From Ambiguity to Clarity: Free Form Input to Code via Sentence Transformation

Nikita Kiran Yeole
Computer Science
Virginia Tech, Blacksburg, USA
nikitay@vt.edu

Michael S. Hsiao
Electrical and Computer Engineering
Virginia Tech, Blacksburg, USA
hsiao@vt.edu

Abstract—**In the realm of natural language programming, translating free-form sentences in natural language into a functional, machine-executable program remains difficult due to the following 4 challenges. First, the inherent ambiguity of natural languages. Second, the high-level verbose nature in user descriptions. Third, the complexity in the sentences and fourth, the invalid or semantically unclear sentences. Our proposed solution is a large language model (LLM)-based artificial intelligence driven assistant to process free-form sentences and decompose them into sequences of simplified, unambiguous sentences that abide by a set of rules, thereby stripping away the complexities embedded within the original sentences. The resulting sentences are then used to generate the code. For the sentences which still contain ambiguity and complexity, they are passed through another 2 step process. This includes transforming the free-form sentences written by users into JavaScript code and then reframing the original sentence using the generated JavaScript code. Although the JavaScript code generated by LLM might not be correct, this step is simply to use the code to help break down sentences into more precise sequence of actions. This effectively addresses various linguistic challenges that arise in natural language programming. We applied the proposed approach to a set of free-form sentences written by middle-school students for describing the logic behind video games. More than 76% of the free-form sentences containing these problems were successfully converted to sequences of simple unambiguous object-oriented sentences by our approach.**

Keywords-Natural language programming; decomposition; chain-of-thought reasoning.

I. Introduction

Natural Language Programming (NLPg) is a concept that attempts to convert instructions/specifications written in free-form natural language (NL) into functional program code. NLPg envisions a world in which everyone can program machines without understanding the intricacies of conventional programming languages. While generative Artificial Intelligence (AI) has shown some success in producing code snippets from natural language text, the code that is produced may not adhere to the intent of the input text. When the code does not meet the intent, the user can do one of two things: (1) manually modify the generated code, or (2) rewrite the natural language text and try to generate new code [1]. For users who are not experienced programmers, option 1 may not be feasible, since the generated code may contain data structures and/or algorithms that the user is unfamiliar with. Hence, the user is left with the second option. In order to generate functionally correct code, the input must be in a format that the system can process so that common problems with general natural languages are removed. In other words, if the input text is semantically unambiguous, the generated code will more likely adhere to the intent of the input text [2].

An additional benefit is that this helps the user to learn to write unambiguous input text, a necessary skill behind the thought processes in coding. In recent years, NL is increasingly applied in education for personalized AI tutoring and interactive learning, aiding educators in various ways [3] [4] [5]. The ability to instruct a machine in NL bridges the gap between human thought processes and the digital world, making technology more accessible and intuitive for students.

There are many factors associated with NL instructions, which makes NLPg extremely challenging [6]. First, NL sentences often contain ambiguity. Second, descriptions provided by humans tend to be verbose and high-level. Third, the structure of sentences can be complex and compound. Fourth, humans may write invalid or erroneous sentences. We will briefly highlight each of these four challenges in the following discussion.

NL sentences can include ambiguities wherein a single word or phrase may have several interpretations. Consider, for instance, the following English sentence employed in game design:

”When the rabbit touches a rock, it explodes.”

Here, the phrase containing the pronoun ’it’ creates uncertainty in this sentence. According to one view, the rabbit explodes after touching the rock, whereas the other contends that the rock explodes.

Secondly, the NL instructions can be excessively verbose, especially written by the people who may not know how to program. Consider, for instance, the English sentence employed in game design:

”In a mysterious realm, a lone pointer and some aliens engage in a cosmic dance. When the pointer touches an alien, it changes colors: original to purple, purple to pink. Pink aliens explode.”

Here, the sentences provided are verbose with extraneous descriptive words and phrases. Although they adhere to proper English grammar, they deviate from a concise format. For instance, phrases such as ’mysterious realm’ and ’cosmic dance’ may be problematic to implement in code.

Thirdly, machines typically demand sentences with a clear structure containing a subject, verb, and object. However, complex sentences that sequentially combine multiple events may complicate the parsing of the sentence and prevent a full understanding of the intent of the user. The following sentence illustrates one such example:

"When the carrot turns into a diamond before the carrot touches a fox, the score increases."

In this above example, sequencing of events is necessary in order to determine when to increase the score.

Fourthly, when humans provide instructions, there is a chance that they might offer sentences that are invalid, illogical, incomplete or erroneous. In such cases, it becomes difficult for the machine to extract the exact task that needs to be executed. The following is one self-explanatory example containing incomplete and/or erroneous sentences:

"Brick spawns at the bottom. 14 cheese at the top in rows. Ball in the middle. w is up. s is down. brick touches border bounce. ball touches cheese bounces back."

To overcome these challenges, we propose an Artificial Intelligence driven assistant using Large Language Models (LLMs), which will attempt to convert the free-form sentences into sequences of simple sentences, each with a clear subject, verb, and object structure. It promotes a paradigm where instead of the user conforming to the machine, the machine adapts to grasp the user's intent. This assistant streamlines, simplifies, and transforms the NL phrases into directives that machines can easily interpret. The design of the assistant prioritizes rule-driven simplification, methodically translating sentences that eliminate unnecessary elements while retaining the core meaning.

Motivating Example: Consider the following free-form description of a game:

"The rabbit wanders, reversing at borders. The fox wanders, chasing the rabbit when spotting the rabbit. When the rabbit touches the fox, the fox turns into a carrot."

Our goal is to convert the above paragraph to the following simplified, precise sentences.

"There is a rabbit. There is a fox. The rabbit wanders. The fox wanders. If the rabbit reaches a border, it reverses. If the fox sees the rabbit, it chases the rabbit. When the rabbit touches the fox, the fox becomes mutated. When the fox is mutated, it turns into a carrot."

