LLMs for Code Translation

Thesis submitted by

Lalit Meena (2019CS50439)
Shashank G (2022AIB2684)

under the guidance of

Prof. Srikanta Bedathur Jagannath (Indian Institute of Technology Delhi)
Dr. Yogish Sabharwal and Dr. Anamitra Roy Choudhury (IBM India Research Lab)

in partial fulfilment of the requirements
for the award of the degree of

Bachelor and Master of Technology

Department Of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

January 2024

THESIS CERTIFICATE

This is to certify that the thesis titled **LLMs for Code Translation**, submitted by **Lalit Meena (2019CS50439)** and **Shashank G (2022AIB2684)**, to the Indian Institute of Technology, Delhi, for the award of the degree of **Masters of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Srikanta Bedathur
Dept. of Computer Science and Engineering
IIT-Delhi, 110 016

Place: New Delhi

Date: 30th June 2024

ACKNOWLEDGEMENTS

We would like to express our deepest appreciation to our advisor, **Prof. Srikanta Bedathur**, whose expertise, understanding, and patience have significantly enriched our graduate experience. Our heartfelt thanks also go to Dr. Yogish Sabharwal and Dr. Anamitra Roy Choudhury from IBM India Research Lab, and Monika, for their valuable insights and feedback. The regular weekly meetings with the IBM Research team were incredibly educational and beneficial. We are also grateful for the computing resources provided for this project.

Lalit and Shashank have contributed equally to this project, and we are both grateful to each other for our contributions and valuable work on this project.

ABSTRACT

Code translation has long posed a challenge for the AI community but is essential for organizations and individual developers seeking to migrate legacy codebases to modern languages or translate important libraries present in one language to another. The scarcity of real-world parallel corpora containing source and target translation pairs exacerbates the problem. Since their advent, LLMs have been successful in various tasks such as code generation, summarization, and question-answering etc. Chain-of-Thought (CoT) reasoning, a technique that enables LLMs to systematically reason through tasks to arrive at a solution, has demonstrated remarkable success without requiring fine-tuning. However, its application in translating source code from one language to another remains unexplored. In this work, we investigate approaches utilizing CoT and propose a metric to evaluate the structural resemblance between source and target code. Our method, CodeTransCoT, significantly outperforms the Vanilla method, with performance improvements from 18% to 42% in Java-Python translation on the Avatar dataset and from 9% to 53% in Python-Java translation on the CodeNet dataset. Additionally, our approach maintains the structural quality and logic of the source program, as demonstrated by the proposed metric, thereby reducing compilation errors.

Contents

List of Tables

List of Figures

ABBREVIATIONS

IITD	Indian Institute of Technology, Delhi
IBM	International Business Machines
RTFM	Read the Fine Manual
LLM	Large Language Model
LLMs	Large Language Models
DNN	Deep Neural Network

NOTATION

<< ... >> - This notation can be used like this << *content_to_fill* >>. It means if it is used inside the input prompt template then it should be appropriately filled to complete the prompt else the content inside will be generated by LLM model according to output template.

Chapter 1

INTRODUCTION

Accurate and effective translation of source code from one programming language to another has become an essential challenge within AI research communities. Organizations and developers also frequently encounter the need to translate code in their work. This necessity arises from various factors, such as the need to migrate legacy codebases to modern languages to enhance performance, maintainability, or compatibility with current technologies. Code translation is also helpful when you need to make libraries which is present in another language. Additionally, translating code from strongly typed languages like Java to dynamically typed languages like Python, which offer low maintenance costs and ease of understanding, has become a common requirement. This translation process can be labor-intensive and error-prone when done manually, underscoring the need for automated solutions.

Large language models (LLMs) have significantly enhanced tasks such as natural language understanding, text generation, and machine translation, improving accuracy and coherence in these areas. They also present a promising approach to automating code translation, offering a potentially less expensive and more scalable solution. However, there are many challenges associated in code translation with the help of LLMs. Firstly, there is a scarcity of real-world parallel corpora containing source and target translation pairs, which are essential for fine-tuning LLMs. Even when such data is available, fine-tuning large models with billions of parameters demands substantial computational resources that are not readily accessible to everyone. Despite their potential, most LLMs still struggle to generate consistently accurate and reliable translated code.

The primary focus of past research in this area has been on improving the execution accuracy of translated code, ensuring that the translated code performs the same functions as the original. Lachaux et al. [LRCL20] address the challenge of limited parallel corpora in this field. They utilize techniques such as Cross-lingual Masked Language Model pretraining, Denoising auto-encoding, and Back-translation to develop a fully unsupervised neural transcompiler, demonstrating high accuracy in translating functions between C++, Java, and Python. Similarly, Szafraniec et al. [SRL$^+$23] utilize these techniques but further enhance the dataset with LLVM IR (Intermediate Representations) to better capture the similarities between programs with different semantics in various languages. Xie et al. [XNFR23] explore data augmentation techniques by constructing comparable corpora (parallel corpora where the source-target pairs have similar functionality) and augmenting available parallel data with different reference translations to increase target translation variability. Other researchers have also employed reinforcement learning to address this problem. Jana et al. [JJJ+24] integrate reinforcement learning into the fine-tuning process, incorporating compiler feedback and symbolic execution (symexec)-based equivalence testing to ensure functional equivalence between input and output programs. Additionally, PPOCoder [PSR23] is an RL-based code translation framework that leverages CodeBLEU-inspired feedback in the LLM training process.

All previous approaches involve fine-tuning or reinforcement learning feedback, which, while effective, are not user-friendly for those inexperienced with fine-tuning LLMs. Additionally, no prior work has analyzed the types of errors occurring in test sets. Pan et al. [PIK+24] conducted a large-scale evaluation and taxonomy of translation bugs, identifying that many bugs stem due to the difference in syntax and semantic difference between two languages, missing dependencies, incorrect logic implementation, and mishandling of input-output. We believe that effective mitigation strategies can only be developed by first classifying the different types of bugs encountered and by distinguishing programming languages based on their paradigms (e.g., strongly typed, loosely typed, object-oriented).

Considering the unresolved challenges, we have selected Java and Python for translation, as Java is object-oriented strongly typed and Python is multi-paradigm dynamically typed. Previous research [PIK+24] has shown that Java-Python and Python-Java translations have historically underperformed, making them ideal candidates for further research. Additionally, Java and Python are among the most widely used programming languages globally, increasing the relevance and impact of our research. We use the CodeNet and Avatar datasets, which contain code written by competitive programmers and submitted to various online judges, including AtCoder, AIZU, Google Code Jam, Codeforces, GeeksforGeeks, and LeetCode. Our approaches relies solely on prompt-based methods, eliminating the need for any fine-tuning. We evaluate the translated programs based on functional correctness.

While execution accuracy is a crucial parameter for judging code translation, it is not the only important factor, as it is essential that the translated code maintains structural resemblance to the original to ensure a smooth transition from one language to another. Having structural similarity also helps preserve function-wise logic of the source code leading to correct translations. To measure this structural similarity score, we propose a quality metric to measure the quality of the translation.

We present various approaches, such as explanation-based and pseudocode methods, where we include explanations or pseudocode of the source program as additional information in the prompt. Additionally, we implement In-Context Learning by selecting samples manually, which significantly reduces compilation errors. Our efforts then shift to CompWiseTrans, a Chain-of-Thought based approach, aiming to preserve the structural similarity to the source program. Ultimately, we achieve impressive results with the CodeTransCoT technique. We found that reducing compilation errors significantly boosts performance, and maintaining structural resemblance which aids in generating compilation-error-free code.

Our main contributions are as follows:

- We introduce CodeTransCoT, a novel Chain-of-Thought based technique designed to efficiently and accurately translate code from one language to another while preserving structural similarity.
- We conduct an extensive study of this approach using various models such as StarCoder, CodeLlama, Granite-8B, and Granite-20B, utilizing the Avatar and CodeNet datasets for their accessibility and the availability of test cases for evaluating functional correctness. We report Pass@1*, Pass@1, and Pass@5 metrics for all settings and analyze the types of errors generated.
- We propose a Quality metric to measure structural similarity and verify if the generated program matches the intended output. This involves performing static analysis of the code and extracting call graph information.

Chapter 2

Problem Statement

Multilingual Code Translation is the process of converting source code written in one programming language into an equivalent code in another language while preserving its functionality and readability. Traditional methods for code translation involve manual rewriting or rule-based systems, both of which are time-consuming and error-prone.

This work investigates the use of LLMs to improve the quality of code translation i.e. to improve the ability to accurately convey the intended functionality of the input code. As errors in translation can result in incorrect behavior, inefficiencies, or non-functional code. This work focuses on improving the quality and accuracy of code translation using state-of-the-art Large Language Models (LLMs).

Chapter 3

Related Work

Rule-based transpilers, or handcrafted rule-based transpilers, typically utilize traditional compiler techniques and concepts such as parsing and abstract syntax trees. Examples include java2python [TMS16] and py2java [Fom19]. The TSS code converter [tss23] is a commercial J2P transpiler. However, many of these tools include disclaimers noting that the translated code may not compile or run without further adjustments.

Lachaux et al. [LRCL20] tackle the challenge of limited parallel corpora by leveraging advanced techniques in unsupervised machine translation. They employ Cross-lingual Masked Language Model pretraining, Denoising auto-encoding, and Back-translation to develop a fully unsupervised neural transcompiler. Their model demonstrates high accuracy in translating functions across C++, Java, and Python. They introduce a test set comprising 852 parallel functions and unit tests to verify translation correctness, showing superior performance over rule-based commercial baselines. Building on this foundation, Szafraniec et al. [SRL+23] expand the dataset with LLVM IR (Intermediate Representations) to enhance capturing semantic similarities between programs across different languages. They augment existing test sets for code translation by including hundreds of functions from Go and Rust.

Xie et. al. [XNFR23] explore data augmentation techniques by constructing comparable corpora (parallel corpora where the source-target pairs have similar functionality) and augmenting available parallel data with different reference translations. They construct multiple types of comparable corpora, such as Naturally available corpora, Generated comparable corpora and even Random Comparable Corpora and analyze their effect on code translation task. Furthermore, they automatically generate additional translation references for available parallel data to reduce overfitting to a single reference translation and increase variation in target translations.

Previous studies have also explored the application of reinforcement learning to address this challenge. Jana et. al. [JJJ+24] train an LLM via reinforcement learning, by modifying the fine-tuning process to incorporate compiler feedback and symbolic execution (symexec)-based equivalence testing feedback that checks for functional equivalence between the input and output program. They achieve the best-known results on the Avatar-TC dataset, with 53% accuracy for Java-Python translations and 48% for Python-Java translations. PPOCoder [PSR23] is a reinforcement learning-based code translation framework that incorporates CodeBLEU-inspired feedback during the training of the large language model. It employs a Boolean compiler feedback mechanism, which returns true if the generated code compiles and false otherwise.

TransCoder-ST [RLCL20] is an unsupervised code translation tool that leverages self-training and automated unit tests to ensure source-target code equivalence. It generates a synthetic dataset of equivalent code in two languages using these unit tests. However, the primary focus of these tools is to achieve logically correct translations, often at the expense of translation quality.

Chain-of-Thought (CoT) prompting has emerged as a groundbreaking tool in Natural Language Processing (NLP), particularly for its efficacy in complex reasoning tasks. Various paradigms of CoT prompting have been developed to enhance the performance of CoT-based approaches. These paradigms include Zero-Shot-CoT, which utilizes a stepwise reasoning prompt (Kojima et al., 2022 [KGR+22]), Manual-CoT , which employs manually crafted demonstrations (Wei et al., 2022 [WWS+22]), and Auto-CoT, which automatically constructs demonstrations (Zhang et al., 2022 [ZZLS22]).

While there is no direct application of CoT in code translation tasks, related research has explored the use of CoT in code generation tasks, such as CodeCoT [HBQC24], Structured CoTs (SCoTs) (Li et al., 2023 [LLLJ23]), and COTTON (Yang et al., 2023 [YZC+23]). CodeCoT [HBQC24] ensures that the generated code follows the correct logical flow through the CoT approach. SCoTs [LLLJ23] leverage the rich structural information in source code, incorporating structured intermediate reasoning steps to produce well-structured source code.

Inspired by these works, we have applied CoT to the code translation task with our CodeTransCoT approach. Our method utilizes CoT with structure information-based reasoning to improve complex code translation, drawing from the strengths demonstrated in related CoT-based code generation research.

Chapter 4

Challenges in Code Translation

In the past, Pan et. al. [PIK+24] have performed a large-scale evaluation and taxonomy of code translation bugs introduced by several general and code LLMs, when translating from a set of source languages to a set of target languages. They concluded that a large percentage of translation bugs can be attributed to the difference in syntax and semantics between the source and the target languages. Also, a large number of bugs are introduced due to the missing dependencies, incorrect logic implementation and incorrect handling of input - output data.

While it is useful to understand a general pool of translation bugs introduced by LLMs, we believe any approach to mitigate them, can be built only after classifying programming languages into various categories like strongly typed languages, loosely typed languages, object-oriented languages etc, based on the programming paradigm they support, and then, analyzing the broad syntactic and semantic distinctions between these source and target language categories. Pan et al. [PIK+24] observed that the effectiveness of code translation can vary significantly depending on the characteristics of the source and target programming languages, including factors such as the type system, available programming APIs, and support for metaprogramming through decorators or annotations. In our experiments also, we observed that understanding the differences in the programming paradigms from a given source code language to the required target code language is the key to produce more successful translations. *Why chose Java, Python* : In this paper, we chose the language pair $(Java, Python)$ for code translation experiments as these two are representatives of two different programming paradigms. Java is an object-oriented language, while Python is a multi-paradigm language that supports object-oriented programming, procedural programming as well as functional programming. The fact that Java is strongly-typed and Python is loosely-typed poses a variety of challenges for LLMs when translating code from Java to Python and vice-versa. Many past work [PIK+24] have also experimented on this set of languages. Many good benchmarks like $AVATAR$ [ATCC21] and $CODENET$ [PKJ+21] are available for experimentation in this set.

Before we present our translation approach, in the next few subsections, we first discuss our key observations on the various bug categories introduced by LLMs, when translating from a more structured and strongly-typed language like Java to a more loosely-typed language like Python and vice-versa. The key observations we made during this investigation is what led us to our $TransCoT$ approach.

The bug-categories we discuss below were generated as part of *vanilla prompting* experiments [PIK+24], where each prompt was of the following format.

```
$SOURCE_LANG:
// Unformatted source code
Translate the above $SOURCE_LANG code to $TARGET_LANG.
end with comment "<END-OF-CODE>".
```

SOURCE_LANG and *TARGET_LANG* were either *Java* or *Python*. We analyzed the translation results of around 500 standalone programs (250 Java programs and 250 Python programs) from *AVATAR* [ATCC21] benchmark, for which we had multiple test cases to account for a successful/unsuccessful translation. As in past work [RLCL20, PIK+24, JJJ+24], we deem a translation successful if it compiles successfully, generates no runtime errors, and passes all tests successfully on the translated code. Furthermore, we go one step beyond a mere successful translation. We also consider the ***Quality of Translation***, which is, how closely the generated code structurally resembles the original source code. We highlight why monitoring this quality of translation is important in the next few subsections

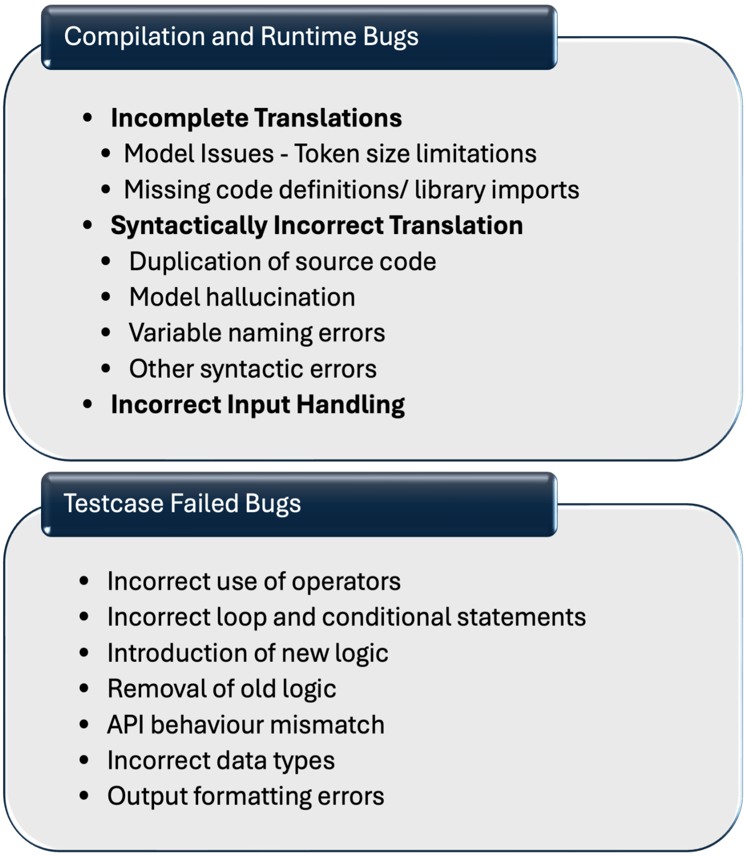

Figure 4.1: Categorisation of Translation Bugs Introduced by LLMs.

4.1 Bug Categories: Java ⇌ Python Translations

Python belongs to the category of loosely or dynamically typed programming languages and gives its programmers the flexibility to write functional code, procedural code, and even object-oriented code. In comparison, Java is an object-oriented and strongly typed language that requires proper data type declarations. These programming paradigm differences between these two languages make automatic translation of code from Java to Python (and vice-versa) a daunting task.

