report and the source code.

Regarding the dataset employed in this project, I hereby affirm that the code has been authentically developed by myself. The Tree2Tree Model code has been cited in the source code files. The structure of the dataset draws inspiration from the architectural paradigms featured on *GeeksforGeeks* and *TutorialsPoint*, adapted with original contributions to suit the specific needs of this study.

Lastly, it is pertinent to mention that the scope of this project does not necessitate the retention of users' personal information, thereby obviating concerns related to data privacy and protection. This approach further solidifies the project's adherence to ethical standards and respects the privacy of individuals, in alignment with best practices in research and development.

Chapter 10

FutureWork

10.1 Dataset Enrichment

The poor performance observed in this project's outcomes can be significantly attributed to the insufficiency of datasets available for training. In machine learning, it is generally accepted that a substantial dataset—at a minimum of one thousand[51] instances—is crucial for the effective training of models. This benchmark ensures a diverse range of scenarios for the model to learn from, enhancing its accuracy and efficiency. However, sourcing datasets in Pascal, given its dwindling presence in contemporary programming practices, considers as a challenge in this project. The scarcity of Pascal datasets leads to a lack of comprehensive pairs of source and target codes, which further compounds the difficulty of training a proficient model. Future work ideas might be helpful for the improvement of the datasets:

1. Data Augmentation Techniques: Utilizing code-specific data augmentation techniques can significantly expand the training dataset. Methods including the modification of code snippets, renaming of variables, and adjustments to logical structures—while maintaining the original functionality—add essential diversity and intricacy to the dataset. Such strategies allow the model to train on a wider array of examples, improving its capacity to adapt to varied coding patterns and conventions. By doing so, it increase the amount of datasets.

2. Cross-Language Transfer Learning[21]: Using transfer learning techniques from models trained on more widely used programming languages could be beneficial. Transfer learning takes what a model has learned from one task (like understanding Python or Java code) and applies it to another task (translating Pascal to Java). Since Pascal datasets

are hard to come by, the model can learn from more common programming languages instead. This is possible because many programming languages share similar structures and rules. So, by learning from a broader range of code examples, the model can become better at translating between Pascal and Java, making it more accurate and reliable in its translations.

3. Collaborative Dataset Collection: Since there is no collaboration for this project, the resource will not be enough to create a suitable amount of datasets for training and testing.Engaging in collaborative efforts to collect datasets, possibly through partnerships with academic institutions or contributions from the programming community, offers a practical solution to dataset scarcity. This collective approach can gain more knowledge and resources available within these communities, leading to the discovery or creation of diverse and comprehensive Pascal datasets. Furthermore, promoting a culture of sharing and collaboration could provide a continuous stream of data, ensuring the model's iterative improvement and adaptation to evolving coding practices.

10.2 Alternative Model

10.2.1 Seq2Seq for Comparison

This project primarily focuses on the Tree2Tree model, constrained partly by the limitations of time, which precluded a thorough exploration of the Seq2Seq model within the same context. Nonetheless, the comparative study by Chen et al.[9], which trained four distinct machine learning models for CoffeeScript and JavaScript translation, underscores the value of a multifaceted approach in discerning the performance across different models for program translation. It is possible to do in the comparative study approach for neural networks in program translation for training popular models in program translation analysis. Given the noted efficacy and popularity of the Seq2Seq model[46] in facilitating program translation tasks, a compelling avenue for future research emerges. It involves integrating and comparing the Seq2Seq model alongside the Tree2Tree model. This addition promises to enrich our understanding by offering a comprehensive comparative analysis, thereby paving the way for more informed decisions regarding the optimal model selection for specific program translation challenges.

10.2.2 Statistical Machine Translation

The field of software translation has been greatly influenced by the use of machine translation (SMT) methods as shown in studies that apply these techniques to handle the challenges of code migration[30][29][31]. These strategies, often utilizing the grammatical aspects found in programming languages provide a comprehension of code organization making it easier to translate between different languages. The utilization of phrase based SMT models and the innovative application of Word2Vec representations, for API matching highlight the effectiveness of approaches in capturing the nuances of language libraries and frameworks. Nevertheless with the rise of learning there is potential for an era in software translation that focuses on models capable of grasping more profound and abstract code representations. Future research could explore a combined approach that merges the benefits of SMTs understanding of language structure with learning models advanced representation learning skills. This exploration may reveal techniques for software translation that combine strengths, with modern capabilities potentially resulting in more precise translations tailored to specific programming languages and their associated libraries. Such advancements could streamline code migration processes while broadening the reach of software translation tools to encompass an array of programming languages and scenarios.