The deconstruction of complex sentences and then rewriting them in basic, simple sentences is the most novel aspect of our strategy. The NL expression frequently combines various thoughts or directives in a single, complex sentence [7]. So, these sentences are decomposed and rewritten in a format that abides by imposed rules. In our approach, the input sentences are parsed, during which the engine identifies key components and breaks them down into their basic elements. By analyzing the relationships between these elements, the system deciphers the user's intention. With this insight, it reconstructs the information into simple sentences that are structured and guided by rules.

The novelty of this paper lies in its specific methodology for simplifying natural language sentences into structured directives through a rule-based system, a departure from traditional semantic parsing and tree-based neural network models, which often struggle with the ambiguity and complexity of natural language [6]. We also integrate an educational platform, GameChangineer, to demonstrate the practical application of this approach, showcasing how it facilitates the learning of object-oriented programming concepts by converting these simplified sentences into functional game code.

We applied our approach to process 1000 free-write sentences, out of which 800 sentences contained at least one of the four aforementioned problems, and 200 sentences are non-problematic sentences. The rewritten sentences are then given to an educational platform called GameChangineer [8] [9] that can convert the object oriented English sentences to a functional game [10]. An object-oriented English sentence structures natural language to reflect object-oriented programming concepts. It clearly defines objects (nouns), their attributes, and their methods (actions). GameChangineer is an AI-Enabled Design and Education Platform, which helps students to discover and practice logical reasoning, problem-solving, algorithmic design, critical and computational thinking [8]. Beginners may find Object-Oriented Programming (OOP) to be abstract and challenging to understand due to its emphasis on classes, objects, inheritance, polymorphism, encapsulation, and abstraction. Students can express their thoughts and queries in a way that comes naturally to them when they are able to interact with an educational software through natural language. This reduces the cognitive load associated with learning new, technical syntax and concepts, allowing them to focus more on the underlying principles of OOP. The results showed that more than 60% of the problematic sentences were successfully converted by our approach. The sentences, which were successfully converted led to a correct functional game, which adheres to the intent of the user. Nevertheless, this method had limitations, particularly when dealing with sentence constructs that are ambiguous and complex. If user input sentences still contain ambiguity or complexity after this process, they are passed through a two-step process. We address these limitations by concentrating on directly generating JavaScript code from user-provided free-form sentences. We note that the JavaScript code generated by the LLM from the original sentences might not be correct. Nevertheless, this step is simply to use the generated code to help break down the sentences into more precise, unambiguous sequence of instructions. There are two main steps to this process:

1) Code Generation: To translate the user's natural language instructions into JavaScript code, we make use of a LLM, which is GPT 4. The generated code may not be correct; but this is fine since we simply want to use the code structure to inference the sentence transformation in the next step [11].
2) Sentence Transformation: The original sentence is broken down and reframed into a set of clear and concise sentences using the JavaScript code that was generated.

The rest of the paper is organized as follows. Section II describes the related work. Section III lays out the methodology I in our work, Section IV lays out the methodology II and Section V presents the evaluation of our Methodology I and discusses its implications. Section VI presents the evaluation of our Methodology II. Finally, Section VII concludes the paper.

II. Related Work

A curated list of groundbreaking studies that has had an impact in this field is included in this section.

One approach to addressing these NL challenges is through semantic parsing, where natural language utterances are encoded and translated into syntactically correct target code snippets using tree-based neural network models [6]. This technique shows promise in generating accurate code snippets from natural language descriptions by focusing on the structural aspects of language to reduce ambiguity and manage complexity. Even sophisticated semantic parsing models, while capable of generating syntactically correct code from natural language inputs, often face difficulties in capturing the user's intent accurately. This is because a single phrase can be interpreted in multiple ways, leading to code that, while technically correct, does not fulfill the intended function [6].

Another sophisticated method involves using execution-based selection processes and Minimum Bayes Risk (MBR) decoding to minimize expected errors in the generated code [12]. This approach selects the most accurate output by considering the execution results of the generated code samples, helping to ensure that the generated code aligns with the intended functionality described in natural language. This approach has its limitations. It requires executing several generated code snippets to determine the best candidate, which can be computationally expensive and inefficient. Furthermore, if the initial pool of generated code contains errors or fails to capture the user's intent accurately, the selection process may still result in sub-optimal code [12].

Deep learning techniques offer significant advancements in understanding and generating code from natural language. By leveraging the encoder-decoder framework, these models can learn from vast datasets of code to improve the accuracy and relevance of generated code snippets, addressing issues of verbosity and complex sentence structures by focusing on the semantic content of the instructions [13]. Although deep learning has shown promise in understanding and generating code, the models still struggle with sentences that contain multiple actions or intertwined concepts, reflecting a gap in handling real-world complexity [13]. These limitations underline the necessity for a proposed solution that addresses these core issues.

The Transformer model was first presented by Vaswani et al. in their landmark study, "Attention Is All You Need" [14]. In order to deal with ambiguity, the architecture's self-attention mechanism, which is skilled at capturing context, is essential.

Generative pre-trained transformer (GPT)-3 showed its skill in deciphering a wide range of human expressions and offered a solution to unclear or lacking instructions [15]. Despite its outstanding powers, GPT-3 occasionally produces overly detailed or irrelevant answers [15]. GPT-3 also frequently requires particular fine-tuning for certain tasks [15]. BERT's (Bidirectional Encoder Representations from Transformers) pre-training procedure was improved by Liu et al., who published "RoBERTa: A Robustly Optimized BERT Pretraining Approach" [16] [17].