As shown in figure 4.1, we categorize the bugs introduced by LLMs during Java ⇌ Python translations into following two broad categories based on what time they manifest themselves - **Compilation or Runtime failed bugs**, and **Test case failed bugs**.

4.1.1 Compilation or Runtime Failed Bugs

Compilation bugs or Runtime failed bugs are where the translated code either fails to compile, or, breaks down before completing a full execution run. These result mostly from no or incomplete translations or syntactically incorrect translations.

Incomplete translations can happen due to model specific constraints like token size limitations. Missing code definitions and/or missing library imports in the translated code can break the code execution flow, thus generating compilation or runtime errors in the translated code.

Syntactically incorrect translations can happen when the model is unable to translate portions of source code and hence emits them as-is in the translated code. Another reason for syntactically incorrect translations is due to the hallucination problem in LLMs. Models may hallucinate sometime due to their limited contextual understanding of the given input source code.

Yet another reason for compilation or runtime-failed bugs is due to variable naming issues. An example of this is illustrated below.

```java
// Original Java code
int[][] range = new int[q][2];
...
range[i][0] = sc.nextInt();
```

```python
# Incorrect Translated Python Code
range = [list(map(int , input.split())) for _ in range(q)]
```

As shown, the variable name *range* is perfectly valid in the Java code, whereas in Python, it is the name of a commonly used built-in function and, therefore, does not qualify to be used as a variable name as it is causing an error when these built-in functions are called later in the program. Other examples of such problematic variable names in Python are *max, min, in, out* etc.

Mismatch in input parsing format between the input source code and the translated code is another reason for a fairly significant number of runtime errors.

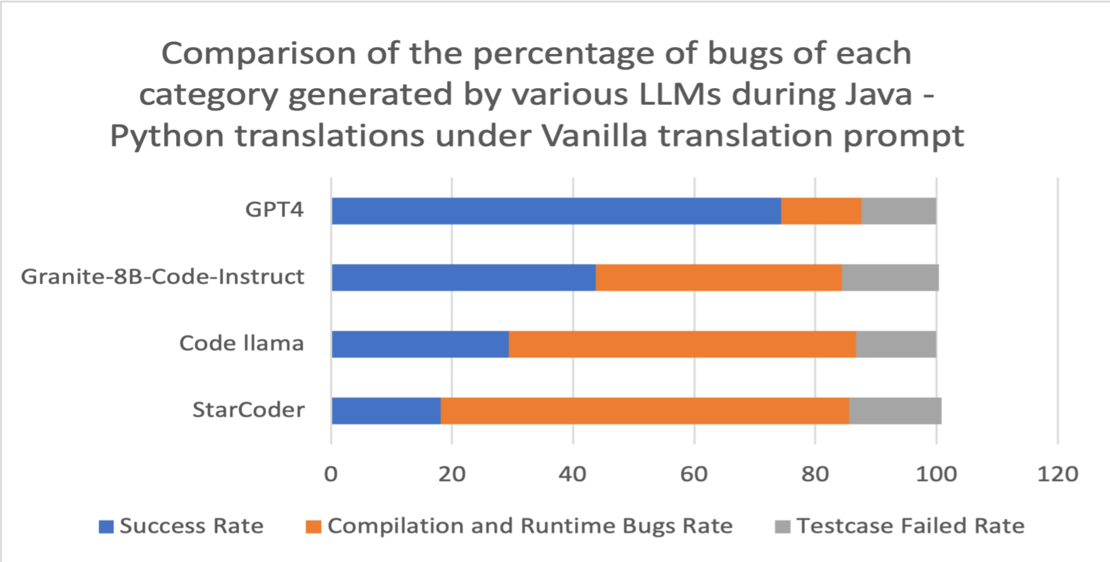

Figure 4.2: Comparison of percentage of bugs of each category in translated Python code. Benchmark: AVATAR.

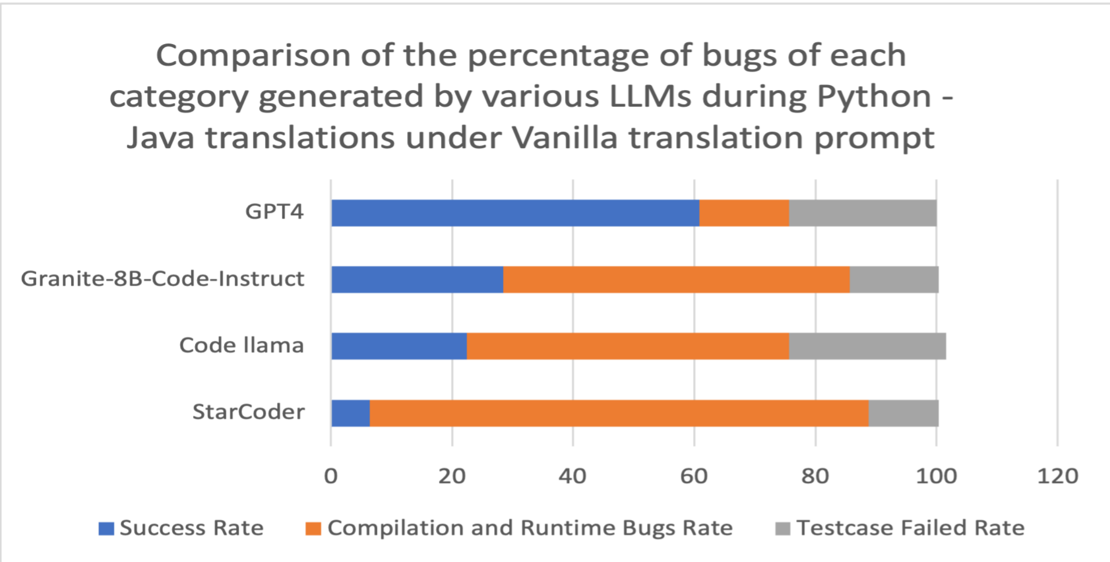

Figure 4.3: Comparison of percentage of bugs of each category in translated Java code. Benchmark: AVATAR.

We experimented with three open-source LLMs : *StarCoder* [LAZ+23] and *Code Llama* [?], and *Granite-xB-Code-Instruct* [MSZ+24] and one closed-source LLMs : *GPT-4* [OAA+23], to get an estimate of the percentage of bugs of each category, introduced by various LLMs, in the translated Python and Java codes. Benchmark used in this experiment was *AVATAR* [ATCC21]. The results of the same are shown in figures 4.2 and 4.3.

Observation 1. Python programmers often structure solutions in the form of functions to write code that is shorter, easier to test, less prone to bugs, and well-suited for multiple applications. This functional programming paradigm considerably differs from Java's object-oriented programming paradigm, and therefore results in a lot of compilation and run-time errors (figure 4.3) in the translated Java code. In the translated Java code, for open-source LLMs, between 53% - 82% bugs are compilation or runtime failed bugs.

Observation 2. In the translated Python code, for open-source LLMs, between 40% - 67% bugs are compilation or runtime failed bugs.

Learning 1. **During our experiments, we observed that if we instruct the LLMs to closely follow a well-structured, progressive, chain-of-thought based reasoning approach to translate code, it helps reduce the number of cases where the model may generate incomplete or syntactically incorrect translations, thereby considerably reducing a good number of compile-time errors. This learning forms the basis of our *TransCoT* approach.**

4.1.2 Test Case Failed Bugs

Test case failed bugs are those that are due to logical errors in the translated code. Consider the example of a logical error shown below. In the translated Python code, the *XOR operator* from the original Java code is replaced with the *NotEqualTo operator*. Due to a mismatch in the precedence order of these two operators, an erroneous translation of the input conditional statement happens resulting in a logical error in the translated code.

```
if (a[m] >= 0 ^ a[i] >= 0) # Original Java code
if a[m] >= 0 != a[i] >= 0: # Incorrect Translated Python Code
```

Another example of a logical error is shown next. In the original Java source code, the loop index variable i is to be incremented within the loop body, if certain condition holds true. In the translated Python code, the normal loop formulation get replaced with a well-known Python built-in function called *range*. However, in Python, the *range* function prevents its loop index variable to be modified anywhere within the loop body, thereby breaking the code logic in this case.

```
// Original Java code
for (int i=0; i<input.length(); i++) {
    // more code
    if (count>max)
        i++;
    else
```

```
        count++
    // more code
}
```

```
# Incorrect Translated Python Code
for i in range(len(input)):
    # more code
    if count > max:
        i+=1
    else
        count+=1
    # more code
```

Many more examples of various reasons behind test case failed bugs were presented by Pan et. al. [PIK+24].

Observation 3. We observed that, while compilation and runtime failed bugs could be attributed to programming paradigm differences between Java and Python; the test case failed category of bugs occurred mostly due to language specific differences, e.g., API differences, between Java and Python. We also observed that prompt crafting techniques are not sufficient to resolve test case failed category of bugs, and that, these may only be resolved with more specific training of LLMs for it

4.2 Translation Quality Bugs

Compilation and Runtime failed bugs, and, Test case failed bugs are well discussed in all past work. We now introduce a new third category of translation bugs - *Translation Quality Bugs*.

A translation is deemed successful if it compiles successfully, generates no runtime errors, and passes all tests successfully on the translated code. However, we claim that the story does not end here. To motivate why *translation quality* is important, consider the code translation example shown below.

As shown in figure 4.4, the original Java source code defines two nested classes: *BUnhappyHackingABCEdit* and *LightScanner*. The static member class *BUnhappyHackingABCEdit* defines a *solve* method, which first reads an input string from a given input stream, and then uses it to iteratively build and print an output string.

```java
// Original Input Java Source Code
// Code Source - AVATAR Benchmark
public class XXX {
   public static void main(String[] args) {
     InputStream inputStream = System.in;
     OutputStream outputStream = System.out;
     LightScanner in = new LightScanner(inputStream);
     PrintWriter out = new PrintWriter(outputStream);
     BUnhappyHackingABCEdit solver=new BUnhappyHackingABCEdit();
     solver.solve(1, in, out);
     out.close();
   }
   static class BUnhappyHackingABCEdit {
     public void solve(int testNumber, LightScanner in,
                                         PrintWriter out) {
       String s = in.string();
  StringBuilder d = new StringBuilder();
  for (char c : s.toCharArray()) {
    switch (c) {
      case '0':
       d.append("0"); break;
    case '1':
       d.append("1"); break;
    case 'B':
       if (d.length() > 0)
          d.setLength(d.length() - 1).trimToSize();
       break;
    }
  }
  out.println(d);
     }
   }

   static class LightScanner {
     private BufferedReader reader = null;
     private StringTokenizer tokenizer = null;

     public LightScanner(InputStream in) {
  reader = new BufferedReader(new InputStreamReader(in));
     }

     public String string() {
      if (tokenizer == null || !tokenizer.hasMoreTokens()) {
  try {
    tokenizer = new StringTokenizer(reader.readLine());
  } catch (IOException e) {
    throw new UncheckedIOException(e);
  }
      }
      return tokenizer.nextToken();
     }
   }
}
```

Figure 4.4: Example of an input Java source code.

The other static member class is *LightScanner* class, which has methods to initialize an input stream and read an input string from it. The main class *XXX* defines a *main* method, which first instantiates an input stream using the *LightScanner* class, and then calls the *solve* method of class *BUnhappyHackingABCEdit*.

Next, consider the Python translation (figure 4.5) of the above Java code as generated by common LLM - *GPT-4*. The translation is syntactically and logically correct, and, it passes all test cases. However, the interesting point to note in the translated Python code is that, all class and subclass structure from the original Java source code has been scrapped, and, just the basic functionality of the program, i.e., to read an input string, and, use it to generate and print an output string has been put into a global method.

```
# Logically Correct Translation of above Java Code to Python Code
def solve():
    s = input().strip()
    d = []
    for c in s:
        if c == '0':
            d.append('0')
        elif c == '1':
            d.append('1')
        elif c == 'B':
            if len(d)>0
                d.pop()
    print(''.join(d))
solve()
```

Figure 4.5: Python translation of Java code in figure 4.4, as generated by GPT-4 under vanilla prompting technique.

The above translation, although correct, raises a concern. What if the original Java program of figure 4.4 is a part of a larger Java package, which is to be entirely translated to Python. If in the package, there are some other Java classes, which instantiate and call the *solve* method of *BUnhappyHackingABCEdit* class, the connectivity between these classes would completely break after translation.

Take another example of a simple Python program (figure 4.6) this time. In the global scope of this program, is an integer variable named a, and a method named $func$, which computes and prints the sum of the digits of the integer variable a.

Next, consider the Java translation (figure 4.7) of the above Python code, as generated by common LLM - *StarCoder*, under vanilla prompt. The translation has compilation issues about missing imports, and, is also logically incorrect due to removal of last *if-else-statement block* in method $func$. However, the interesting point to note in the translated Java code is that, all method declarations and global variables declarations has been scrapped, and, just the basic functionality of the program has been put into a *main* method.

As in the previous example, the concern here also remains the same. What if the original

```python
# Original Input Python Source Code
# Code Source - CodeNet Benchmark
a = int(input())
def func(a):
    digits = []
    temp = a
    while temp != 0:
        digits.append(temp%10)
        temp = int(temp/10)
    result = sum(digits)
    if result == 1:
        print('10')
    else:
        print(result)
func(a)
```

Figure 4.6: Example of an input Python source code.

Python program of figure 4.6 is a part of a larger Python package, which is to be entirely translated to Java. If in the package, there are some other Python programs, which call the *func* method and/or access the global variable *a*, the connectivity between these programs would completely break after this translation.

```java
// Logically Incorrect Java Translation of above Python Code
// Code also has Compilation Errors
public class YYY{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        int sum = 0;
        while(a!= 0) {
            sum += a % 10;
            a /= 10;
        }
        System.out.println(sum);
    }
}
```

Figure 4.7: Java translation of Python code in figure 4.6, as generated by StarCoder under vanilla prompting technique.

An important question now to ask here is -
What is the purpose of using LLMs for code translation? Do we need them to translate standalone programs or entire legacy code bases?
If standalone programs be the answer, we are great with what we are seeing at this level, i.e., more syntactically and logically correct translations. However, we believe the answer to the above question is legacy code bases, as standalone programs may well be translated manually rather than spending so many model resources on them. Now let us see what comprises a code base? Indeed it is a set of interconnected programs, and not just a bunch of standalone programs. This brings us to an important research question.

RQ **: How can entire code bases, which are essentially a large set of inter-**

connected programs, be effectively translated from one programming language to another, without disturbing their interconnections?

The answer to the above question is - by as far as possible, preserving the original program structure even after translating to the target language.

Learning 2. **During our experiments, we observed that, LLMs produce better quality translations, if we instruct them to follow a chain-of-thought based reasoning approach in which they first understand the original source code structure, and then progressively translate portions of source code while still preserving the original structure in the output code. This learning also forms the basis of our *TransCoT* approach**

Since *quality of translation* is an important parameter, for our experiments, we use it as another metric to measure the translation quality of LLMs.

Chapter 5

METHODOLOGY

In the previous Chapter, we have seen different types of bugs and challenges associated with the code translation task. The motivation behind our approach is to deal with challenges faced by LLMs in code translation. Based on the observations and learnings in Chapter 4, some of the main things to focus on while working with LLMs are to make them understand what task we want to solve. Another thing is to assist them in solving the task without getting hallucinated. These things can be achievable with LLMs by providing them with relevant information and descriptions of the problem task with appropriate approaches. To make LLMs work for any task, we can interact with LLMs by assuming them as black boxes, with the help of prompting instructions to provide input information and decoding-based strategies for generating output. We have shown the evaluation part and output generation part of LLMs in Chapter 7. So, we can mainly formulate approaches for LLMs by encoding relevant information in the form of prompts or instructions.

5.1 Vanilla Approach

One of the naive ways to give a description of the problem task to LLMs is a simple vanilla prompt. For this code translation task, the vanilla prompt provides simple instructions for translating the source language code to the target language code. Figure 5.1 has a vanilla prompting template for any LLM, but it can differ slightly according to different LLMs based on specific tokens and generation styles.

```
$SOURCE_LANG:
$SOURCE_CODE
// Unformatted source code
Translate the above $SOURCE_LANG code
to $TARGET_LANG end with comment
"<END-OF-CODE>".
$TARGET_LANG:
// Code generated
```

Figure 5.1: Vanilla prompting template.

5.1.1 Discussions

The Vanilla Approach is a really simple approach that has faced a low translation rate and different types of bugs. In Figure 4.2 and Figure 4.3, we can see that a significant portion of errors are runtime and compilation errors. Specifically, it constitutes more than 50% failing rate in most open source LLMs. One of the main reasons is that LLMs are not able to figure out relevant information for code translation itself, hence resulting in significant errors due to hallucination. So there is a need to provide additional contextual information in Prompt based approaches.

5.2 Additional Context based Approach

In addition to simple prompting instruction we can also provide relevant additional information to make easy for LLM to do code translation of *complex programs*. Additional information for programs in source code language in dataset, can be sample INPUT/OUTPUT format information, explanation of source language code in nature language description, program analysis information extracted out of source code language such as type information, Abstract Syntax Tree(AST) related information, etc.