10.3 Model Optimisation

10.3.1 Metrics

In this project, we primarily relied on three key metrics, largely inspired by the work of Chen et al. [9]. However, there's considerable potential to broaden our evaluative horizon with additional, nuanced metrics that delve deeper into the structural and semantic similarities between source and target programming languages. One intriguing avenue is to assess the similarity of Abstract Syntax Trees (ASTs) between two languages by examining the number of subnodes each node possesses (referred to as the node's arity). A comparative analysis based on corresponding nodes' arity could yield insights into structural similarities or disparities, offering a novel layer of evaluation for program translation tasks.

Furthermore, expanding the evaluation framework to include Semantic Metrics presents another promising direction. Such metrics, which focus on the meaning and functionality conveyed by the code beyond its syntactic structure, could enhance the thoroughness and depth of our analysis. This approach aligns with emerging research trends, where the semantic integrity

of translated code is a focal point. By integrating these advanced metrics—Arity Metrics for structural evaluation and Semantic Metrics for functional fidelity—we can significantly enrich the evaluative landscape, driving forward the field of neural network-based program translation with more comprehensive and insightful analyses.

10.3.2 Artificial Evaluation

For future work, implementing a comprehensive artificial evaluation framework presents an essential step towards validating the effectiveness and accuracy of the Tree2Tree Model in translating AST (Abstract Syntax Trees) from Pascal to Java. Engaging professional software engineers or experienced programmers in the evaluation process can provide insightful feedback on the correctness and practicality of the translated ASTs.

In addition to expert reviews, future efforts could also incorporate automated testing strategies that utilize unit tests or integration tests designed around the expected outputs of the translation process. These automated tests can quickly verify whether the translated Java code behaves as intended when executed, providing an objective measure of translation accuracy.

Moreover, crowd-sourcing platforms could be another viable approach to gather evaluations from a broader audience, including both experts and community members. Participants could be asked to rate the quality of translations or even correct any inaccuracies, with their feedback used to further refine and improve the Tree2Tree model.

Lastly, incorporating these artificial evaluation results into a continuous learning loop can ensure that the Tree2Tree model iteratively improves over time. By analyzing the feedback and identified inaccuracies, the model can be adjusted, trained with augmented datasets to lead to more accurate and reliable translations.

10.4 Expanding the Scope to Additional Legacy Programming Languages - COBOL

Exploring the translation capabilities for a wider array of structurally similar legacy programming languages presents an exciting opportunity for future research. By extending the scope of this project to include more languages with legacy status, we can uncover valuable insights into the versatility and adaptability of the Tree2Tree model across various programming paradigms. For instance, COBOL (Common Business-Oriented Language) represents a prime candidate for future exploration. COBOL is still using in some banking, finance industries,

or governments.[42]Therefore, employing the Tree2Tree model to translate COBOL code could offer a significant contribution to modernizing legacy systems that are still crucial in various sectors. This approach not only aids in bridging the gap between old and new technology platforms but also has the potential to enhance the efficiency, maintainability, and functionality of critical systems. Consequently, it serves as a meaningful venture towards supporting and advancing societal infrastructure through technology.

10.5 Software Engineering Future Work

Although this project mainly focus on the machine learning part, there is still some improvement could be done for the website development part.

Security Improvements: Implementing comprehensive security practices for containerized applications, including regular updates to Docker images and employing AWS security tools like AWS Shield and AWS WAF, can enhance the protection of the web application against common web exploits and DDoS attacks.

User Experience and Accessibility: Future developments could include building a more interactive and user-friendly web interface, hosted on AWS Amplify for enhanced performance and scalability. Additionally, implementing accessibility features to make the web application more inclusive.