Wei et al.'s study on "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models" forms a crucial basis for understanding how Chain of Thought (CoT) in LLMs can decompose complex reasoning tasks into a series of simpler, logical steps [18]. The authors demonstrate that CoT prompting significantly improves the ability of LLMs to perform complex reasoning tasks across various domains. We employ CoT not for general reasoning enhancement, but specifically for tackling linguistic challenges in programming, such as verbosity, ambiguities, and complex phrase structures.

We focus on preserving the fundamental semantic meaning of the given instructions while simultaneously addressing the inherent difficulties and limitations of human language. The subtleties of freely written phrases can have a profound impact on the semantic meaning, which is the fundamental core of a communication [19]. Therefore, a major goal in this area should be to transform these statements into more straightforward forms without distorting or losing the original meaning that the user intended. This balance makes sure that, despite the language being more structured or standardized for computational processing, the converted sentences remain true to the message the user intended to convey.

III. Methodology I

The foundation of our research is a representative dataset, which was used as the LLM's main input. The data included 1000 student-written free-form sentences as game descriptions. 800 of these sentences have been identified as potentially problematic and 200 sentences have been identified as non-problematic. These descriptions offered a variety of linguistic patterns and semantic complexities. The game descriptions were diverse, varied in their lengths, and offered a number of difficulties. These sentences showed some ambiguity because they frequently contained intricate structures and relationships that were not always clear. This dataset was also chosen to evaluate the LLM's capacity to comprehend and translate the ambiguous and complex texts into more rule-based, simplified formats.

We used the GPT-3.5 Turbo, a powerful language model created by OpenAI, for the purposes of this study. We made this choice after carefully comparing the performance of GPT-3.5 Turbo and GPT-4, two recent revisions of OpenAI's generative models. Although GPT-4 is a more recent model and is anticipated to offer higher capabilities in many contexts [20], GPT-3.5 Turbo showed improved sentence construction in the most basic form and coherence for the particular prompt utilized in this research. This underscored the need of selecting a model that is tailored to the precise specifications of the work at hand as opposed to just selecting the most recent version. This model was deployed by means of direct integration with the OpenAI API, which allowed us to operate the model locally in our computational environment. Python was selected as our primary programming language because of its extensive libraries for data manipulation and its seamless integration with the OpenAI API.

The model's temperature was set to zero. The choice was made to guarantee deterministic performance from the model.

The top_p parameter was set to 1. This implies that at each stage of the generation process, the model will only take into account the tokens that are the most likely.

It should be emphasized that these combinations signify that we used the model outside of its intended parameters. We purposefully restricted the model to create consistent and repeatable results customized to our needs rather than utilizing its potential for creative and varied outputs. These settings came in helpful in situations where consistency and predictability were crucial.

Our method employed a split strategy that made use of both user prompts and system prompts. The user prompt constitutes the primary interaction point with the user. It is necessary to convert these user-provided free-form sentences into a (sequence of) more simplified structure. The model must understand these inputs robustly due to the inherent variation in how users phrase their queries or utterances. Free-form phrases can be anything from simple sentences to more complex thoughts or assertions, and the challenge lies in distilling the essence of what the user wants to communicate and converting it into a form that the model can process efficiently.

The system prompt serves primarily as a tool to direct the model towards a specific context or mode of operation. We directed the model's potential and ensured that we receive the desired output by creating a structured system prompt. It encompasses a chain-of-thought reasoning via (1) Question Answering, (2) Sentence Reframing, (3) Sentence Decomposition. Figure 1 shows the process flow with an example prompt for each step.

A series of iterative tests and comparisons with additional approaches, such as few-shot learning [21] and model fine-tuning [22], revealed that the suggested strategy performed better overall, especially with unrestricted sentence structures.

Let us consider an input text:

The apricot slows down at border. The rabbit turns into a diamond when hitting a carrot.

Here is a step-by-step trace through the outlined process using the provided input sentence.

1) Question Answering (QA): The QA component extracts crucial information from the input sentence by asking questions and taking the output in a specific format. It identifies the objects (apricots, rabbits, borders, diamonds), the default actions (apricots and rabbits move), and the conditional actions (speed decrease for apricots, transformation for rabbits).
2) Sentence Re-framing: Using the information from the above QA, the sentences are then re-framed according to a set of predefined rules that reflect the original free-form sentences. The main goal here is to use a specified set of rules to reconstruct the sentences in a paragraph, which are in their basic form in the format subject-verb-object. For example, stating the conditional actions of various objects: when apricots touch a border, their speed decreases, and when rabbits touch a carrot, they turn into diamonds.
   Re-framed sentence: If the apricot touches a border, the speed of the apricot decreases. If the rabbit touches a carrot, the rabbit turns into a diamond.
3) Sentence Decomposition: Next, the Sentence Decomposition step would break down complex sentences into simpler, object-oriented structures. The input would be analyzed to discern patterns of object interactions, such as the apricot's speed change upon touching a border, and the rabbit's transformation upon touching a carrot. An intermediate attribute "mutated" is added while decomposing the sentence resulting in the following sequence of unambiguous sentences [23].
   Decomposed sentence (Final Output): If the apricot touches a border, the speed of the apricot decreases. When the rabbit touches a carrot, the rabbit becomes mutated. When the rabbit is mutated, it turns into a diamond.

To sum up our methodology, it offers a comprehensive, structured, and systematic approach to interpret and process natural language text with a high degree of precision and consistency, enabling the user to more accurately describe their intent. Our innovation lies in the strategic application of existing LLM capabilities through a series of system prompts that guide the model to produce outputs in line with specific, predefined rules. This ensures that the transformations maintain the core meaning of the original sentences while stripping away unnecessary complexities, making the text more suitable for generating executable code.

Few-shot learning was initially considered due to its prowess in addressing edge cases with limited data. However, given the vast array of edge cases, rules, and potential issues to address in this domain, few-shot learning proved insufficient. The model would occasionally produce out-of-bound prompts leading to sub-optimal performance. In contrast, our proposed approach, which integrates QA, reframing, and sentence decomposition exhibits robustness against diverse sentence structures, making it an ideal choice for our purpose.