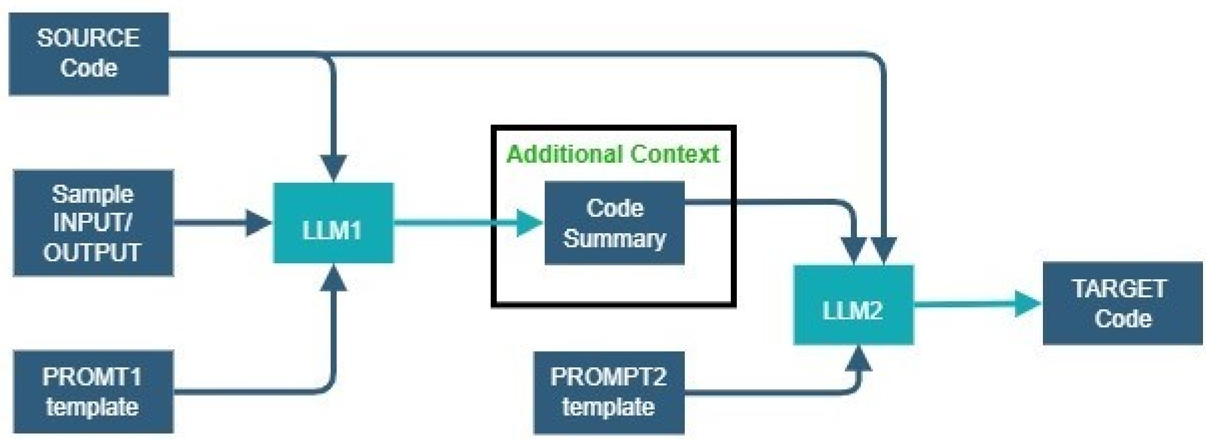

Figure 5.2: Additional Context-based Approach

5.2.1 Code Summary-based Approach

One of the most relevant additional information about source code to LLMs will be an explanation of the complex programs in natural language. These language models better understand the logic expressed in natural language statements compared to statements in source programming language. The idea is that the code summary of the source and target language codes in natural language is more similar due to the explanation of the same problem's solution logic and in the same natural language. So, in this approach, we have utilized a code summary of the source code language as additional information in context for the code translation task.

More Details

Here is the prompting template for this approach in Figure 5.3, which is used in the Additional Context-based approach's pipeline depicted in Figure ??. In this pipeline, Code Summaries are generated using the first prompt (Prompt1) with *gpt-3.5-turbo* model [Ope23] (LLM1). Then these code summaries are fed in as additional context in the second prompt (Prompt 2). Finally, we will generate a target language code by giving Prompt 2 to LLM2, which is, Starcoder model [LAZ+23] for this experiment.

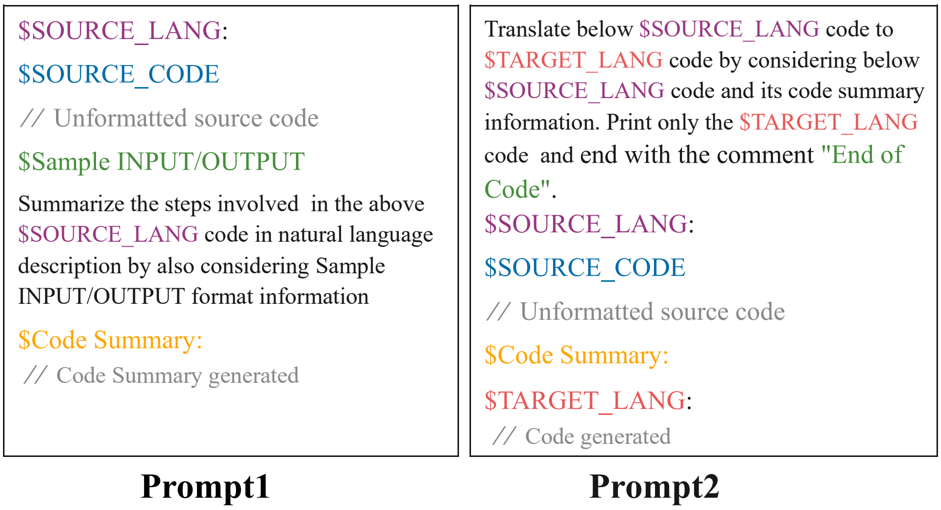

Figure 5.3: Code Summary-based Approach's prompting templates.

Discussions

With this information from the Code Summary, we got an improvement of 4-5% increment (Table 7.4) in successful translations as compared to the vanilla approach. It helped in mitigating the Data-related bugs (especially incorrect input parsing and output formatting bugs) as we had provided sample Input/Output also and an explanation about it in the code summary. ***Observation 4.*** We have observed that the pre-translation Step of generating good, natural language code summaries is a major cost and time hurdle in this approach.

Also, even with efficient hand-crafted code summaries, there was not major improvement in the percentage of successful translations.

Learning 3. Code Summary-based approach is an amalgamation of code translation and code generation tasks, and it does not help much in improving the percentage of successful translations.

5.2.2 Pseudocode based Approach

According to Observation 4. and Learning 3. in the previous section, we can observe that natural language explanation of source code is also not improving the task much as it is also coming with new errors possibility from the task of Code-to-Text summarization with LLMs. After seeing vanilla and Code Summary-based approaches, we can claim that neither source code information nor natural language explanation of source code is most relevant or close to the corresponding target language code. Our new approach is based on the hypothesis that the pseudocode of source code language is most relevant or close to the target code.

More Details

The Prompting template for this approach is also similar to Figure 5.3 that is used similarly in the Additional Context-based approach's pipeline depicted in Figure 5.2.

There are different notations for writing pseudocodes of given programming code: Structured Programming Constructs, Control Structure Notation, and Indentation-based. We have used *GPT-3.5-turbo* model [Ope23] to generate pseudocode in Control Structure notation (shown in Figure 5.4) as it is more concise than English, have loops notation and clarity regarding scope separation.

Discussions

Our experiments result (Table 7.4) shows around 3% improvement as compared to the Code Summary-based approach and 7% improvement as compared to the vanilla approach. These results show the effectiveness of this pseudocode-based approach and hence support our hypothesis that it is more relevant for the generation of target language code.

This approach can handle input/output format better than the previous one, resulting in some reduction of compilation errors. However, still, in this approach, there is much content for LLM to focus on or chances of getting hallucinated.

```java
// Java Code
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        long ans = 0;
        for (int i = 1; i < n + 1; i++) {
            ans += (long) i * ((long) n / (long) i) * (((long) n / (long) i) + 1) / 2;
        }
        System.out.println(ans);
    }
}
```

```
# Pseudocode Explanation

START
    Import Scanner class
    Define class Main
        Define main method
            Create a Scanner object'sc' to read input
            Read an integer 'n' from the user
            Initialize a long variable 'ans' to 0
            Start a for loop with index 'i' from 1 to n + 1
                Update 'ans' by adding the product of i, n/i, and (n/i + 1)/2
            End of loop
            Print 'ans'
        End of method
    End of class
End
# End of Pseudocode
```

```python
# Python Code
import sys

def main():
    input = sys.stdin.readline
    n = int(input())
    ans = 0
    for i in range(1, n+1):
        ans += i * (n // i) * ((n // i) + 1) // 2
    print(ans)

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

Figure 5.4: An example of the comparison of Pseudocode's similarity with the source Java code and the target Python code

5.3 Few-shot based Approach

Despite LLMs being trained on large amounts of data and tuned to follow instructions (Wei et al., 2021 [WBZ+21]), we can't put much information at once to LLMs for solving complex tasks. These things can be considered zero-shot prompting, which means that these previous zero-shot-based approaches used prompts that did not contain examples or demonstrations related to the task.

Whenever zero-shot falls short on more complex tasks than in Natural Language Processing (NLP) literature, it is common to use Few-shot or in-context learning (Brown et al., 2020 [BMR+20]). Few-shot-based approaches help LLMs to understand the task with a demonstration of the task and to get an idea of how to solve the task. They are focused on task-agnostic performance and control the direction of LLM's generation as compared to zero-shot based by providing information from demonstration examples such as overall format of sequence, label space and input text (Min et al., 2022[MLH+22]).

5.3.1 Few-shot Prompting

Few-shot Prompting (Brown et al., 2020 [BMR+20]) refers to the prompting setting where the model is provided with few demonstrations (input-label pairs) for the conditioning at the time of inference. This enables in-context learning (ICL) for the model for better task-agnostic performance.

5.3.2 More Details

For this few-shot-based approach, we have provided Few-shot prompts specific to the code translation task. Here is the prompting template for this approach in Figure 5.5.

For this few-shot-based approach, we have manually crafted three simple demonstrations for each of the AVATAR and CODENET datasets (Section 7.2). This was done to ensure that few-shot would work well for a particular dataset if demonstrations have been chosen from the same distribution of problems in that dataset. We have only taken three demonstrations for in-context learning due to context-length related limitations, which is 2048 tokens in this case. For each of the datasets, most of the demonstrations are a few line codes but with some small variations like input/output formatting. One of the demonstrations for CODENET is a complex program involving multiple functions, etc. This demonstration is indicating a program may involve diverse and complex components. Other demonstrations are simple Java programs with a "Main" class and a "main" function to help the model understand CoT reasoning steps easily.

```
Translate the $Source_lang code to corresponding
$Target_lang code-

1. $Source_lang:
$Source_Code
1. $Target_lang:
$Target_Code

2. $Source_lang:
$Source_Code
2. $Target_lang:
$Target_Code

3. $Source_lang:
$Source_Code
3. $Target_lang:
$Target_Code

4. $Source_lang:
$Source_Code
4. $Target_lang:
// Target_Code to be generated
```

Figure 5.5: Few-shot-based Approach's prompting templates.

5.3.3 Dicussions

There is almost doubled accuracy from 17.67% in the vanilla approach to the 30.12% (Table 7.4) ,as LLM is figuring out relevant knowledge to solve code translation task better. There was a 70% increase in the total number of correct translations as compared to the vanilla approach. These three examples in the prompt did help to resolve 92% compilation errors as LLM was not hallucinating much, and from conditioning with demonstrations, LLM has become aware of the target code, some syntax, and its understanding of the target language. There is further scope for improvement for this few-shot-based approach by carefully crafting demonstrations, as we have manually selected three trivial examples as in-context examples. For it, we can consider many factors like the diversity of examples, order and other factors described in Min et al., 2022 [MLH+22].

This approach did not effectively resolve compilation errors, but it migrated most of them into Runtime error cases. Also, we have observed less improvement in Python to Java translation as LLM might not be able to figure out much information from loosely typed Python. The generation of new source code is now more biased towards the demonstrations provided in the context. There were cases where incorrect translation happened due to simplification assumptions about the input format that it was expecting input format like an in-context demonstration. This is not a perfect technique, especially when dealing with more complex reasoning tasks.

5.4 Step-Wise Translation

Previous approaches have limitations associated with this complex reasoning involved code translation task, which resulted in more compilation errors and low execution scores. Also, one of the main problems common in all previous approaches is the loss of code structure, which is inconsistency in declarations of classes and functions between Target and source language codes.

The quality of translation can be improved with approaches based on chain-of-thought (CoT) prompting. As for complex reasoning tasks and for maintaining structural information, Chain-of-Thought (CoT) prompting with In-Context Learning (ICL) has emerged as a promising technique in Natural Language Processing (NLP).

5.4.1 Paradigms of CoT Prompting

When zero-shot and few-shot prompting is insufficient, it suggests that the model's learned knowledge is inadequate for complex tasks that require robust reasoning. In such cases, it is advisable to consider fine-tuning the models or exploring more advanced prompting techniques.

Chain-of-Thought (CoT) prompting involves decomposing the problem into sequential reasoning steps and demonstrating this approach to the model to solve the task. CoT prompting can be categorized into three main paradigms. One paradigm involves adding a single prompt like "Let's think step by step" after the test question to facilitate reasoning chains in LLMs (Kojima et al., 2022 [KGR+22]). This method, known as Zero-Shot-CoT, is task-agnostic and does not require input-output demonstrations. Despite its simplicity, Zero-Shot-CoT has proven effective for zero-shot reasoning tasks. The other paradigm is few-shot prompting with manually designed reasoning demonstrations (Wei et al., 2022 [WWS+22]). In this Manual-CoT approach, each demonstration includes a question and a reasoning chain consisting of intermediate reasoning steps and the expected answer. While effective, this manual process can be labor-intensive and may not always yield optimal results. To address this, Zhang et al., 2022 [ZZLS22] introduced an Auto-CoT paradigm, where LLMs use the "Let's think step by step" prompt to automatically generate reasoning chains for demonstrations. Auto-CoT has reduced the need for manual efforts and consistently matches or exceeds the performance of other CoT paradigms (Figure 5.6) by automatically constructing demonstrations with questions and reasoning chains.

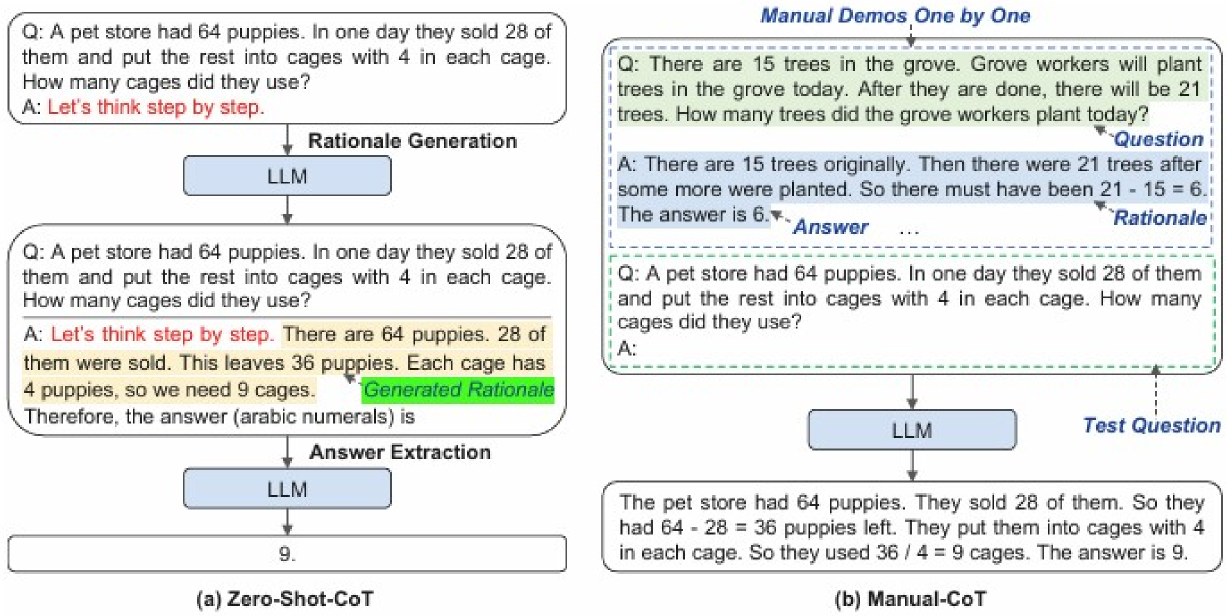

Figure 5.6: Zero-Shot-CoT [KGR+22] (utilizing the “Let’s think step by step” prompt) and Manual-CoT [WWS+22] (utilizing manually crafted demonstrations) illustrated with example inputs and outputs of an LLM. Figure-[ZZLS22]

5.4.2 Component-Wise Translation (CompWiseTrans)

(In practice, manual-CoT (Wei et al., 2022 [WWS+22]) has generally shown stronger performance than Zero-Shot-CoT (Kojima et al., 2022 [KGR+22]), but its effectiveness relies on the quality of manually crafted demonstrations. While Auto-CoT (Zhang et al., 2022 [ZZLS22]) attempts to enhance diversity during demonstration construction, it faces challenges in code translation tasks due to the complexity of programming languages. Therefore, we propose a Component Wise Translation (CompWiseTrans) approach, combining structured code translation-specific few-shot CoT reasoning with the "Let’s think step by step" prompt. The complex source language programs may have many components like multiple classes, nested classes, multiple functions in each class, "main" function, library import requirements, etc. Here in this CoT reasoning, we are doing component-wise translation of a complex source language program in a stepwise manner, and later, we combine all steps to get a final translation of the whole source language program. This method aims to address the unique challenges of code translation by ensuring the retention of structural and logical integrity during the translation process.

More Details

The general prompt template for the Component Wise Translation (CompWiseTrans) approach is shown in 5.4.2 box. Here below is an explanation of the different parts of this general prompt template.

Source code and sample Input/Output: For translation of a source language program, we considered source code and sample Input/Output in the context. Here, the source code is raw source code. We have provided a sample Input/Output in the context so that there is some additional information on the Input/Output format, as we can't rely solely on the Input/output handling mechanism in source Java code, which can read input either token-wise or line-wise. Also, we have taken sample Input, which has the smallest file size when read as string, due to minimize prompt length.

Class Components' Translation Steps: Given the context, in the CoT reasoning chain, we first wanted to identify the number of classes defined in the Java code that constitute the number of components in the source Java code. Then, to simplify the translation of complex source code, we translate these components in a stepwise manner. So, in the next few steps, we did a translation of a separate class component and its functions.

Main Class Translation Step: We translated the main class after all the other classes. This is because all other components might be used inside of the "main" function, and there is a need for their syntactical and semantical meaning understanding to use them.

Library Imports Step: After translating these class components, LLM got the idea of the related import requirements. So, there is one more Step related to library imports.

Final Combination Step: The final Step was to combine all steps to get the whole Python Code. Here, LLM would combine separately translated components with appropriate syntactical and semantical correctness.

Demonstrations for in-context learning: For this few-shot-CoT-based approach, we have manually crafted two demonstrations from the AVATAR dataset (Ahmad et al., 2021 [ATCC21]). Given context-length related limitations, we have only taken two demonstrations for in-context learning. One demonstration is a complex program involving multiple classes, functions, etc. This demonstration is indicating a program may involve diverse and complex components. Another demonstration is a simple program with a "Main" class and a "main" function to help the model understand CoT reasoning steps easily.

1. General Prompt template J2P_CompWiseTrans

1. Whole Java Code:

```
####################
// <<Source_Code in Java>>
```

```
import <<requirements>>
public class Main {
    //<<class variables declarations>>
    public static void main ( String [ ] args ){
        //<<"main" function definition>>
    }
    static class CustomScannerClass {
        //<<class variables declarations>>
        String next ( ){
            //<<"next" function definition>>
        }
        long otherFunctions ( ){
            //<<"otherFunctions" function definition>>
        }
    }
    static class HelperClasses {
        //<<class variables declarations>>
        HelperClasses(){
            //<<class variables initialization and other statements>>
        }
        String helper ( ){
            //<<"helper" function definition>>
        }
    }
}
```

1. Sample Input:

```
// <<Sample Input Test case>>
```

1. Expected Output:

```
// <<Expected Output of given Sample Input Test case>>
```

1. Steps: Let's think step by step.
Firstly find classes defined in the Java code that are- CustomScannerClass, HelperClasses, Main. Let's translate all classes definition and its functions step by step. Here is order in which steps to be followed- First translate all classes instead of "Main" class, then translate "Main" class and then "library imports" step.
Let's translate CustomScannerClass class. Functions present in CustomScannerClass class- next, otherFunctions. Translation of CustomScannerClass class and its functions from Java to Python-

Step 1: Translation of "CustomScannerClass" class and its functions -
Python Code:

```
class CustomScannerClass:
    def __init__(self):
        #<<some initializing statements>>
    def next(self):
        #<<"next" function definition>>
```

```
    def otherFunctions(self):
        #<<"otherFunctions" function definition>>
```

```
Similar to above class, there is also more translation Steps of
other classes, like here <<Step 2 of  HelperClasses..>>
```

Let's translate Main class. Functions present in Main class- helper_main, main. Consider above sample input and output format information while translating "main" function to handle input related code properly. Translation of Main class and its functions from Java to Python-

Step 3: Translation of "Main" class and its functions -
Python Code:

```
#Definitions of functions in Main class
def helper_main(self):
    #<<"next" function definition>>
def main():
    #<<"main" function definition>>
```

Considering all steps, add required library imports in Python Code-

Step 4: Library imports -
Python Code:

```
# <<required library imports>>
```

Finally Combined all steps- Step 1, Step 2, Step 3, Step 4 to get syntactically correct whole Python Code-
1. Whole Python Code:

```
# <<whole combined Python Code>>
```

```
2. Whole Java Code:
####################
<<2nd demonstration example>>
Similarly there can be other demonstration example for the
In-context learning (ICL). In our approaches, we have used two
demonstration examples.
```

3. Whole Java Code:

```
####################
// <<$Source_Code of new Test program in Java>>
```

3. Sample Input:

```
// <<Sample Input Test case for this new Test program>>
```

3. Expected Output:

```
// <<Expected Output of this Sample Input Test case>>
```

3. Steps: Let's think step by step.

```
<<LLM will generate it >>
Now in this chain-of-thought (CoT) approach, LLM will follow
intermediate reasoning  steps to finally translate this Test
Program like in demonstration examples.
```

For more details about the implementation of this approach's prompt template, see 10.1 in the Appendix

Discussions

This approach is an initial step towards formulating a method that emphasizes maintaining structural and logical information for accurate code translation. We observed that both GPT-4 and StarCoder models effectively followed intermediate CoT steps, identifying components and their translation steps, resulting in improved translations that preserve structural information. The CompWiseTrans results (Table 7.4) fall short in terms of functional correctness compared to the few-shot based approach. This shortfall is primarily due to an increase in compilation errors caused by longer prompt sizes, leading to the generation of truncated code. Significant errors also arise from the misuse of keywords and built-in functions as variables, such as in, map, and range. Unlike the powerful GPT-4 model, StarCoder frequently hallucinates during the "Library Imports" step, failing to accurately identify necessary imports.

A notable shortcoming in the formulation of the approach is the inadequate handling of dependencies among components, which affects the translation order. Currently, the approach only explicitly manages the translation of the main function in the last after obtaining the context of other components' signatures and definitions. There is a need to develop a better mechanism to manage component dependencies, ensuring that the translation of components follows the appropriate order. This would help LLMs understand the involved components and their signatures for structure-consistent code translation.

5.5 CodeTransCoT

Our primary approach is concise and focused on maintaining structural and logical information for accurate code translation. Instead of considering the whole information of source code components, this approach represents the source code by providing skeletal structural information of components, that is, class and function declarations. It uses a high-level structured representation of the code with Chain-of-Thought (CoT) reasoning.

This method combines structure-consistent CoT reasoning, automatic reasoning, and few-shot learning. Its prompt template is similar to the CompWiseTrans approach, incorporating few-shot CoT and stepwise prompting instructions from Zero-Shot-CoT [KGR+22].

5.5.1 Java to Python Translations

The general prompt template for the Code Translation CoT (CodeTransCoT) approach for the Java to Python Translations is shown in 5.5.1 box. Here below is an explanation of the different parts of this general prompt template.

Source code and sample Input/Output: For translation of a source language program, we considered source code and sample Input/Output in the context. Here, the source code is raw source code. We have provided a sample Input/Output in the context so that there is some additional information on the Input/Output format, as we can't rely solely on the Input/output handling mechanism in source Java code, which can read input either token-wise or line-wise. Also, we have taken sample Input, which has the smallest file size when read as string, due to minimize prompt length.

Skeletal Structure of Java program: Given the context, in the CoT reasoning chain, we first wanted to identify the skeletal structure of the source Java code.

Skeletal Structure of Python program: Given the skeletal structure of the source Java code, the LLM would generate an equivalent skeletal structure of the target Python code. This is done to ensure the preservation of the same structural and logical information during translation of the source Java code.

Final Completion Step: The final step was to fully complete the translation by considering all the steps to get the translated Python Code. Here, LLM should consider appropriate information, such as keeping track of library imports and input/output format information, to ensure the generated code is syntactically and semantically correct.

Demonstrations for in-context learning: Sure, here is the revised text: For this few-shot CoT-based approach, we manually created two demonstrations using the AVATAR dataset (Ahmad et al., 2021). Due to limitations related to context length, we only used two demonstrations for in-context learning. One demonstration is a complex program that includes multiple classes, functions, etc., indicating that a program may involve diverse and complex components. The other demonstration is a simple program with a "Main" class and a "main" function, designed to help the model understand CoT reasoning steps easily.

2. General Prompt template J2P_CodeTransCoT

###$####

1. Java Code:

```java
// <<Source_Code in Java>>
import <<requirements>>
public class Main {
    //<<class variables declarations>>
    public static void main ( String [ ] args ){
```

```
        //<<"main" function definition>>
    }
    static class CustomScannerClass {
        //<<class variables declarations>>
        String next ( ){
            //<<"next" function definition>>
        }
        long otherFunctions ( ){
            //<<"otherFunctions" function definition>>
        }
    }
    static class HelperClasses {
        //<<class variables declarations>>
        HelperClasses(){
            //<<class variables initialization and other statements>>
        }
        String helper ( ){
            //<<"helper" function definition>>
        }
    }
}
```

1. Sample Input:

```
// <<Sample Input Test case>>
```

1. Expected Output:

```
// <<Expected Output of given Sample Input Test case>>
```

1. Steps: Let's think step by step.
Step 1: First of all identify classes and functions declarations present in the above Java Code-
Identified classes and functions declarations in the Java code:

```
public class Main {
    public static void main ( String [ ] args );
    static class CustomScannerClass {
        String next ( );
        long otherFunctions ( );
    }
    static class HelperClasses {
        HelperClasses();
        String helper ( );
    }
}
```

Step 2: Considering Step 1, classes and functions declarations should be presented in the corresponding Python Code-
Identified classes and functions declarations in the Python code:

```
Class Main:
```

```
    def main(self):

    class CustomScannerClass:
        def __init__(self):

        def next(self):

        def otherFunctions(self):

    class HelperClasses:
        def __init__(self):

        def helper(self):
```

Step 3: Complete translation of the above Java code to Python code by considering identified classes and functions declarations in Step 2. Consider above sample input and output format information to handle input and output related code properly in the generated Python code. While generating the Python code, keep track of required library imports to be added in the Python Code. Make sure your generated code is syntactically correct-

1. Python Code:

```
import sys
Class Main:
    def main(self):
        #<<"main" function definition>>

    class CustomScannerClass:
        def __init__(self):
            #<<some initializing statements>>
        def next(self):
            #<<"next" function definition>>
        def otherFunctions(self):
            #<<"otherFunctions" function definition>>

    class HelperClasses:
        def __init__(self):
            #<<some initializing statements>>
        def helper(self):
            #<<"helper" function definition>>

if __name__ == "__main__":
    obj = Main()
    obj.main()

###$###
2. Java Code:
<<2nd demonstration example>>
Similarly there can be other demonstration example for the In-context learning (ICL).
In our approaches, we have used two demonstration examples.
```

3. Java Code:

```
###$####
// <<$Source_Code of new Test program in Java>>
```

3. Sample Input:

```
// <<Sample Input Test case for this new Test program>>
```

3. Expected Output:

```
// <<Expected Output of this Sample Input Test case>>
```

3. Steps: Let's think step by step.

```
<<LLM will generate it >>
Now in this chain-of-thought (CoT) approach, LLM will follow intermediate reasoning
steps to finally translate this Test Program like in demonstration examples.
```

For more details about the implementation of this approach's prompt template for Java to Python translation, see 10.2 in the Appendix

5.5.2 Python to Java Translations

The general prompt template for the Code Translation CoT (CodeTransCoT) approach for the Python to Java Translations is shown in 5.5.2 box. Here below is an explanation of the different parts of this general prompt template.

Source code and sample Input/Output: For translation of a source language program, we considered source code and sample Input/Output in the context. Here, the source code is raw source code. We have provided a sample Input/Output in the context so that there is some additional information on the Input/Output format, as we can't rely solely on the Input/output handling mechanism in source Java code, which can read input either token-wise or line-wise. Also, we have taken sample Input, which has the smallest file size when read as string, due to minimize prompt length.

Global variables Identification: Information such as global variables, local and global scope of variables, etc., plays a critical role in standard Python programs. First of all, we identified Global variables involved in the given source Python code. They can occur in assignments involving statements in the global scope, can be declared with the "global" keyword, or can be associated with "For" loops. So here, the LLM would extract all possible Global variables.

Skeletal Structure of Python program: Given the context, in the CoT reasoning chain, we first wanted to identify the skeletal structure of the source Java code.

Skeletal Structure of Java program: The LLM generates an equivalent skeletal structure of the target Java code based on the skeletal structure of the source Python code. This is done to preserve the same structural and logical information during the translation process.

Final Completion Step: The final step was to fully complete the translation by considering all the steps to get the translated Java Code. Here, LLM should consider appropriate information, such as keeping track of library imports and input/output format information, to ensure the generated code is syntactically and semantically correct.

Demonstrations for in-context learning: Sure, here is the revised text: For this few-shot CoT-based approach, we manually created two demonstrations using the AVATAR dataset (Ahmad et al., 2021). Due to limitations related to context length, we only used two demonstrations for in-context learning. One demonstration is a complex program that includes a function, some statements in the global scope, challenging data structure class and its multiple functions, etc., indicating that a program may involve diverse and complex components. The other demonstration is a simple program with a function, some statements in the global scope, and a "main" function, which is designed to help the model understand CoT reasoning steps easily.

3. General Prompt template P2J_CodeTransCoT

###$####

1. Python Code:

```python
# <<Source_Code in Python>>
import <<requirements>>
var0 = value

def function1():
    local_var =var0 + 5
    #<<"function1" function’s complete definition>>

for i in range(<<range_constraints>>):
    var1 = value1
    var2 = value2

class HelperClass:
    def __init__(self):
        #<<some initializing statements>>
    def helper(self):
        global var1
        var1 = value3
        var2 = value4

        global new_var
        new_var = "new_global"
```

```
        #<<"helper" function’s complete definition>>

def main():
    val = function1()
    print(val)
    #<<"main" function complete definition>>

if __name__=="__main__":
    helper_obj = HelperClass()
    helper_obj.helper()
    print("var1 value-", var1)
    print("var2 value-", var2)
    print("new_var value-", new_var)
    main()
```

1. Sample Input:

```
# <<Sample Input Test case>>
```

1. Expected Output:

```
# <<Expected Output of given Sample Input Test case>>
```

1. Steps: Let’s think step by step.
Step 1: First of all identify global variables and their types. Global variables are variables that are not inside scope of function and class declarations, present in the above Python Code-
Global variables:

```
var0, i, var1, var2, new_var, helper_obj,<<other possible Global variables>>
```

Step 2: Identify classes and functions declarations present in the above Python Code-
Identified classes and functions declarations in the Python code:

```
def function1():

class HelperClass:
    def __init__(self):
        #<<some initializing statements>>
    def helper(self):

def main():

if __name__=="__main__":
```

Step 3: Considering Step 1 and Step 2, the corresponding Java Code should have declarations of these classes, functions and Global variables -
Identified classes, functions and Global variables declarations in the Java code:

```
public class Main {
    static int var0;
    static int i;
    static int var1;
```

```
    static int var2;
    static String new_var;
    static Main.HelperClass helper_obj;

    public static int function1 ( );
    static class HelperClass {
        //<<class variables declarations>>
        HelperClass ();
        String helper ( );
    }
    public static void main ( String [ ] args );
}
```

Step 4: Complete translation of the above Python code to Java code by considering identified classes, functions and Global variables declarations in Step 3. Consider above sample input and output format information to handle input and output related code properly in the generated Java code. While generating the Java code, keep track of required library imports to be added in the Java Code. Make sure your generated code is syntactically correct-

1. Java Code:

```
import <<requirements>>;
public class Main {
    static int var0;
    static int i;
    static int var1;
    static int var2;
    static String new_var;
    static Main.HelperClass helper_obj;

    public static int function1 ( ){
        int local_var =var0 +5 ;
        //<<"function1" function complete definition>>
    }
    static class HelperClass {
        //<<class variables declarations>>
        HelperClass (){
        //<<class variables initialization and other statements>>
        }
        String helper ( ){
            Main.var1 = value3;
            int var2 = value4;

            Main.new_var ="new_global";
            //<<"helper" function complete definition>>
        }
    }
    public static void main ( String [ ] args ){
        var0 = value;
```

```
        for (<<range_constraints with global variable i >>){
            var1 = value1;
            var2 = value2;
        }
        helper_obj =new Main.HelperClass();
        helper_obj.helper();
        System.out.println("var1 value- " + var1);
        System.out.println("var2 value- " + var2);
        System.out.println("new_var value- " + new_var);
        int val =function1();
        System.out.println(val);
        //<<"main" function complete definition>>
    }
}


###$####
2. Python Code:
<<2nd demonstration example>>
Similarly there can be other demonstration example for the In-context learning (ICL).
In our approaches, we have used two demonstration examples.
```

3. Python Code:

```
###$####
// <<$Source_Code of new Test program in Python>>
```

3. Sample Input:

```
// <<Sample Input Test case for this new Test program>>
```

3. Expected Output:

```
// <<Expected Output of this Sample Input Test case>>
```

3. Steps: Let's think step by step.