User Interface Feedback: This project does not evaluate the user interface. It will be better get feedback from users for improvements and evaluation.

References

[1] Amazon Web Services. Free cloud computing services - aws free tier. https://aws.amazon.com/free/?gclid=CjwKCAjwwr6wBhBcEiwAfMEQs53MZbuwK2Wf2fGrNUwy7iXCkrV03utzj6VwMG08ohnAl_4YZY94IhoCWLsQAvD_BwE&trk=d5254134-67ca-4a35-91cc-77868c97eedd&sc_channel=ps&ef_id=CjwKCAjwwr6wBhBcEiwAfMEQs53MZbuwK2Wf2fGrNUwy7iXCkrV03utzj6VwMG08ohnAl_4YZY94IhoCWLsQAvD_BwE:G:s&s_kwcid=AL, 2024. Accessed: 5 April 2024.

[2] Amazon Web Services, Inc. What is nosql? — nonrelational databases, flexible schema data models. https://aws.amazon.com/nosql, 2023. Accessed: 2023-04-02.

[3] ANTLR. Grammars-v4. https://github.com/antlr/grammars-v4, 2023. Accessed: 28-Nov-2023.

[4] Neeha Ashraf and Manzoor Ahmad. Maximum likelihood estimation using word based statistical machine translation. In *2019 International Conference on Communication and Electronics Systems (ICCES)*, pages 1919–1923, July 2019.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] Chris Bail. Word embeddings. https://cbail.github.io/textasdata/word2vec/rmarkdown/word2vec.html, 2023. Accessed 19 Mar. 2024.

[7] Kirsten Barkved. How to know if your machine learning model has good performance, 2022. Accessed 5 Apr. 2024.

[8] Joel Barnard. What are word embeddings? https://www.ibm.com/topics/word-embeddings, January 2024. Accessed 19 Mar. 2024.

[9] Xinyun Chen et al. Tree-to-tree neural networks for program translation. In *NeurIPS 2018: 32nd Conference on Neural Information Processing Systems*, pages 1–15, 2018.

[10] Francois Chollet. A ten-minute introduction to sequence-to-sequence learning in keras. Blog Name, 9 2017.

[11] DockerHub. The world's largest container registry — docker. https://www.docker.com/products/docker-hub, Oct 2021. Accessed: 2024-04-05.

[12] Jane Doe. Understanding encoder-decoder sequence to sequence model, 2024.

[13] John Doe and Jane Roe. A comprehensive review of long short-term memory (lstm) networks. *Journal of Neural Network Research*, 45(3):123–145, 2023.

[14] Embarcadero. Delphi IDE for native apps: Community edition. https://www.embarcadero.com/products/delphi/starter?utm_source=Google&utm_medium=PPC&utm_campaign=&utm_content=&utm_term=&gad_source=1&gclid=CjwKCAiA6KWvBhAREiwAFPZM7oNtWiW1n_Pt3KylcjOBlGhxlPL5oUn6jas0Vmot6mPH_QKBqQ0SzRoCTgsQAvD_BwE, 1995. Accessed: 7 Mar. 2024.

[15] Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. Tree-to-sequence attentional neural machine translation. *arXiv*, 2016.

[16] freeCodeCamp.org. A history of machine translation from the cold war to deep learning, Mar 2018.

[17] GeeksforGeeks. Flask tutorial. https://www.geeksforgeeks.org/flask-tutorial/, 2023. Accessed: 2023-04-02.

[18] Alexander Gillis. What is mongodb? a definition from whatis.com. https://www.techtarget.com/searchdatamanagement/definition/MongoDB, Mar 2023. Accessed: 2023-03-01.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[20] A.M. Jasmine Hashana, P. Brundha, Mohamed Uvaze Ahamed Ayoobkhan, and Fazila S. Deep learning in chatgpt - a survey. In *2023 7th International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 1001–1005, 2023.

[21] A. Hosna, E. Merry, J. Gyalmo, et al. Transfer learning: a friendly introduction. *Journal of Big Data*, 9:102, 2022.

[22] Oleksandr Hutsulyak. 10 key reasons why you should use react for web development, Mar 2022. Accessed: 2022-03-15.