IV. Methodology II

In order to improve the model's capacity to handle the NL ambiguity and complexity, this next method implements a transformation to turn Free-form NL descriptions into JavaScript code. Even though the produced code may not be correct or instantly executable, this approach uses the formal structures of programming languages to clarify, disambiguate, and decompose the original NL input. The main goal of code generation is to act as an intermediate step that helps in the NL descriptions' rewriting to eliminate any inherent ambiguity and complexity.

Even though NL is intricate yet adaptable for human communication, it frequently has ambiguities and complexities that cause issues for computational tasks that demand accuracy and

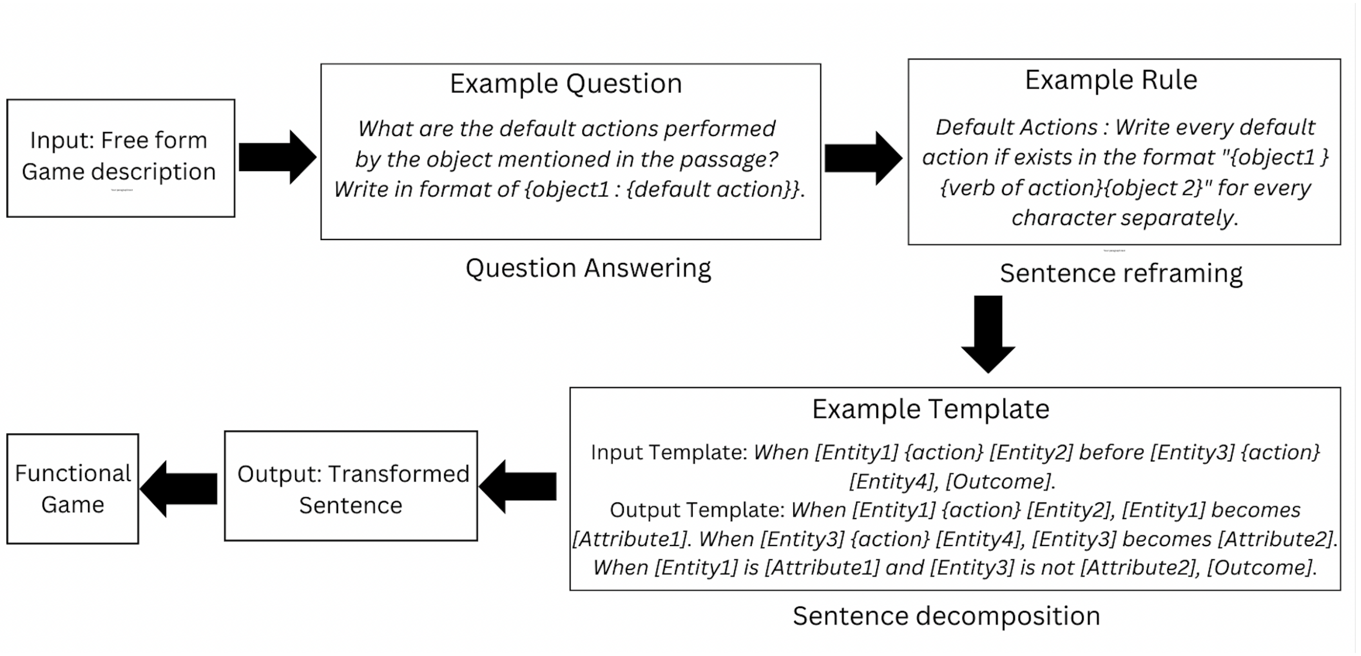
Input: Free form Game description
Example Question
What are the default actions performed by the object mentioned in the passage? Write in format of {object1 : {default action}}.
Question Answering
Example Rule
Default Actions : Write every default action if exists in the format "{object1 } {verb of action}{object 2}" for every character separately.
Sentence reframing
Example Template
Input Template: When [Entity1] {action} [Entity2] before [Entity3] {action} [Entity4], [Outcome].
Output Template: When [Entity1] {action} [Entity2], [Entity1] becomes [Attribute1]. When [Entity3] {action} [Entity4], [Entity3] becomes [Attribute2]. When [Entity1] is [Attribute1] and [Entity3] is not [Attribute2], [Outcome].
Sentence decomposition
Output: Transformed Sentence
Functional Game

Figure 1. Process flow with example prompt for each step.

precision, frequently found in many of engineering tasks [6]. Due to these ambiguities and complexities, translating NL directly into executable code without taking any additional steps may result in incorrect implementations or misinterpretations.

In order to address this, we first transform the NL descriptions into a structured code format as part of our strategy. This structured code refines the NL input by providing an unambiguous description of the game logic because of the strict syntactic rules of programming languages, even though the code may not be correct or executable. JavaScript was chosen for its alignment with event-driven game logic, though Prolog's declarative nature could aid in reasoning about constraints but may struggle with sequential operations. The process flow of the methodology is shown in Figure 2.

This approach uses a transformative process to translate game mechanic descriptions from natural language into structured JavaScript code and back again into refined, unambiguous natural language. This method accomplishes two goals: it first serves as a proof of concept for producing syntactically sound (but may be programmatically unsound) logic presented in the input text; second, it improves the original natural language description's understanding by using the explicitness of programming constructs to remove ambiguities.

Let us look at an example to understand the process flow. Below is the example user input written by middle school student on the GameChangineer platform:
"If the ball hits the sides, then it moves in the other direction."