```
<<LLM will generate from here >>
Now in this chain-of-thought (CoT) approach, LLM will follow intermediate reasoning
steps to finally translate this Test Program like in demonstration examples.
```

For more details about the implementation of this approach's prompt template for Java to Python translation, see 10.3 in the Appendix

*more things and explanations to be added soon..

Chapter 6

Evaluation Metrics

Large Language Models (LLMs) have shown exceptional proficiency in generating text that closely resembles human writing, and their applications now extend to code translation. However, evaluating the quality of the generated code presents a unique set of challenges. Traditional metrics like the BLEU score are match-based metrics requiring a reference solution and are also unable to capture complex space of functionally equivalent translations.

This chapter addresses these challenges and introduces the evaluation metrics used. We use the unbiased estimator pass@k [CTJ+21] to evaluate our translated code for functional correctness with the help of the test cases. We also introduce a new metric, **Quality Metric**, to assess structural quality correctness that aligns with our CoT approach.

6.1 Pass@k estimator

The pass@k metric is defined as the probability that at least one of the top k-generated code samples for a problem passes the unit tests.

For a given task, $n \geq k$ samples are sampled. Let the number of correct samples be denoted by c, among the n samples, which pass all the tests. The probability of sampling only incorrect generated code as top-k samples is $\binom{n-c}{k}/\binom{n}{k}$.

Then pass@k is defined as :

$$pass@k = \mathbb{E}_{problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

Direct computation of this estimator is numerically unstable. We use the numerically stable numpy implementation introduced by [CTJ+21].

For a special case when $k = 1$, pass@1 evaluates down to:

$$\begin{aligned} pass@1 &= \mathbb{E}_{problems}\left[1 - \frac{n-c}{n}\right] \\ &= \mathbb{E}_{problems}\left[\frac{c}{n}\right] \end{aligned}$$

But if greedy decoding is used, the number of correct samples c is either $c = n$ or $c = 0$. Let P be the set of all the programs in the target language which are functionally equivalent to the task at hand and x is the generated program. Then we define pass@1*,

$$pass@1^* = \mathbb{E}_{problems}\left[\mathbb{1}\{I_P(x) = 1\}\right]$$

where $I_P(x)$ is an indicator function.

6.2 Quality Metric

Given the reliance of our Cot method on identifying structural components in the source language and converting them into an analogous structure in the target language, it is essential to assess the quality of the translation to guarantee and verify the reasoning steps and make sure we have indeed generated what we have expected.

Hence, we introduce a Structural Quality assessment measure that evaluates the similarity between Static call graphs derived from both the source code and the translated code by employing graph edit distance as the evaluation criterion

```java
// Java Code
public class Main(){
    Main() {
        //some code
    }
    long calc() {
        return dfs(arg1, arg2);
    }
    boolean isOK(arg1) {
        //some code
    }
    long dfs(arg1, arg2) {
        //some code
        isOK(arg1);
        dfs(arg1, arg2);
    }
    public static void main(String[] args) {
        Main ins = new Main();
        System.out.println(ins.calc());
    }
}
```

```python
# Python Code
class Main:

    def __init__(self):
        #some code

    def calc(self):
        return self.dfs(arg1, arg2)

    def isOK(self, arg1):
        #some code

    def dfs(self, arg1, arg2):
        #some code
        self.isOK(arg1):
        self.dfs(arg1, arg2)
        #some code

if __name__ == "__main__":
    ins = Main()
    ins.calc()
```

Figure 6.1: Comparison of structural representations between a Java source code and its Python translation using the Vanilla approach through the StarCoder LLM.

6.2.1 Call Graph

A call graph is a control-flow graph that depicts the calling relationships between subroutines in a program. In this graph, each node represents a procedure, and each edge (f, g) signifies that procedure f calls procedure g. Consequently, a cycle in the graph indicates recursive procedure calls.

For example, consider two functionally equivalent programs in two languages (Java and Python), as shown in Figure 6.1. The Python program was obtained by translation using vanilla prompting technique on the source Java Code with the help of StarCoder[LAZ+23]. The corresponding Call Graphs for the two programs are given in Figure 6.2.

It is evident that there needs to be a better match in the call graphs of the two programs. There needs to be a `Main.main` function in the Python program, which is supposed to call `Main.calc`. Even though the programs run correctly without errors, the source code's function calls and definition structure are not preserved in the generated code.

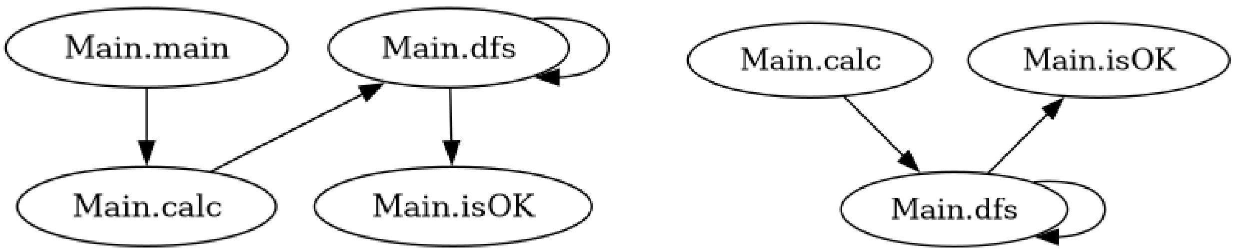

Figure 6.2: Corresponding Java call graph (left) and Python call graph (right) of the Code in 6.1

6.2.2 Graph Edit Distance

To compare the static call graphs in Figure 6.2, we need a function which takes inputs as two graphs and returns a metric indicating the similarity between the two graphs.

Generally, given a set of graph edit operations (also known as elementary graph operations), the graph edit distance between two graphs g_1 and g_2 , denoted as $GED(g_1, g_2)$ can be defined as,

$$GED(g_1, g_2) = \min_{(e_1,\cdots,e_k)\in\mathcal{P}(g_1,g_2)} \sum_{i=1}^{k} c(e)$$

where $\mathcal{P}(g_1, g_2)$ denotes the set of edit paths transforming g_1 into (a graph isomorphic to) g_2 and $c(e) \geq 0$ is the cost of each graph edit operation e. For simplicity, We use a cost of 1 for each operation in all our experiments.

The set of elementary graph edit operations includes:

- Vertex insertion: Introducing a new labeled vertex to the graph.
- Vertex deletion: Removing a vertex from the graph.

- Edge insertion: Adding a new edge between a pair of vertices.
- Edge deletion: Removing an edge between a pair of vertices.

The elementary operations correcting the Python call graph in Figure 6.2 can be visualized in Figure 6.3. The reference program is also given in Figure 6.4. We add a new node `Main.main` and a new edge connecting `Main.main` and `Main.calc` giving us a cost of 2. It can be proved that this is the minimum-cost edit path transforming the Python call graph.

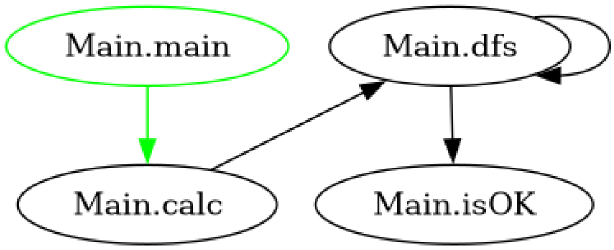

Figure 6.3: Correcting the call graph

```python
# Python Code
class Main:

    def __init__(self):
        #some code

    def calc(self):
        return self.dfs(arg1, arg2)

    def isOK(self, arg1):
        #some code

    def dfs(self, arg1, arg2):
        #some code
        self.isOK(arg1):
        self.dfs(arg1, arg2)
        #some code

    def main(self):
        #some code
        self.calc()

if __name__ == "__main__":
    ins = Main()
    ins.main()
```

Figure 6.4: Reference program for the source Java program in Figure 6.2

Creating a reference Java program while translating from Python to Java is a bit tricky. For example, the reference for the program given in Figure 4.7 is shown in Figure 6.5. For more details on the implementation refer chapter 7

```java
import java.util.*;
import java.io.*;
import java.util.stream.Collectors;

public class YYY{
    static int a;