[23] IBM. Recurrent neural networks - overview, 2021. [Online; accessed 17-March-2024].

[24] IBM. What is java spring boot? `https://www.ibm.com/topics/java-spring-boot`, 2022. Accessed: 2022-04-02.

[25] Kaltura. What is transcoding? all you need to know. `https://corp.kaltura.com/blog/what-is-transcoding/`, 2023. Accessed on: 2023-12-15.

[26] kv391. ANTLR4 Grammar: A Quick Tutorial. `https://medium.com/@kv391/antlr4-grammar-a-quick-tutorial-e1f0fb6ca4ff`, Mar 2023. Accessed: 2024-04-02.

[27] David Lester. Topology in pvs: Continuous mathematics with applications. pages 11–20, 11 2007.

[28] Lili Mou et al. Convolutional neural networks over tree structures for programming language processing. *arXiv*, 2015.

[29] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 651–654, New York, NY, USA, 2013. Association for Computing Machinery.

[30] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 585–596. IEEE Press, 2015.

[31] Tien Nguyen. Code migration with statistical machine translation. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 2–2, September 2016.

[32] Franz Josef Och and Hermann Ney. A comparison of alignment models for statistical machine translation. In *COLING 2000 Volume 2: The 18th International Conference on Computational Linguistics*, 2000.

[33] omniscien. What is rules-based machine translation? `https://omniscien.com/faq/what-is-rules-based-machine-translation/`, 2021. Accessed: 2024-03-17.

[34] Omniscien Technologies. What is neural machine translation? https://omniscien.com/faq/what-is-neural-machine-translation/. Accessed: 7 Mar. 2024.

[35] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[36] Sida Peng et al. The impact of ai on developer productivity: Evidence from github copilot, Feb 2023. https://arxiv.org/pdf/2302.06590.pdf.

[37] P.C. Petersen. Neural network theory. http://pc-petersen.eu/Neural_Network_Theory.pdf, 2021. Accessed: 2024-03-16.

[38] Aminah Mardiyyah Rufai. Statistical machine translation: A data driven translation approach, Jul 2022.

[39] Hendra Setiawan et al. Phrase-based statistical machine translation: A level of detail approach, 2005. Accessed: 2024-03-25.

[40] Irina Sigler. Example-based explanations to build better ai/ml models, Aug 2022.

[41] Derya Soydaner. Attention mechanism in neural networks: Where it comes and where it goes. *Neural Computing and Applications*, 34(16):13371–13385, May 2022. Accessed 15 Mar. 2024.

[42] Alvise Spanò, Michele Bugliesi, and Agostino Cortesi. Typing legacy cobol code. In María José Escalona, José Cordeiro, and Boris Shishkov, editors, *Software and Data Technologies*, pages 151–165, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[43] Chao Su et al. Neural machine translation with gumbel tree-lstm based encoder. *Journal of Visual Communication and Image Representation*, 71:102811, 2020.

[44] Taha Sufiyan. What is node.js: A comprehensive guide. Simplilearn.com, May 2023. www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs.

[45] Zeyu Sun, Qihao Zhang, Zhongyu Fu, Peng Su, Xuan Xing, Xiaoguang Si, Sen Chen, Guoqing Zhang, and Jian Zhao. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 8984–8991. AAAI Press, Apr 2020.

[46] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[47] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566. Association for Computational Linguistics, 2015.

[48] Talend. What is a legacy system. `https://www.talend.com/resources/what-is-legacy-system/#:~:text=A%20legacy%20system%20is%20outdated,doesn't%20allow%20for%20growth.`, Accessed: 2023. Accessed on: 2023-12-15.

[49] Omniscien Technologies. `https://omniscien.com/faq/what-is-neural-machine-translation/`. Accessed on: 2023-12-15.

[50] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does bleu score work for code migration? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176, 2019.

[51] Akshay Ugalmogale. How much minimum data do we need?, 2022. Accessed: 5 Apr. 2024.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv.org*, arXiv:1706.03762, Dec 2017.

[53] Jiajun Zhang et al. Beyond word-based language model in statistical machine translation. *ArXiv (Cornell University)*, Feb 2015. Accessed: 2024-03-25.