1) Transformation into JavaScript Code: LLM converts the user input sentence into JavaScript using a system prompt. This transformation is enabled by a system prompt that explicitly incorporates a predefined vocabulary list. The incorporation of this vocabulary provides a structural foundation for the LLM, guaranteeing that the produced code accurately reflects the logic and semantics articulated in the students' natural language inputs. By restricting the LLM to utilize only the permitted vocabulary, we improve the accuracy of code generation and ensure uniformity across many implementations. The vocabulary is explicitly provided with the prompt during the code generating process. It consists of a curated compilation of nouns, verbs, adjectives, and other pertinent terminology related to game development and computational thinking. The generated code must accurately reflect the logic described in the natural language. This code serves as an intermediary to validate and refine the logical structure implied by the natural language description. The system prompt, intended to guide the transformation of natural language game descriptions into JavaScript, was carefully constructed to guarantee that the transformation adheres to a structured programming interpretation. The prompt provided to the LLM is articulated as follows:
"Write the following game idea described in free-form natural language into a javascript code using the allowed vocabulary given."
Embedded Contextual Guidance:: "In the context of a game description where objects and characters represent elements and entities in a programming language, actions and interactions symbolize functions, and conditions are akin to if statements. Consider the sentence strictly within this gaming and programming framework."
This context was directly incorporated into the system prompt utilized for code development, serving as a

guiding framework for the LLM. This guarantees that the LLM interprets the input description not simply as a narrative but as a structured set of instructions that can be directly converted into executable code. The predefined vocabulary includes game-specific entities (e.g., ”rabbit,” ”diamond”), actions (e.g., ”move,” ”touch”), and conditions (e.g., ”if,” ”when”), which constrain the model to operate within a defined semantic space, reducing ambiguity. By framing the game elements within this context, the LLM is primed to generate code that is logically consistent with game programming paradigms, hence improving the accuracy and relevance of the output regarding its intended application in game development.

Below is an example of JavaScript code generated by the LLM given a student's Game idea *”If the ball hits the sides, then it moves in the other direction.”* and the system prompt with the allowed vocabulary:

```
// Initialize variables
let ballX = 200;
let ballY = 200;
let ballSpeedX = 5;
let ballSpeedY = 5;

// Function to move the ball
function moveBall() {
   ballX += ballSpeedX;
   ballY += ballSpeedY;

   // Check if the ball hits the sides
   if (ballX >= 400 || ballX <= 0) {
      ballSpeedX = -ballSpeedX;
   }
   if (ballY >= 400 || ballY <= 0) {
      ballSpeedY = -ballSpeedY;
   }
}

//Call the moveBall function in a loop
setInterval(moveBall, 1000/60);
```

2) Retranslation into Structured Natural Language: The structured code is then retranslated into a refined NL description using the rules and the allowed Vocabulary using GPT 4. The generated javascript code is then passed with the system prompt and the context to the LLM to generate the final transformed Sentences. The example system prompt used for this step is:
”Write the game plan from the the code by using the allowed vocabulary below in subject-verb-object format using the rules.”
This step aims to articulate the logic captured in the code back into natural language, ensuring it is devoid of any ambiguities present in the original description. The rules used for this step is same as the rules used for the reframing step of Methodology I.
An Example rule for win/lose condition is:
”Win/Lose condition: write using ’if’ statement only. Write using the format ” Player win/lose when objectaction”
The final transformed game description is:
”There is a ball. The ball moves. When the ball hits the sides, it reverses direction.”

We utilized GPT-4 using the OpenAI API to convert NL descriptions into JavaScript code. The user prompt comprised the student's natural language input, and the system prompt had a predetermined vocabulary list that directed the LLM in code development. To ensure deterministic performance from the model and reduce variability in the outputs, we configured the model's temperature parameter to zero. Selecting a zero temperature guarantees that the model yields same outputs for identical inputs, which is essential for repeatability in a research setting. The top_p parameter was configured to 1, enabling the model to evaluate the complete probability distribution of all potential tokens during generation. This setup guarantees that the generated code is consistent and closely aligned with the input prompts, hence improving the dependability of the intermediate code representation in refining natural language descriptions.

We have incorporated our proposed technique into an educational platform named GameChangineer. This platform aims at developing computational thinking skills in students by encouraging them to think like computer scientists. GameChangineer converts English sentences written by students into functional games, therefore successfully connecting natural language with computational logic. In this work, we employed a dataset of such 1000 sentences written by middle school students, which were processed by GameChangineer to produce corresponding game implementations.

V. Evaluation I

This section evaluates the performance of the proposed AI-driven assistant in processing 1000 free-form sentences categorized into five types: (1) Grammar/typos, (2) Ambiguous, (3) Unrealizable actions, (4) Overly complex/descriptive, and (5) Non-problematic sentences. Sentences containing grammatical or typographical errors fall under the first category, ”Grammar or Typos” that could cause misinterpretations or inaccurate code translations. The second category, ”Ambiguity” refers to statements that have ambiguous references or meanings. Examples of this type of sentence include ”It chases it”, where pronouns make it difficult to determine exact entities and actions. The third category, ”Unrealizable Actions”, consists of sentences that describe actions not feasibly translatable into programming logic, exemplified by phrases like ”It jumps to heaven”. Sentences falling into the ”Overly Complex or Descriptive” category are weighed down with too many information or complex structures, which makes it difficult to translate them into concise, executable computer commands. Each of these categories represents a unique facet of the complexity inherent in translating natural language into machine-executable code. The final ”Non-problematic sentences” category refers to the sentences, which are successfully translatable by the GameChangineer platform into executable code [8] [9]