    public static void func(int a) {
        int digits[] = new int[100];
        int temp = a;
        int i = 0;
        while (temp!= 0) {
            digits[i] = temp % 10;
            temp = temp / 10;
            i++;
        }
        int result = Arrays.stream(digits).sum();
        if (result == 1) {
            System.out.println("10");
        } else {
            System.out.println(result);
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        a = scanner.nextInt();
        func(a);
    }
}
```

Figure 6.5: Reference Java program for quality analysis of Python code in figure 4.6

Chapter 7

Experiments

This chapter presents a detailed account of the experimental setup, methodologies, and procedures employed to answer the research question. Subsequent subsections outline the models used, the evaluation results of the approaches mentioned in preceding chapters, and the metrics mentioned in (chapter 6). We have used 1x A100-Nvidia GPUs to perform all experiments unless stated otherwise.

7.1 Models

StarCoder [LAZ+23]: A pre-trained LLM of 15.5B parameters with 8K context length and infilling capabilities. It is trained on 1 trillion tokens sourced from The Stack [KLBA+22], a comprehensive collection of permissively licensed GitHub repositories, and further fine-tuned with 35 billion Python tokens.

GPT-4 [OAA+23]: We use the OpenAI API to access the gpt-4 model, which has a context window of 8,192 tokens and uses training data up to Sep 2021.

Code Llama [RGG+24]: An open-source large language model for code derived from Llama 2, offering state-of-the-art performance, infilling capabilities, and support for large input contexts. Specifically, we use the Code Llama-Instruct 13B model, which has been instructionally fine-tuned after being pre-trained on 500 billion tokens from a code-centric dataset.

IBM-Granite Code Models [MSZ+24]: A family of decoder-only code models designed for code generation tasks, trained on a diverse corpus encompassing code written in 116 programming languages. It consists of models ranging in size from 3 to 34 billion parameters. We use the Granite-8B-Code-Instruct and Granite-20B-Code-Instruct for our experiments, where the Code-Instruct models is fine-tuned using a combination of Git commits paired with human instructions and open-source synthetically generated code instruction datasets.

7.2 Datasets

CodeNet [PKJ+21] is a comprehensive collection of code samples accompanied by extensive metadata. It is sourced from two online judge websites: AIZU and AtCoder, which present programming challenges through courses and contests. CodeNet encompasses 13,916,868 submissions across 4,053 unique problems. The submissions are written in 55 different programming languages, with the majority (95%) utilizing C++, Python, Java, C, Ruby, and C# We randomly sample 250 problems of Java and Python among the 4053 problems and further sample one correctly submitted program for each problem along with its testcase.

Avatar [ATCC21] (jAVA-pyThon progrAm tRanslation) includes solutions written in Java and Python for 9,515 programming problems gathered from competitive programming sites like Codeforces, Google Code Jam, AIZU Online Judge, AtCoder, and from online platforms such as LeetCode, GeeksforGeeks, Project Euler, alongside contributions from open-source repositories. AVATAR contains unit tests for 250 evaluation examples to perform functional accuracy evaluation of the translation models. The unit tests are gathered from publicly available test cases published by AtCoder.

7.3 Results

In the Results Section, firstly, we present the results of Pass@k metrics for k in $\{1^*, 1, 5\}$.

7.3.1 Pass@1* results - Error types and their analysis

Table 7.1: Pass @1* results for Vanilla and CodeTransCoT techniques on different LLMs

Dataset	Source	Target	Approach	Models			
				StarCoder	Code Llama	Granite-8B	Granite-20B
Avatar	Java	Python	Vanilla	18.1	29.2	43.8	15.2
			CodeTransCoT	**45.4**	**38.4**	**44.6**	**39.75**
	Python	Java	Vanilla	6.4	22.4	28.4	**38.8**
			CodeTransCoT	**28.4**	**26.8**	**32.4**	33.2
CodeNet	Java	Python	Vanilla	29.5	**48.5**	49	29.5
			CodeTransCoT	**52**	44.5	**55**	**38.5**
	Python	Java	Vanilla	9	42	56.49	**69.5**
			CodeTransCoT	**53.5**	**62.5**	**66.5**	60

The Pass@1* metric (refer 6.1) results are shown in Fig .7.1. The CodeTransCoT prompt method demonstrates significant improvements across almost all the models, outperforming the vanilla prompt technique. StarCoder[LAZ+23], in particular, performs exceptionally well, giving at least 22% improvement across all settings. For example, translation from Java to Python on the Avatar[ATCC21] dataset with vanilla prompting technique gives us only 18% while using CodeTransCoT boosts it upto 45.4% giving a gain of 27.3%.

We can also see great improvement in Python to Java translations of CodeNet[PKJ+21] dataset where all the models except for Granite-20B shows substantial gains, with CodeLlama giving 62.5% compared to 42% on the Vanilla prompting technique. There is a slight drop in the performance about 5-9% when using Granite-20B and translating from Python to Java.

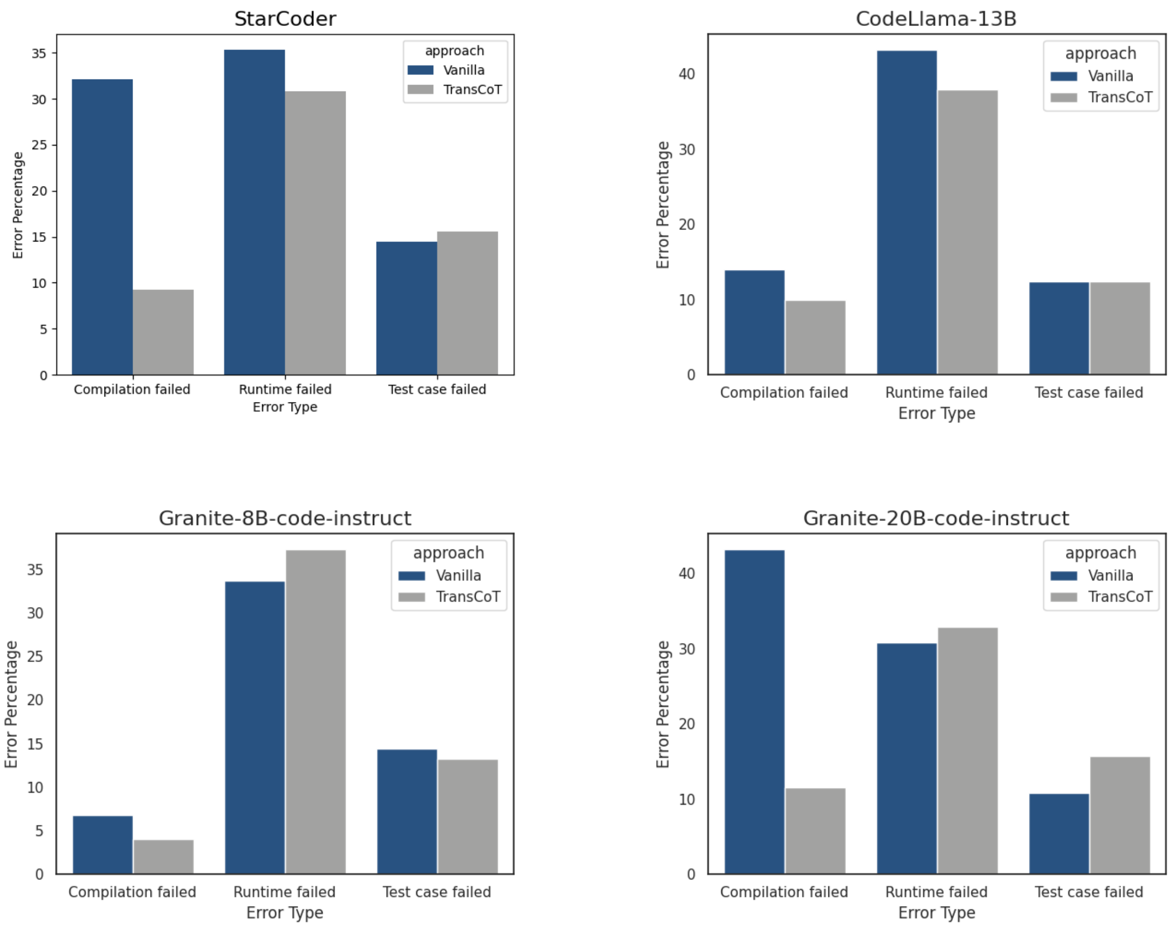

Figure 7.1: Comparison of different error types of CodeTransCoT approach vs vanilla approach. This is was obtained on Avatar Dataset and translation was from Java to Python

The main reason for such an overall improvement using the CodeTransCoT method is due to the huge reduction in compilation failed rate as shown in Figure 7.1, especially in the cases where the compilation failed rates are high (>30%). The reduction is at least 25%. Most of the corrected samples in compilation failed types end up in either other error categories or the correct category. The same can also be for the translations from Python to Java as there is a decrease in Compilation failed category of atleast 20% when its error rate is high as shown in Figure 7.2.

We have observed that Granite-20B performs better using the vanilla approach while translating from Python to Java (Fig 7.2). However, going through IBM-Granite Code Model[MSZ+24] results for the Code Lingua dataset (formed by combining samples from

Avatar and CodeNet), we see that Granite-8b outperforms Granite-20b on many occasions. We do not have enough evidence to support this, but we suspect this might be due to how the model was pre-trained.

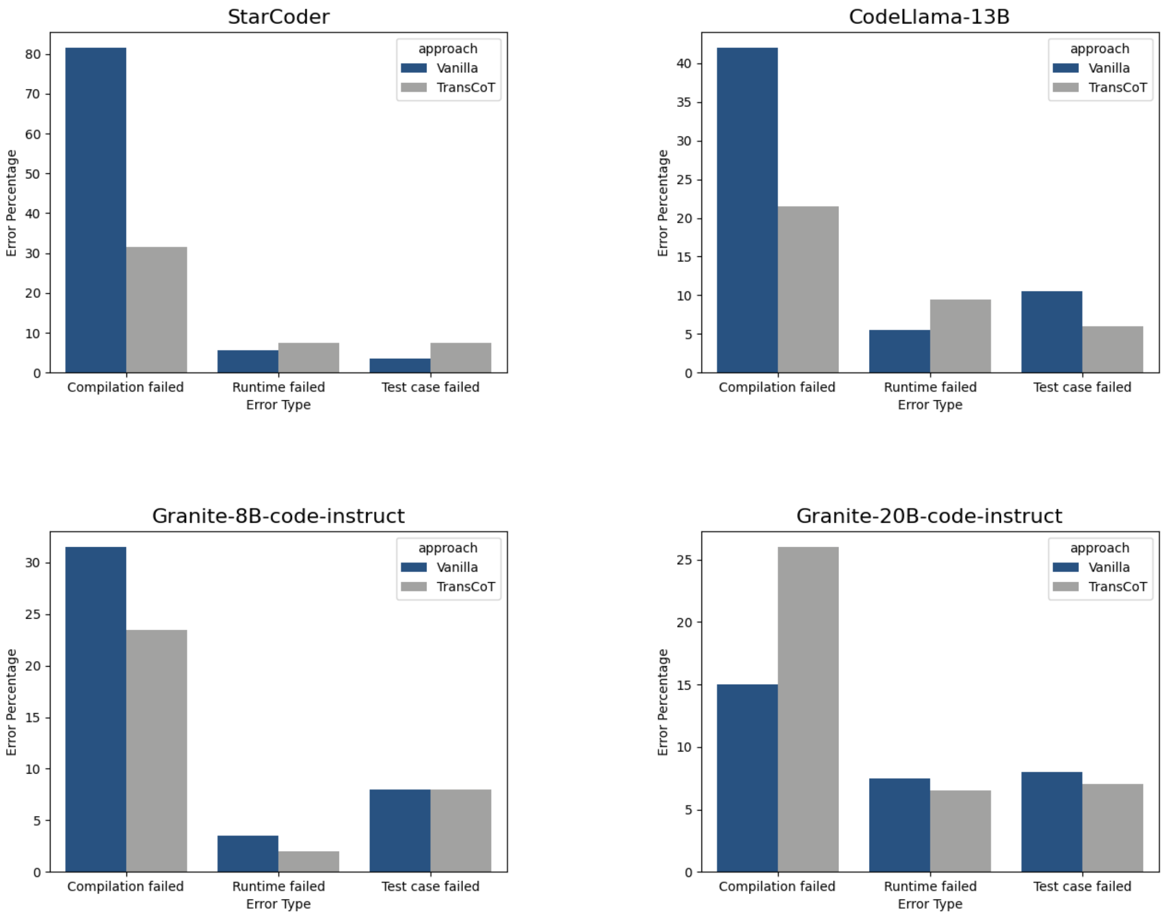

Figure 7.2: Comparison of different error types of CodeTransCoT approach vs Vanilla approach while translating from Python to Java using the CodeNet dataset

7.3.2 Pass@1 and Pass@5 results for various models and datasets

We also report the Pass@1 and Pass@5 results for the all pairs of datasets and models in Table 7.2 and 7.3. For computation of both the metrics, we use $n = 10$ due to resource constraints as the models are large and it is easy to run out of memory and used a temperature setting of 0.2 as the generations are more deterministic and task-relevant for Open-source models in this setting. As intended, CodeTransCoT consistently outperforms the Vanilla approach across various datasets and model configurations. For instance, Pass@1 score for the CodeNet dataset, the Java to Python translation sees a remarkable increase from 23 (Vanilla) to 46.3 (CodeTransCoT) with StarCoder. Even for Pass@5 score, CodeTransCoT achieves a score of 57.9 with StarCoder, compared to the Vanilla score of 37.2 indicating

CodeTransCoT is more effective at identifying correct solutions within the top 5 candidates. The trends are very similar to the Pass@1* and Pass@1 results.

Table 7.2: Pass @1 results for Vanilla and CodeTransCoT techniques on different LLMs

Dataset	Source	Target	Approach	Models			
				StarCoder	Code Llama	Granite-8B	Granite-20B
Avatar	Java	Python	Vanilla	14.4	28.4	38.8	15
			CodeTransCoT	**41.2**	**35.7**	**42**	**40.1**
	Python	Java	Vanilla	5.9	23.7	29.3	**36.4**
			CodeTransCoT	**27.1**	**26.8**	**31.1**	32.3
CodeNet	Java	Python	Vanilla	23	43.2	43.9	26
			CodeTransCoT	**46.3**	**45.5**	**48.2**	**34.4**
	Python	Java	Vanilla	8.1	41.4	**61.2**	**70.1**
			CodeTransCoT	**44.5**	**63.4**	59.2	47.2

Table 7.3: Pass @5 results for Vanilla and CodeTransCoT techniques on different LLMs

Dataset	Source	Target	Approach	Models			
				StarCoder	Code Llama	Granite-8B	Granite-20B
Avatar	Java	Python	Vanilla	27.5	**45.5**	52.4	25.9
			CodeTransCoT	**53.2**	45.1	**57.9**	**51.3**
	Python	Java	Vanilla	9	33	35.6	**43.7**
			CodeTransCoT	**35.9**	**34.6**	**38.2**	33.5
CodeNet	Java	Python	Vanilla	37.2	**57.4**	55.6	34.3
			CodeTransCoT	**57.9**	56.1	**64.3**	**46.1**
	Python	Java	Vanilla	13.3	44.2	**75.9**	**79**
			CodeTransCoT	**55.2**	**64.5**	75.1	64.1

7.3.3 Comparison of different Approaches

Our methodologies have been progressively developed to articulate and formulate the primary approach, CodeTransCoT. Here are the experimental results shown in Table 7.4 for performance comparison among these approaches. These results are for Java to Python Translations using StarCoder model and Avatar dataset. To ensure a fair comparison, we maintained consistent generation settings across all approaches. In particular, for the used sampling-based generation strategy, we kept the temperature as 0.2 and other parameters such as *top_k* and *top_p* with default values. As previously discussed, a temperature of 0.2 is optimal for open-source models.

In the comparison table 7.4, we can also observe a shift in the number of bugs from one category to another. A detailed discussion of these approaches and their comparative analysis during the development process can be found in Methodology Chapter 5. Each section dedicated to the individual approaches includes a discussion about the approach's effectiveness, a comparison with preceding methods, a broad error analysis, and an exploration of its limitations.

Table 7.4: Results of Different approaches while translating from Java to Python using StarCoder and Avatar Dataset

Prompt Approach	**Test passed**	**Test failed**	**Runtime failed**	**Compilation failed**
Vanilla	17.67	8.4	27.6	46
Explanation based	22.08	11.2	26.4	40
Pseudocode based	24.8	9.6	31.2	34
Few-shots/ICL	30.12	14	52	3.6
CompWiseTrans	26.1	10.4	42.8	20.7
CodeTransCoT	43.25	20	32.4	4.8

7.3.4 Evaluation results on GPT-4

Table 7.5: Results on GPT-4 on Avatar dataset

Method	Accuracy
Vanilla	75.2
CompWiseTrans	66
CodeTransCoT	69.4

Table 7.5 presents the results of evaluating GPT-4 using both Vanilla and Chain-of-Thought (CoT) approaches. The performance dropped to 66% when using CompWiseTrans. However, incorporating CodeTransCoT improved the accuracy to 69%, though it still lagged behind the Vanilla approach. Despite preserving structural resemblance with CodeTransCoT, the accuracy dipped slightly. Upon analysis, we found that the errors were primarily due to logical inconsistencies or syntactical differences between the two languages.

7.3.5 Evaluations of the Code Translation Quality

Table 7.6 shows the translation quality metric (refer to chapter 6) for all the settings. Since we follow Chain-of-Thought to generate a structured program similar to the source program, we expect the average graph distance computed to be lower in the CodeTransCoT approach than when computed with the Vanilla approach.

We would need a reference call graph and the generated program's call graph to compute the graph edit distance. Some examples for obtaining such graphs are shown in chapter 6.

Table 7.6: Quality metric results for the Vanilla and CodeTransCoT approach for all the models. The reported number is average graph edit distance across the whole dataset

Dataset	Source	Target	Approach	Models			
				StarCoder	Code Llama	Granite-8B	Granite-20B
Avatar	Java	Python	Vanilla	3.40	3.72	3.86	2.94
			CodeTransCoT	**2.84**	**1.35**	**2.81**	**2.86**
	Python	Java	Vanilla	2.66	2.35	2.34	2.70
			CodeTransCoT	**2.01**	**2.18**	**2.19**	**1.92**
CodeNet	Java	Python	Vanilla	7.17	7.34	7.30	7.01
			CodeTransCoT	**6.89**	**5.80**	**6.30**	**6.17**
	Python	Java	Vanilla	1.74	1.31	1.26	**1.02**
			CodeTransCoT	**1.15**	**0.98**	**1.03**	1.05

Some implementation steps we followed are:

1. We do not consider library calls as nodes in the graph and only consider user-defined functions and classes and the interconnections between them, forming the edges.
2. To simplify it, We removed any implicit/explicit init or constructors.
3. Even though computing graph edit distance is an NP-hard problem, the number of nodes in the source graph does not exceed 20, which makes it inexpensive.
4. We used Java-callgraph [Gou11] and PyCG - Practical Python Call Graphs [SSL+21] to extract Java and Python call graphs, respectively

The datasets we have used contain programs from competitive programming platforms, which means that the program's complexity and structure vary widely on the problem and the user writing the program. So, there is a need to classify and analyze the problems in the dataset based on structural complexity. For this reason, we break the dataset into different categories based on the number of nodes in the program's call graph. Figure 7.3 Shows some line plots where, on the x-axis, we plot the number of nodes, and on the y-axis, we plot the average graph edit distance values.

Figure 7.3 shows the plots for all the models while translating from Java to Python, and for Python to Java, the plots are shown in Figure 7.4. Firstly, there seems to be a positive correlation between the number of nodes and the quality metric value for both methods. In other words, the more complex the source program gets, the harder it is to maintain structure while translating. Secondly, CodeTransCoT does show better performance regarding quality metrics but is only marginally better. This is mainly due to the bias in CoT where the functions or classes in the In-Context demonstrations are added to the target program when it was not necessary, thus reducing the measure of the quality metric.

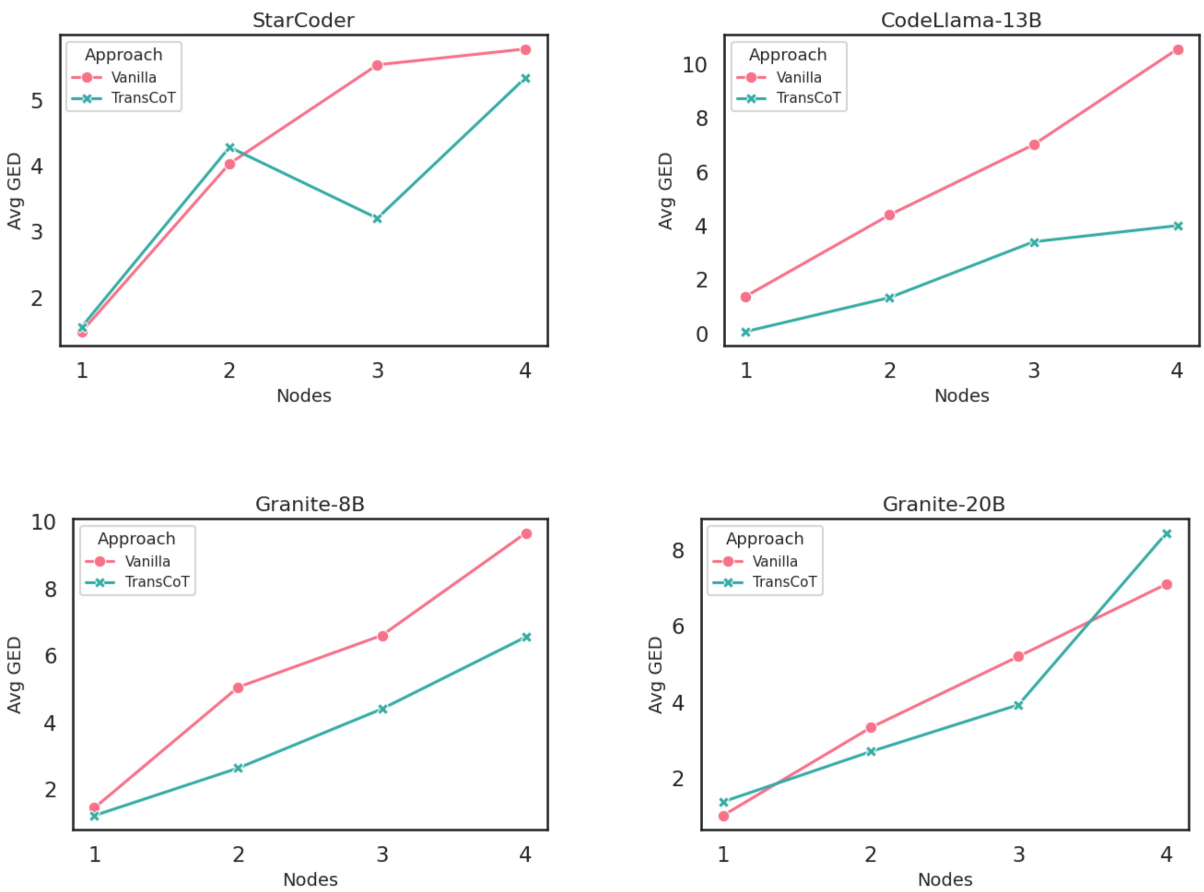

Figure 7.3: Number of nodes vs Average graph edit distance plots. This is on Avatar dataset and translation was from Python to Java

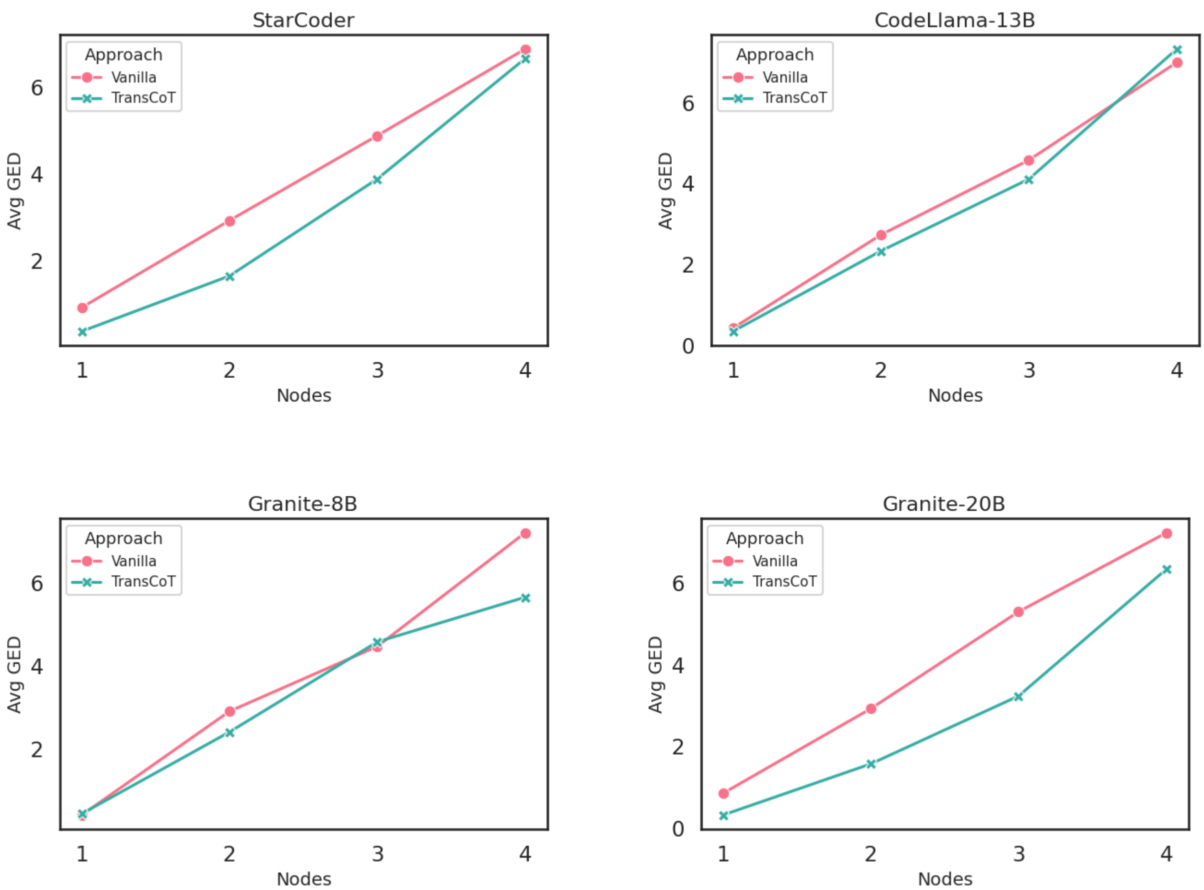

Figure 7.4: Number of nodes vs Average graph edit distance plots. This is on Avatar dataset and translation was from Java to Python

Chapter 8

Discussion

Our approaches have proven effective in achieving structure-consistent code translation. Experimental results, coupled with manual evaluations of our quality metrics, demonstrate the ability to capture and preserve the structural and logical correctness of translated code. Notably, our methods are designed to be efficient and scalable, as we have avoided any form of training involvement.

We have highlighted the importance of maintaining structural and logical information for high-quality code translation. There is, however, room for further improvement and future work. Potential directions include exploring more complex prompting techniques such as Tree of Thoughts (Yao et el., 2023 [YYZ+24]) and utilizing program analysis information to enhance consistency in CoT steps. In the quality metric part, future work may involve incorporating more information about scope and global variables. Additionally, future work should focus on better formulating iterative improvement methods for our approaches, leveraging automatic demonstration sampling relevant to specific problems.

Chapter 9

Conclusion

We have presented an approach for maintaining structural and logical information in code translation tasks while ensuring functional correctness. This approach incorporates reasoning steps related to the skeletal structure of programs within advanced Chain-of-Thought (CoT) prompting. Our experiments, conducted on two datasets and various models, consistently demonstrate superior performance compared to other baselines. Additionally, we have introduced a novel quality evaluation metric that has proven effective in capturing structural information relevant to code translation quality.

Our research underscores the significance of structural and logical information in achieving high-quality and improved code translation. The Discussion chapter outlines the scope for future work and the limitations of our current approach.

Reproducibility Statement. To ensure the reproducibility of our proposed work, we have included prompting templates for each approach. Detailed generation settings are provided in the Experiments Chapter, with additional information on crafting the prompts, including examples, available in the Appendix. The code of the proposed method and related artifacts will be released after the acceptance of the research paper based on this work.

Chapter 10

Appendix

10.1 Example of CompWiseTrans Approach for Java to Python translation (J2P)

Eg.1-Input Prompt J2P_CompWiseTrans

1. Whole Java Code:

```
####################
import java.io.BufferedReader ;
import java.io.IOException ;
import java.io.InputStreamReader ;
import java.util.* ;

public class Main {
public static void main ( String [ ] args ) {
FastScanner input = new FastScanner ( ) ;
int n = input.nextInt ( ) ;
HashMap < Integer, Integer > map = new HashMap < > ( ) ;
for ( int i = 0 ; i < n ; i ++ ) {
int val = input.nextInt ( ) ;
map.put ( val, map.getOrDefault ( val, 0 ) + 1 ) ;
}
int max = Integer.MIN_VALUE ;
for ( Map.Entry < Integer, Integer > entry : map.entrySet ( ) ) {
Integer value = entry.getValue ( ) ;
max = Math.max ( max, value ) ;
}
System.out.println ( max + "" + map.size ( ) ) ;
}
static class FastScanner {
BufferedReader br = new BufferedReader ( new InputStreamReader ( System.in ) ) ;
StringTokenizer st = new StringTokenizer ( "" ) ;

String next ( ) { while (! st.hasMoreTokens ( ) ) {
try { st = new StringTokenizer ( br.readLine ( ) ) ;
} catch ( IOException e ) {
e.printStackTrace ( ) ;
} }
return st.nextToken ( ) ;
```

```
}
int nextInt ( ) {
return Integer.parseInt ( next ( ) ) ;
}
long nextLong ( ) {
return Long.parseLong ( next ( ) ) ;
}
double nextDouble ( ) {
return Double.parseDouble ( next ( ) ) ;
}
String nextLine ( ) throws IOException {
return br.readLine ( ) ;
} } }
```

1. Sample Input:

```
1
1
```

1. Expected Output:

```
1 1
```

1. Steps: Let's think step by step.
Firstly find classes defined in the Java code that are- FastScanner, Main. Let's translate all classes definition and its functions step by step. Here is order in which steps to be followed- First translate all classes instead of "Main" class, then translate "Main" class and then "library imports" step.
Let's translate FastScanner class. Functions present in FastScanner class- next, nextInt, nextLong, nextDouble, nextLine. Translation of FastScanner class and its functions from Java to Python-

Step 1: Translation of "FastScanner" class and its functions -
Python Code:

```python
class FastScanner:
    def __init__(self):
        self.buf = sys.stdin.readline
        self.tokens = None

    def has_next(self):
        while self.tokens is None or len(self.tokens) == 0:
            self.tokens = self.buf().split()
        #some warning need if no input or below line
        if (self.tokens is None or len(self.tokens) == 0):
            return False
        return True

    def next_int(self):
        if not self.has_next():
            return None
        val = self.tokens.pop(0)
```

```
        return int(val)

    def next_str(self):
        if not self.has_next():
            return None
        val = self.tokens.pop(0)
        return val
```

Let's translate Main class. Functions present in Main class- main. Consider above sample input and output format information while translating "main" function to handle input related code properly. Translation of Main class and its functions from Java to Python-

Step 2: Translation of "Main" class and its functions -
Python Code:

```
def main():
    input = FastScanner()
    n = input.next_int()
    map = {}
    for i in range(n):
        val = input.next_int()
        map[val] = map.get(val, 0) + 1
    max = -1
    for entry in map.items():
        value = entry[1]
        max = max if max > value else value
    print(max, len(map))
```

Considering all steps, add required library imports in Python Code-

Step 3: Library imports -
Python Code:

```
import sys
```

Finally Combined all steps- Step 1, Step 2, Step 3 to get syntactically correct whole Python Code-
1. Whole Python Code:

```
import sys

class FastScanner:
    def __init__(self):
        self.buf = sys.stdin.readline
        self.tokens = None

    def has_next(self):
        while self.tokens is None or len(self.tokens) == 0:
            self.tokens = self.buf().split()
        #some warning need if no input or below line
        if (self.tokens is None or len(self.tokens) == 0):
            return False
```

```
        return True

    def next_int(self):
        if not self.has_next():
            return None
        val = self.tokens.pop(0)
        return int(val)

    def next_str(self):
        if not self.has_next():
            return None
        val = self.tokens.pop(0)
        return val

def main():
    input = FastScanner()
    n = input.next_int()
    map = {}
    for i in range(n):
        val = input.next_int()
        map[val] = map.get(val, 0) + 1
    max = -1
    for entry in map.items():
        value = entry[1]
        max = max if max > value else value
    print(max, len(map))

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

2. Whole Java Code:

```
####################
import java.util.Scanner ;

public class Main {
    public static void main ( String [ ] args ) {
        Scanner in_ = new Scanner ( System.in ) ;
        int F = in_.nextInt ( ) ;
        int T = in_.nextInt ( ) ;
        int S = in_.nextInt ( ) ;
        int q = in_.nextInt ( ) ;
        long previous = S ;
        int answer = 0 ;
        while ( previous < T ) {
            answer ++ ; previous *= q ;
        }
        answer*=F;
```

```
        System.out.println ( answer ) ;
} }
```

2. Sample Input:

```
4
5 2 2
```

2. Expected Output:

```
8
```

2. Steps: Let's think step by step.
Firstly find classes defined in the Java code that are- Main. Let's translate all classes definition and its functions step by step. Here is order in which steps to be followed- First translate all classes instead of "Main" class, then translate "Main" class and then "library imports" step.
Let's translate Main class. Functions present in Main class- main. Consider above sample input and output format information while translating "main" function to handle input related code properly.
Translation of Main class and its functions from Java to Python-

Step 1: Translation of "Main" class and its functions -
Python Code:

```
def main():
    F=int(input().strip())
    T, S, q = map ( int, input ( ).split ( ) )
    previous = S
    answer = 0
    while ( previous < T ):
        answer+=1
        previous *= q
    answer*=F
    print( answer )
```

Considering all steps, add required library imports in Python Code-

Step 2: Library imports -
Python Code:
No library import required in Python Code

Finally Combined all steps- Step 1, Step 2 to get syntactically correct whole Python Code.
2. Whole Python Code:

```
def main():
    F=int(input().strip())
    T, S, q = map ( int, input ( ).split ( ) )
    previous = S
    answer = 0
    while ( previous < T ):
        answer+=1
        previous *= q
```

```
    answer*=F
    print( answer )

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

3. Whole Java Code:

```
####################
import java.io.OutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.io.UncheckedIOException;
import java.util.StringTokenizer;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Main {
    public static void main(String[] args) {
        InputStream inputStream = System.in;
        OutputStream outputStream = System.out;
        LightScanner in = new LightScanner(inputStream);
        PrintWriter out = new PrintWriter(outputStream);
        BUnhappyHackingABCEdit solver = new BUnhappyHackingABCEdit();
        solver.solve(1, in, out);
        out.close();
    }

    static class BUnhappyHackingABCEdit {
        public void solve(int testNumber, LightScanner in, PrintWriter out) {
            String s = in.string();
            StringBuilder d = new StringBuilder();
            for (char c : s.toCharArray()) {
                switch (c) {
                    case '0':
                        d.append("0");
                        break;
                    case '1':
                        d.append("1");
                        break;
                    case 'B':
                        if (d.length() > 0) {
                            d.setLength(d.length() - 1);
                            d.trimToSize();
                        }
                        break;
                }
```

```
            }
            out.println(d);
        }
    }

    static class LightScanner {
        private BufferedReader reader = null;
        private StringTokenizer tokenizer = null;

        public LightScanner(InputStream in) {
            reader = new BufferedReader(new InputStreamReader(in));
        }

        public String string() {
            if (tokenizer == null || !tokenizer.hasMoreTokens()) {
                try {
                    tokenizer = new StringTokenizer(reader.readLine());
                } catch (IOException e) {
                    throw new UncheckedIOException(e);
                }
            }
            return tokenizer.nextToken();
        }
    }
}
```

3. Sample Input:
01B0

3. Expected Output:
00

3. Steps: Let's think step by step.

Eg.1-Generated Output J2P_CompWiseTrans

Firstly find classes defined in the Java code that are- LightScanner, BUnhappyHackingABCEdit, Main. Let's translate all classes definition and its functions step by step. Here is order in which steps to be followed- First translate all classes instead of "Main" class, then translate "Main" class and then "library imports" step.
Let's translate LightScanner class. Functions present in LightScanner class- string. Consider above sample input and output format information while translating "string" function to handle input related code properly. Translation of LightScanner class and its functions from Java to Python-

Step 1: Translation of "LightScanner" class and its functions -
Python Code:

```
class LightScanner:
```

```python
    def __init__(self):
        self.reader = sys.stdin.readline
        self.tokenizer = None

    def string(self):
        if self.tokenizer is None or len(self.tokenizer) == 0:
            self.tokenizer = self.reader().split()
        if self.tokenizer is None or len(self.tokenizer) == 0:
            return None
        val = self.tokenizer.pop(0)
        return val
```

Let's translate BUnhappyHackingABCEdit class. Functions present in BUnhappyHackingABCEdit class- solve. Translation of BUnhappyHackingABCEdit class and its functions from Java to Python-

Step 2: Translation of "BUnhappyHackingABCEdit" class and its functions -
Python Code:

```python
class BUnhappyHackingABCEdit:
    def solve(self, testNumber, inScanner, out):
        s = inScanner.string()
        d = []
        for c in s:
            if c == '0':
                d.append('0')
            elif c == '1':
                d.append('1')
            elif c == 'B':
                if len(d) > 0:
                    d.pop()
        out(d)
```

Let's translate Main class. Functions present in Main class- main. Consider above sample input and output format information while translating "main" function to handle input related code properly. Translation of Main class and its functions from Java to Python-

Step 3: Translation of "Main" class and its functions -
Python Code:

```python
def main():
    inScanner = LightScanner()
    out = print
    solver = BUnhappyHackingABCEdit()
    solver.solve(1, inScanner, out)
```

Considering all steps, add required library imports in Python Code-
Step 4: Library imports -
Python Code:

```python
import sys
```

Finally Combined all steps- Step 1, Step 2, Step 3, Step 4 to get syntactically correct whole Python Code.

3. Whole Python Code:

```
import sys

class LightScanner:
    def __init__(self):
        self.reader = sys.stdin.readline
        self.tokenizer = None

    def string(self):
        if self.tokenizer is None or len(self.tokenizer) == 0:
            self.tokenizer = self.reader().split()
        if self.tokenizer is None or len(self.tokenizer) == 0:
            return None
        val = self.tokenizer.pop(0)
        return val

class BUnhappyHackingABCEdit:
    def solve(self, testNumber, inScanner, out):
        s = inScanner.string()
        d = []
        for c in s:
            if c == '0':
                d.append('0')
            elif c == '1':
                d.append('1')
            elif c == 'B':
                if len(d) > 0:
                    d.pop()
        out(''.join(d))

def main():
    inScanner = LightScanner()
    out = print
    solver = BUnhappyHackingABCEdit()
    solver.solve(1, inScanner, out)

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

10.2 Example of CodeTransCoT Approach for Java to Python translation (J2P)

Eg.2-Input Prompt J2P_CodeTransCoT

###$####

1. Java Code:

```
import java.io.BufferedReader ;
import java.io.IOException ;
import java.io.InputStreamReader ;
import java.util.* ;

public class Main {
public static void main ( String [ ] args ) {
FastScanner input = new FastScanner ( ) ;
int n = input.nextInt ( ) ;
HashMap < Integer , Integer > map = new HashMap < > ( ) ;
for ( int i = 0 ; i < n ; i ++ ) {
int val = input.nextInt ( ) ;
map.put ( val , map.getOrDefault ( val , 0 ) + 1 ) ;
}
int max = Integer.MIN_VALUE ;
for ( Map.Entry < Integer , Integer > entry : map.entrySet ( ) ) {
Integer value = entry.getValue ( ) ;
max = Math.max ( max , value ) ;
}
System.out.println ( max + " " + map.size ( ) ) ;
}
static class FastScanner {
BufferedReader br = new BufferedReader ( new InputStreamReader ( System.in ) ) ;
StringTokenizer st = new StringTokenizer ( "" ) ;

String next ( ) { while ( ! st.hasMoreTokens ( ) ) {
try { st = new StringTokenizer ( br.readLine ( ) ) ;
} catch ( IOException e ) {
e.printStackTrace ( ) ;
} }
return st.nextToken ( ) ;
}
int nextInt ( ) {
return Integer.parseInt ( next ( ) ) ;
}
long nextLong ( ) {
return Long.parseLong ( next ( ) ) ;
}
```

```
double nextDouble ( ) {
return Double.parseDouble ( next ( ) ) ;
}
String nextLine ( ) throws IOException {
return br.readLine ( ) ;
} } }
```

1. Sample Input:

```
1
1
```

1. Expected Output:

```
1 1
```

1. Steps: Let's think step by step.
Step 1: First of all identify classes and functions declarations present in the above Java Code-
Identified classes and functions declarations in the Java code:

```
public class Main {
public static void main ( String [ ] args );
static class FastScanner {
String next ( );
int nextInt ( );
long nextLong ( );
double nextDouble ( );
String nextLine ( ) throws IOException;
}
}
```

Step 2: Considering Step 1, classes and functions declarations should be presented in the corresponding Python Code-
Identified classes and functions declarations in the Python code:

```
Class Main:
    def main(self):

    class FastScanner:
        def __init__(self):

        def has_next(self):

        def next_int(self):

        def next_str(self):
```

Step 3: Complete translation of the above Java code to Python code by considering identified classes and functions declarations in Step 2. Consider above sample input and output format information to handle input and output related code properly in the generated Python code. While generating the Python code, keep track of required library imports to be added in the Python Code. Make sure your generated code is syntactically correct-
1. Python Code:

```
import sys

class Main:
    def main(self):
        input = self.FastScanner()
        n = input.next_int()
        map = {}
        for i in range(n):
            val = input.next_int()
            map[val] = map.get(val, 0) + 1
        max = -1
        for entry in map.items():
            value = entry[1]
            max = max if max > value else value
        print(max, len(map))

    class FastScanner:
        def __init__(self):
            self.buf = sys.stdin.readline
            self.tokens = None

        def has_next(self):
            while self.tokens is None or len(self.tokens) == 0:
                self.tokens = self.buf().split()
            #some warning need if no input or below line
            if (self.tokens is None or len(self.tokens) == 0):
                return False
            return True

        def next_int(self):
            if not self.has_next():
                return None
            val = self.tokens.pop(0)
            return int(val)

        def next_str(self):
            if not self.has_next():
                return None
            val = self.tokens.pop(0)
            return val

if __name__ == "__main__":
    obj = Main()
    obj.main()

###$####
```

2. Java Code:

```
import java.util.Scanner ;

public class Main {
    public static void main ( String [ ] args ) {
        Scanner in_ = new Scanner ( System.in ) ;
        int F = in_.nextInt ( ) ;
        int T = in_.nextInt ( ) ;
        int S = in_.nextInt ( ) ;
        int q = in_.nextInt ( ) ;
        long previous = S ;
        int answer = 0 ;
        while ( previous < T ) {
            answer ++ ; previous *= q ;
        }
        answer*=F;
        System.out.println ( answer ) ;
} }
```

2. Sample Input:

```
4
5 2 2
```

2. Expected Output:

```
8
```

2. Steps: Let's think step by step.
Step 1: First of all identify classes and functions declarations present in the above Java Code-
Identified classes and functions declarations in the Java code:

```
public class Main {
    public static void main ( String [ ] args );
}
```

Step 2: Considering Step 1, classes and functions declarations should be presented in the corresponding Python Code-
Identified classes and functions declarations in the Python code:

```
Class Main:
    def main(self):
```

Step 3: Complete translation of the above Java code to Python code by considering identified classes and functions declarations in Step 2. Consider above sample input and output format information to handle input and output related code properly in the generated Python code. While generating the Python code, keep track of required library imports to be added in the Python Code. Make sure your generated code is syntactically correct-
2. Python Code:

```
class Main:
    def main(self):
        F=int(input().strip())
```

```
        T , S , q = map ( int , input ( ).split ( ) )
        previous = S
        answer = 0
        while ( previous < T ):
            answer+=1
            previous *= q
        answer*=F
        print( answer )

if __name__ == "__main__":
    obj = Main()
    obj.main()

###$####
```

3. Java Code:

```
import java.io.OutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;
import java.io.UncheckedIOException;
import java.util.StringTokenizer;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Main {
    public static void main(String[] args) {
        InputStream inputStream = System.in;
        OutputStream outputStream = System.out;
        LightScanner in = new LightScanner(inputStream);
        PrintWriter out = new PrintWriter(outputStream);
        BUnhappyHackingABCEdit solver = new BUnhappyHackingABCEdit();
        solver.solve(1, in, out);
        out.close();
    }

    static class BUnhappyHackingABCEdit {
        public void solve(int testNumber, LightScanner in, PrintWriter out) {
            String s = in.string();
            StringBuilder d = new StringBuilder();
            for (char c : s.toCharArray()) {
                switch (c) {
                    case '0':
                        d.append("0");
                        break;
                    case '1':
                        d.append("1");
```

```
                    break;
                case 'B':
                    if (d.length() > 0) {
                        d.setLength(d.length() - 1);
                        d.trimToSize();
                    }
                    break;
            }
        }
        out.println(d);
    }
}

static class LightScanner {
    private BufferedReader reader = null;
    private StringTokenizer tokenizer = null;

    public LightScanner(InputStream in) {
        reader = new BufferedReader(new InputStreamReader(in));
    }

    public String string() {
        if (tokenizer == null || !tokenizer.hasMoreTokens()) {
            try {
                tokenizer = new StringTokenizer(reader.readLine());
            } catch (IOException e) {
                throw new UncheckedIOException(e);
            }
        }
        return tokenizer.nextToken();
    }
}
}
```

3. Sample Input:

```
01B0
```

3. Expected Output:

```
00
```

3. Steps: Let's think step by step.

Eg.2-Generated Output J2P_CodeTransCoT

Step 1: First of all identify classes and functions declarations present in the above Java Code-
Identified classes and functions declarations in the Java code:

```
public class Main {
    public static void main ( String [ ] args );
    static class BUnhappyHackingABCEdit {
        public void solve ( int testNumber, LightScanner in, PrintWriter out ) ;
    }
    static class LightScanner {
        public LightScanner ( InputStream in ) ;
        public String string ( ) ;
    }
}
```