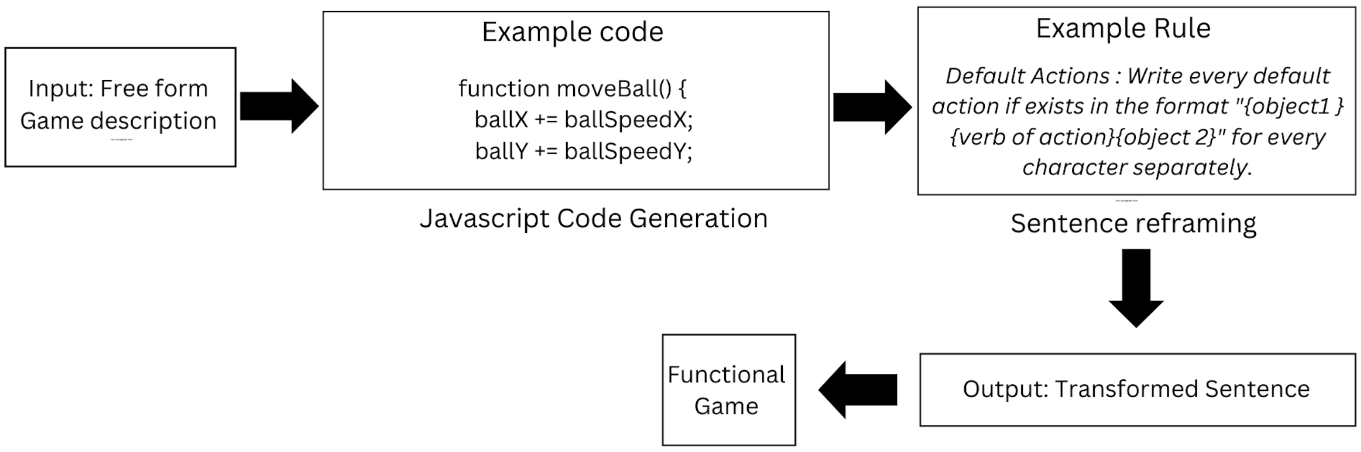
Input: Free form Game description
Example code
function moveBall() {
ballX += ballSpeedX;
ballY += ballSpeedY;
Javascript Code Generation
Example Rule
Default Actions : Write every default action if exists in the format "{object1 } {verb of action}{object 2}" for every character separately.
Sentence reframing
Output: Transformed Sentence
Functional Game

Figure 2. Process flow of Methodology II

[10]. These sentences are unambiguous and in object oriented structure.

There are several reasons why the final category of ”Non-problematic sentences” is included. It serves primarily as a benchmark, providing a point of comparison to assess the efficiency and precision of the AI-powered assistant while processing and interpreting texts that do not present inherent challenges. Furthermore, this category aids in determining whether and how Language Models (LMs) intervention may unintentionally add errors into previously error-free sentences. This will help in evaluating the preservation of sentence integrity after processing and is essential for preserving the overall quality and validity of the research.

The above categorization is based on the platform’s algorithms that use symbolic AI to detect grammatical errors, ambiguity, complexity, and unrealizable actions in sentences, indicating potential issues for translating these into executable code. The platform automatically logs the problematic sentences. All logged erroneous sentences are analyzed in this paper.

We discuss the effectiveness of the assistant in identifying and rectifying these issues, thereby enabling accurate translation into executable code. These sentences were written by middle school students with different degrees of experience in both natural language expression and game design when they were first created as parts of game descriptions. This diversity guarantees a wide range of linguistic difficulties, reflective of the intricacies typically seen in natural language programming.

These middle school students received a basic introduction to writing a few simple games with the GameChangineer platform. A small percentage of the students have prior programming experience. However, a vast majority of the students have never programmed before. Participants were given the following instructions to create their game plan: ”Write a game plan for creating a game utilizing the available characters.”

To ensure the accuracy and feasibility of the translated sentences produced by the LLM, they were given as an input into the GameChangineer platform [8]. This platform provides a score for each sentence that measures the compatibility with the platform’s expected input format [8] [9] [10]. Although some complex sentences can already be decomposed into a sequence of sentences by the GameChangineer platform, it cannot process all the nuances in natural language. We note that all the original problematic sentences were not accepted by the GameChangineer platform.

After the original input sentences were re-written by the LLM using our proposed approach, these new sentences underwent the validation process. Whenever the rewritten sentence(s) are understood with more than 90% certainty by the GameChangineer platform, the conversion will be regarded to have been translated correctly; on the other hand, when it falls below this mark, the output program generated may contain errors. The output program is generated by the GameChangineer Platform. The accuracy and relevance of the LLM-generated results were also assessed manually to ensure the translations effectively communicated the intended meaning. This dual evaluation provides a comprehensive measure called success rate of the AI assistant’s efficacy in translating complex natural language into machine-executable code by combining automated accuracy assessment with manual semantic verification.

Table I presents the results of the sentence categorization from the data-set, highlighting the success rate for each category. The table is divided into three main columns: Sentence Category, Number of Sentences, and Success Rate. These categories include Grammar/Typos, Ambiguous, Unrealizable Actions, Overly Complex/Descriptive, and Non-problematic. Note that when all 5 categories are considered, the success rate was more than 68%. However, if we consider only those first 4 categories (excluding the Non-Problematic category), the success rate by our approach is more than 60%.

The category of Unrealizable Actions encompasses game descriptions featuring actions that are either illogical or infeasible within the game context. This category was the least represented in student written game descriptions, accounting for its minimal proportion. All erroneous sentences are logged by the platform and the distribution of categories reflect the distribution of error types. The presence of actions that cannot be executed within the game complicates the task of rewriting such sentences, contributing to the lowest success rate observed for this category.

A sample of successful and unsuccessful transformations are listed in Table II. In the following, we will explain a few specific cases.

Let us first examine a successful transformation:
Input Sentence: "Each carrot you eat it gives you 1o speed of the rabbit."
Transformed Sentence: "When the player eats a carrot, the speed of the rabbit increases by 10."

The rewritten sentence establishes a clear subject "the player" and action "eats a carrot" by explicitly stating "When the player eats a carrot" as the condition. The usage of "it" in the original sentence lacks a clear antecedent, which could cause parsing issues. This ambiguity is eliminated in the rewritten sentence by defining the action and its result clearly without the use of potentially confusing pronouns. The phrase "the speed of the rabbit increases by 10" explicitly describes the effect of the action in a structured way that is easy for a machine to parse. This approach makes it easy to transform into a computational operation because it explicitly states the change ("increases by 10") and the attribute being impacted ("the speed of the rabbit").

Let us look at a few examples that show how the AI-driven assistant can recognize and fix typos and grammatical problems in sentences.
User Input: "When Dis pressed the bird moves right."
Transformed Sentence: "When the D key is pressed, the bird moves right."

The original sentence contained a typo ("Dis" instead of "D") and lacked a verb, making it unclear. The AI model successfully recognized the typo, corrected it to "D", and inserted the necessary word "key" to complete the sentence.
User Input: "When the rabbit is silver, the speed of the rabbit."
Transformed Sentence: "If the rabbit is silver, the speed of the rabbit increases."

The user input was incomplete, with no clear action described for the rabbit. The AI assistant inferred the most logical action "speed increase" based on the context, making the instruction actionable for programming.

Consider one unsuccessful transformation, which could not adhere to the intent of the user.
User Input: "When the rabbit is yellow, the rabbit touches the fox, the fox explodes."
Transformed Sentences: "If the rabbit is yellow, the rabbit touches the fox. If the rabbit touches the fox, the fox explodes."
Intended Output: " If the rabbit is yellow and the rabbit touches the fox, the fox explodes."

The input is incorrectly divided into two conditional statements by the original transformation. This method falsely implies that the fox's explosion is a two-step process that depends on the requirements being satisfied sequentially as opposed to simultaneously. The intended output, on the other hand, combines the two conditions into a single compound condition meaning that the fox will explode if both conditions are met simultaneously and directly. This showed that the input sentence is ambiguous and the AI-assistant could not successfully transform the sentence.

Let us look at an unsuccessful example in the fifth category, Non-problematic sentences.
User Input: "When a ball sees the rock, the ball flees from the rock."
Transformed sentences: "When the ball sees the rock, the ball becomes scared and flees from the rock."

The transformed sentence is considered unsuccessful here, primarily due to the addition of an unwanted attribute "scared" to the output sentence. This is an example where the LLM hallucinated leading to add an extra and unnecessary attribute [24]. Such hallucinations can significantly impact the utility and accuracy of LLMs, especially in applications requiring strict adherence to input data without the addition of interpretative or speculative elements. LLMs occasionally "hallucinate," or provide missing information [24]. We found that unsuccessful conversions due to hallucination account for 6% of Non-problematic sentences. For the problematic sentences in the other four categories, hallucination is responsible for about 12% of the unsuccessful transformations.

We did not compare our results with LLM based code generation platforms such as Copilot [25] because our goal is to rewrite erroneous sentences so that they become clear and unambiguous. On the other hand, while Copilot may be able to generate code on an erroneous sentence, it generates the code by its own interpretation arbitrarily. In addition, GameChangineer can process hundreds of sentences at a time, but the user must interface Copilot differently by feeding a few sentences at a time.

TABLE I. SENTENCE CATEGORIZATION RESULTS

Category	**Sentence proportion**	**Success Rate**
Grammar/Typing mistakes	22%	75.6%
Ambiguous Sentences	18%	57.7%
Unrealizable Actions	11.6%	50%
Overly Complex/Descriptive	28.4%	59%
Non-problematic sentences	20%	94%
Total Success Rate	100%	68.4%

TABLE II. INPUT AND OUTPUT SENTENCES

Input Sentence (Free-form)	Output Sentence	Successful?
fox explodes rabbit when it is the rabbit is touched by it.	If the rabbit is touched by the fox, the fox explodes.	Successful
When the fox touches the carrot before the rabbit, the fox dies.	When a fox touches a carrot, the fox becomes happy. When a rabbit touch the carrot, the rabbit becomes happy. When the fox is happy and the rabbit is not happy, the fox dies	Successful
if the fox sees the rabbit and the fox chases it and catches it the rabbit explodes.	When the fox sees the rabbit, it chases the rabbit. When the fox catches the rabbit, the rabbit explodes.	Successful
Rabbit move with wsad.	When the W key is pressed, the rabbit moves up. When the S key is pressed, the rabbit moves down. When the A key is pressed, the rabbit moves left. When the D key is pressed, the rabbit moves right.	Successful
The kitten must touches all the cakes to win without having the fox touch the kitten.	When the kitten touches all the cakes, the player wins the game. If the fox touches the kitten, the player loses the game.	Successful
When cake is touched by rabbit the speed of the rabbit is 0.5 for 3 seconds and cake disappears.	When the rabbit touches the cake, the rabbit's speed becomes 0.5 for 3 seconds. When the rabbit touches the cake, the cake disappears.	Successful
When the ball touches the bottom border it gains 1 point.	When the ball touches the bottom border, the score adds 1.	Successful
the pointer shoot the ball with the spacebar.	The pointer shoots the ball when the spacebar is pressed.	Successful
The rabbit dies when it touches the diamond for 5 seconds.	When the rabbit touches the diamond for 5 seconds, the rabbit becomes dead. When the rabbit is dead, it dies.	Successful
The rabbit must flee from the fox.	If the rabbit sees the fox, the rabbit flees.	Successful
the sheep is trying to eat all the broccoli.	If the sheep sees broccoli, the sheep eats broccoli.	Successful
When a tiger is shot for the 2 times, it disappears.	If a tiger is shot twice, it becomes mutated. When a tiger is mutated, it disappears.	Successful
if you do not touch the apricot within 3 minutes game is over.	If the apricot is not touched within 3 minutes, the game is over.	Unsuccessful
If the donut is blue and W is pressed, the donut moves up	There is a donut. When the W key is pressed, the donut moves up 10. If the donut is blue, then the color of the donut changes.	Unsuccessful
fox explode 20 seconds after start.	There is a fox. If the start is 20 seconds, the fox explodes.	Unsuccessful
The panda kills foxes when it reaches a score of greater than or equal to 14.	If the score is greater than or equal to 14, the score adds 1.	Unsuccessful
There are 3 snakes going side to side at 3 pixels per frame.	The snakes move side to side at 3 pixels per frame.	Unsuccessful