Step 2: Considering Step 1, classes and functions declarations should be presented in the corresponding Python Code-

Identified classes and functions declarations in the Python code:

```
Class Main:
    def main(self):

    class BUnhappyHackingABCEdit:
        def solve(self, testNumber, in, out):

    class LightScanner:
        def __init__(self):

        def string(self):
```

Step 3: Complete translation of the above Java code to Python code by considering identified classes and functions declarations in Step 2. Consider above sample input and output format information to handle input and output related code properly in the generated Python code. While generating the Python code, keep track of required library imports to be added in the Python Code. Make sure your generated code is syntactically correct-

3. Python Code:

```
import sys

class Main:
    def main(self):
        in_ = self.LightScanner(sys.stdin)
        out = sys.stdout
        solver = self.BUnhappyHackingABCEdit()
        solver.solve(1, in_, out)

    class BUnhappyHackingABCEdit:
        def solve(self, testNumber, in, out):
            s = in.string()
            d = ""
            for c in s:
                if c == '0':
                    d += "0"
                elif c == '1':
```

```
                d += "1"
            elif c == 'B':
                if len(d) > 0:
                    d = d[:-1]
        out.println(d)

    class LightScanner:
        def __init__(self, in_):
            self.reader = sys.stdin
            self.tokens = None

        def string(self):
            if self.tokens is None or len(self.token)==0:
                self.tokens = self.reader.readline().split()
            return self.tokens.pop(0)

if __name__ == "__main__":
    obj = Main()
    obj.main()
```

10.3 Example of CodeTransCoT Approach for Python to Java translation (P2J)

Eg.3-Input Prompt P2J_CodeTransCoT

###$####

1. Python Code:

```
l = [ ]
limit = 10000000000
def gen ( number , four , seven ) :
    global debug
    debug = True
    status = "bad"
    if ( number > limit ) :
        return
    if ( number > 0 and four == seven ) :
        l.append ( number )
    gen ( number * 10 + 4 , four + 1 , seven )
    gen ( number * 10 + 7 , four , seven + 1 )

debug= False
status = "good"
```

```python
def main ( ) :
    gen ( 0 , 0 , 0 )
    l.sort ( )
    n = int ( input ( ) )
    ans = 0
    for val in l :
        if ( val >= n ) :
            ans = val
            break
    print ( ans )
```

1. Sample Input:

```
9
```

1. Expected Output:

```
47
```

1. Steps: Let's think step by step.
Step 1: First of all identify global variables and their types. Global variables are variables that are not inside scope of function and class declarations, present in the above Python Code-
Global variables:

```
l, limit, debug, status
```

Step 2: Identify classes and functions declarations present in the above Python Code-
Identified classes and functions declarations in the Python code:

```python
def gen ( number , four , seven ) :

def main ( ) :
```

Step 3: Considering Step 1 and Step 2, the corresponding Java Code should have declarations of these classes, functions and Global variables -
Identified classes, functions and Global variables declarations in the Java code:

```java
public class Main {
    static ArrayList<Long> l;
    static long limit;
    static boolean debug;
    static String status;

    public static void gen(long number, int four, int seven);

    public static void main ( String [ ] args );

}
```

Step 4: Complete translation of the above Python code to Java code by considering identified classes, functions and Global variables declarations in Step 3. Consider above sample input and output format information to handle input and output related code properly in the generated Java code. While generating the Java code, keep track of required library imports to be added in the Java Code. Make sure your generated code is syntactically correct-

1. Java Code:

```
public class Main{
    static ArrayList<Long> l;
    static long limit;
    static boolean debug;
    static String status;

    public static void gen(long number, int four, int seven) {
        Main.debug = True;
        String status = "bad";
        if (number > limit) return;
        if (number > 0 && four == seven) {
            l.add(number);
        }
        gen(number * 10 + 4, four + 1, seven);
        gen(number * 10 + 7, four, seven + 1);
    }

    public static void main(String[] args) {
        l = new ArrayList<>();
        limit = 10000000000L;
        debug= False;
        status = "good";

        gen(0, 0, 0);
        Collections.sort(l);
        Scanner sc = new Scanner(System.in);
        long n = sc.nextLong();
        long ans = 0;
        for (long val : l) {
            if (val >= n) {
                ans = val;
                break;
            }
        }
        System.out.println(ans);
    }
}


###$####
```

2. Python Code:

```
class BIT :
```

```
    def __init__ ( self , N ) :
        self.size = N
        self.tree = [ 0 ] * ( N + 1 )
        self.depth = n.bit_length ( )
    def _bitsum ( self , i ) :
        ret = 0
        while i :
            ret += self.tree [ i ]
            i ^= i & - i
        return ret
    def bitsum ( self , l , r = None ) :
        if r is None :
            return self._bitsum ( l )
        else :
            return self._bitsum ( r ) - self._bitsum ( l )
    def bitadd ( self , i , x ) :
        i += 1
        while i <= self.size :
            self.tree [ i ] += x
            i += i & - i
        return
n = int ( input ( ) )
m = n * ( n + 1 ) // 4
a = list ( map ( int , input ( ).split ( ) ) )
d = dict ( )
_a = sorted ( set ( a + [ 0 ] ) )
for i , x in enumerate ( _a ) :
    d [ x ] = i
a = [ d [ x ] for x in a ]
def check ( X ) :
    b = [ 0 ] + [ ( y >= X ) * 2 - 1 for y in a ]
    for i in range ( n ) :
        b [ i + 1 ] += b [ i ]
    c = min ( b )
    b = [ x - c for x in b ]
    bit = BIT ( max ( b ) + 2 )
    ans = 0
    for x in b :
        ans += bit.bitsum ( x + 1 )
        bit.bitadd ( x , 1 )
    return ans >= m
t = [ len ( _a ) , 0 ]
while t [ 0 ] - t [ 1 ] > 1 :
    mid = ( t [ 0 ] + t [ 1 ] ) // 2
    t [ check ( mid ) ] = mid
print ( _a [ t [ 1 ] ] )
```

2. Sample Input:

```
1
1
```

2. Expected Output:

```
1
```

2. Steps: Let's think step by step.
Step 1: First of all identify global variables and their types. Global variables are variables that are not inside scope of function and class declarations, present in the above Python Code-
Global variables:

```
n, m, a, d, _a, t
```

Step 2: First of all identify global variables and their types. Global variables are variables that are not inside scope of function and class declarations, present in the above Python Code-
Identified classes and functions declarations in the Python code:

```python
class BIT :
    def __init__ ( self , N ) :
        self.size = N
        self.tree = [ 0 ] * ( N + 1 )
        self.depth = N.bit_length ( )

    def _bitsum ( self , i ) :

    def bitsum ( self , l , r = None ) :

    def bitadd ( self , i , x ) :

def check ( X ) :
```

Step 3: Considering Step 1 and Step 2, the corresponding Java Code should have declarations of these classes, functions and Global variables -
Identified classes, functions and Global variables declarations in the Java code:

```java
public class Main {
    static int n;
    static long m;
    static int[] a;
    static Map<Integer, Integer> d;
    static List<Integer> _a;
    static int[] t;

    public static class BIT {
        public int size;
        public long [] tree;
        public int depth;

        public BIT(int N);
```

```
        public long _bitsum(int i);

        public long bitsum(int l, int r);

        public long bitsum(int l);

        public void bitadd(int i, int x);

        }

    public static boolean check(int X);

    public static void main ( String [ ] args );

}
```

Step 4: Complete translation of the above Python code to Java code by considering identified classes, functions and Global variables declarations in Step 3. Consider above sample input and output format information to handle input and output related code properly in the generated Java code. While generating the Java code, keep track of required library imports to be added in the Java Code. Make sure your generated code is syntactically correct-

2. Java Code:

```java
import java.util.*;
import java.io.*;
import java.util.stream.Collectors;

public class Main{
    static int n;
    static long m;
    static int[] a;
    static Map<Integer, Integer> d;
    static List<Integer> _a;
    static int[] t;

    public static class BIT {
        public int size;
        public long [] tree;
        public int depth;

        public BIT(int N) {
            this.size = N;
            this.tree = new long[N + 1];
            this.depth = Integer.toBinaryString(N).length();
        }

        public long _bitsum(int i) {
```

```
        long ret = 0;
        while (i > 0) {
            ret += this.tree[i];
            i ^= i & -i;
        }
        return ret;
    }

    public long bitsum(int l, int r) {
        return this._bitsum(r) - this._bitsum(l);
    }

    public long bitsum(int l) {
        return this._bitsum(l);
    }

    public void bitadd(int i, int x) {
        i += 1;
        while (i <= this.size) {
            this.tree[i] += x;
            i += i & -i;
        }
        return;
    }
}


public static boolean check(int X) {
    int[] b = new int[n + 1];
    b[0] = 0;
    for (int i = 0; i < n; i++) {
        b[i + 1] = (a[i] >= X ? 2 : 0) - 1;
    }
    for (int i = 0; i < n; i++) {
        b[i + 1]+= b[i];
    }
    int c = Arrays.stream(b).min().getAsInt();
    for (int i = 0; i < b.length; i++) {
        b[i] -= c;
    }
    BIT bit = new BIT(Arrays.stream(b).max().getAsInt() + 2);
    long ans = 0;
    for (int x : b) {
        ans += (long) bit.bitsum(x + 1);
        bit.bitadd(x, 1);
    }
    return ans >= m;
```

```
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();
        m = (long) n * (n + 1) / 4;
        a = new int[n];
        for (int i = 0; i < n; i++) {
            a[i] = scanner.nextInt();
        }
        d = new HashMap<>();
        Set<Integer> set = new HashSet<>();
        for (int x : a) {
            set.add(x);
        }
        set.add(0);
        _a = new ArrayList<>(set);
        Collections.sort(_a);
        for (int i = 0; i < _a.size(); i++) {
            d.put(_a.get(i), i);
        }
        for (int i = 0; i < a.length; i++) {
            a[i] = d.get(a[i]);
        }

        t = new int[]{ _a.size(), 0 };

        while (t[0] - t[1] > 1) {
            int mid = (t[0] + t[1]) / 2;
            if (check(mid)) {
                t[1] = mid;
            } else {
                t[0] = mid;
            }
        }
        System.out.println(_a.get(t[1]));
    }
}
```

###$####

3. Python Code:

```
import queue
n , b = list ( map ( int , input ( ).split ( ) ) )
class Task :
    def __init__ ( self , time : int , duration : int , index : int ) -> None :
        super ( ).__init__ ( )
```

```
        self.time = time
        self.duration = duration
        self.index = index
remaining = queue.Queue ( )
running = False
finish_time = 0
def run_task ( remaining : queue.Queue , finish_time : int ) :
    task_to_run = remaining.get ( )
    finish_time = max ( finish_time , task_to_run.time ) + task_to_run.duration
    result [ task_to_run.index ] = finish_time
    return finish_time , result
result = { }
for i in range ( n ) :
    time , duration = list ( map ( int , input ( ).split ( ) ) )
    task = Task ( time , duration , index = i )
    result.update ( { i : 0 } )
    if task.time > finish_time and remaining.empty ( ) :
        running = True
        finish_time = task.time + task.duration
        result [ i ] = finish_time
    else :
        if task.time >= finish_time and not remaining.empty ( ) :
            finish_time , result = run_task ( remaining = remaining , finish_time =
                finish_time )
        if remaining.qsize ( ) < b :
            remaining.put ( task )
        else :
            result [ i ] = - 1
while not remaining.empty ( ) :
    finish_time , result = run_task ( remaining = remaining , finish_time =
        finish_time )
for key in result :
    print ( result.get ( key ) )
```

3. Sample Input:

```
4 1
1 2
2 1
3 1
4 3
```

3. Expected Output:

```
3 4 5 8
```

3. Steps: Let's think step by step.

Eg.3-Generated Output P2J _ CodeTransCoT

Step 1: First of all identify global variables and their types. Global variables are variables that are not inside scope of function and class declarations, present in the above Python Code-

Global variables:

```
n, b, remaining, running, finish_time, result
```

Step 2: First of all identify global variables and their types. Global variables are variables that are not inside scope of function and class declarations, present in the above Python Code-

Identified classes and functions declarations in the Python code:

```
class Task :
    def __init__ ( self, time : int, duration : int, index : int ) -> None :
        super ( ).__init__ ( )
        self.time = time
        self.duration = duration
        self.index = index

def run_task ( remaining : queue.Queue, finish_time : int ) :
```

Step 3: Considering Step 1 and Step 2, the corresponding Java Code should have declarations of these classes, functions and Global variables -

Identified classes, functions and Global variables declarations in the Java code:

```
public class Main {
    static int n;
    static int b;
    static Queue<Task> remaining;
    static boolean running;
    static int finish_time;
    static Map<Integer, Integer> result;

    public static class Task {
        public int time;
        public int duration;
        public int index;

        public Task(int time, int duration, int index);
    }

    public static int run_task(Queue<Task> remaining, int finish_time);

    public static void main(String[] args);
}
```

Step 4: Complete translation of the above Python code to Java code by considering identified classes, functions and Global variables declarations in Step 3. Consider above sample input and output format information to handle input and output related code properly in the generated Java code. While generating the Java code, keep track of required library imports to be added in the Java

Code. Make sure your generated code is syntactically correct-

3. Java Code:

```
import java.util.*;
import java.util.stream.Collectors;

public class Main{
    static int n;
    static int b;
    static Queue<Task> remaining;
    static boolean running;
    static int finish_time;
    static Map<Integer, Integer> result;

    public static class Task {
        public int time;
        public int duration;
        public int index;

        public Task(int time, int duration, int index) {
            this.time = time;
            this.duration = duration;
            this.index = index;
        }
    }

    public static int run_task(Queue<Task> remaining, int finish_time) {
        Task task_to_run = remaining.poll();
        finish_time = Math.max(finish_time, task_to_run.time) +
            task_to_run.duration;
        result.put(task_to_run.index, finish_time);
        return finish_time;
    }

     public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();
        b = scanner.nextInt();
        remaining = new LinkedList<>();
        running = false;
        finish_time = 0;
        result = new HashMap<>();
        for (int i = 0; i < n; i++) {
            int time = scanner.nextInt();
            int duration = scanner.nextInt();
            Task task = new Task(time, duration, i);
            result.put(i, 0);
            if (task.time > finish_time && remaining.isEmpty()) {
```

```
                running = true;
                finish_time = task.time + task.duration;
                result.put(i, finish_time);
            } else {
                if (task.time >= finish_time &&!remaining.isEmpty()) {
                    finish_time = run_task(remaining, finish_time);
                }
                if (remaining.size() < b) {
                    remaining.add(task);
                } else {
                    result.put(i, -1);
                }
            }
        }
        while (!remaining.isEmpty()) {
            finish_time = run_task(remaining, finish_time);
        }
        for (int key : result.keySet()) {
            System.out.println(result.get(key));
        }
    }
}
```

Bibliography

[ATCC21] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*, 2021.

[BMR+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[CTJ+21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[Fom19] Nikita Fomin. py2java: Python to java language translator, 2019. `https://pypi.org/project/py2java/`, 2019.

[Gou11] Georgios Gousios. Java-callgraph: Java call graph utilities. `https://github.com/gousiosg/java-callgraph`, 2011.

[HBQC24] Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. Codecot: Tackling code syntax errors in cot reasoning for code generation, 2024.

[JJJ+24] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution, 2024.

[KGR+22] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke

Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[KLBA+22] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022.

[LAZ+23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[LLLJ23] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*, 2023.

[LRCL20] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages, 2020.

[MLH+22] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022.

[MSZ+24] Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, Yan Koyfman, Boris Lublinsky, Maximilien de Bayser, Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Yi Zhou, Chris Johnson, Aanchal Goyal, Hima Patel, Yousaf Shah, Petros Zerfos, Heiko Ludwig, Asim Munawar, Maxwell Crouse, Pavan Kapanipathi, Shweta Salaria, Bob Calio, Sophia Wen, Seetharami Seelam, Brian Belgodere, Carlos Fonseca, Amith Singhee, Nirmit

Desai, David D. Cox, Ruchir Puri, and Rameswar Panda. Granite code models: A family of open foundation models for code intelligence, 2024.

[OAA+23] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael

Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2023.

[Ope23] OpenAI. Gpt-3.5 turbo fine-tuning and api updates. `https://openai.com/index/gpt-3-5-turbo-fine-tuning-and-api-updates/`, 2023. Accessed: 2024-06-28.

[PIK+24] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24. ACM, April 2024.

[PKJ+21] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[PSR23] Sindhu Tipirneni Parshin Shojaee, Aneesh Jain and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. arxiv preprint arxiv:2301.13816, 2023, 2023.

[RGG+24] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy

Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[RLCL20] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611, 2020.

[SRL+23] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations, 2023.

[SSL+21] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python, 2021.

[TMS16] Ling Li Iulius Curt Troy Melhase, Brian Kearns and Shyam Saladi. java2python: Simple but effective tool to translate java source code into python. `https://github.com/natural/java2python`, 2016.

[tss23] Tss. the most accurate and reliable source code converters, 2023.(tangible software solutions). `https://www.tangiblesoftwaresolutions.com`, 2023.

[WBZ+21] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

[WWS+22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[XNFR23] Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn Rose. Data augmentation for code translation with comparable corpora and multiple references, 2023.

[YYZ+24] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[YZC+23] Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. Chain-of-thought in neural code generation: From and for lightweight language models. *arXiv preprint arXiv:2312.05562*, 2023.

[ZZLS22] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*, 2022.