VI. Evaluation II

The evaluation for methodology II examines the functionality of a novel artificial intelligence (AI) assistant that converts sentences written in free-form natural language into unambiguous JavaScript code and back again into structured natural language. By stripping away the inherent ambiguities in human language, this approach seeks to increase the reliability of the generated code. The dataset used to evaluate the AI assistant's effectiveness consists of 1000 game descriptions written by middle school students with different degrees of programming and language proficiency. These descriptions were divided into categories according to the kind of linguistic difficulty they posed, such as ambiguity, complex syntax, or unrealistic scenarios. The dataset used is same as the one used for Evaluation I. The sentences were classified into similar categories as in the previous work as shown in Figure 3:

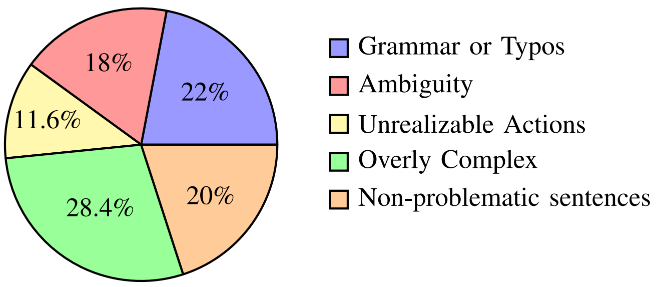

Figure 3. Distribution of Sentence into four Categories

- Ambiguous: Sentences with unclear references or multiple interpretations.
- Complex/Over-descriptive: Sentences that are verbose or syntactically complex.
- Unrealistic Actions: Sentences describing actions not feasible in the code or game environment.
- Non-problematic: Sentences that are straightforward and require minimal transformation.

Key metrics used for evaluation include:

- Success Rate: More than 76% of sentences were effectively converted using both methodologies I and II, representing a substantial enhancement compared to 68% of using methodology I alone. The success rate was determined by assessing the converted sentence's adherence to both grammatical accuracy and semantic intent, as required by the GameChangineer platform.
- Semantic integrity: It was verified through manual evaluation, which demonstrated that the modified phrases preserved the original intent and meaning. The results indicated that over 76% of the transformed sentences retained their original meaning and intent, marking a significant improvement over previous iterations.
- Handling Complex Sentences: An important aspect that showed progress was the model's capacity to process sentences with intricate structures, such as conditional clauses and multi-part instructions. The fine-tuned model decreased the failure rate for these specific sentences by 21%, effectively resolving a significant drawback of the previous method.

The evaluation results of Methodology II in Table III, demonstrate a significant enhancement in handling the ambiguous and complex sentences compared to Methodology I. The results from Methodology II, shows a higher total success rate of 76.5%. The success rate for sentences classified as Ambiguous and Overly Complex/Descriptive experienced saw significant improvements. The introduction of an intermediate JavaScript code transformation step in Methodology II appears to have enhanced the AI's ability to clarify and structure the sentences, reducing the ambiguity and simplifying complex descriptions effectively. This is reflected in the higher success rates reported in these categories. This improvement suggests that the structured nature of JavaScript code helps in clearly defining the game mechanics, which can then be more easily translated back into natural language, reducing ambiguities and simplifying complex sentence structures.

Let us look at a successful transformation example. This example depicts a game scenario in which several foxes chase a rabbit attempting to acquire a diamond. The fundamental mechanics of the game consist of unpredictable movement for the rabbit and calculated movement for the foxes as they chase the rabbit. If the rabbit reaches the diamond first, the player wins. If any fox catches the rabbit, the game ends.
Original Game Idea: *”The foxes will chase the rabbit, and the foxes try not to let the rabbit get the diamond.”*
Once the LLM has identified the key components and interactions, it begins by transforming the natural language description into code. This process involves breaking down the game mechanics into discrete functions and variables. For example:

- Defining Variables: The rabbit, foxes, and diamond are defined as objects with specific properties like position (x, y coordinates) and status (whether the rabbit has the diamond).

```
   let rabbit = { name: "rabbit",
       position: { x: 0, y: 0 },
       hasDiamond: false };
   let diamond = { position: { x: 5, y: 5
       } };
   let foxes = [
   { name: "fox1", position: { x: 0, y: 0
       } },
   { name: "fox2", position: { x: 0, y: 0
       } },
   { name: "fox3", position: { x: 0, y: 0
       } }
   ];
```

TABLE III. SENTENCE CATEGORIZATION RESULTS

Category	Sentence proportion	Success Rate
Grammar/Typing mistakes	22%	75.6%
Ambiguous Sentences	18%	70%
Unrealizable Actions	11.6%	50%
Overly Complex/Descriptive	28.4%	80%
Non-problematic sentences	20%	94%
Total Success Rate	100%	76.5%

- Creating Functions for Movement: The LLM generates functions to move the rabbit randomly and to move the foxes towards the rabbit.
- Game Logic and Conditions:The LLM generates the necessary logic to check if the rabbit has reached the diamond or if a fox has caught the rabbit. This includes conditions to end the game upon success or failure. The resultant JavaScript code executes these mechanisms.
  A part of the generated JavaScript code is mentioned below:

```
// Function to check if a fox caught the
    rabbit
function checkFoxesCaughtRabbit() {
   foxes.forEach(fox => {
      if (fox.position.x ===
          rabbit.position.x
      && fox.position.y ===
          rabbit.position.y)
      {
         console.log("fox caught the
             rabbit!");
         // Game over logic here
      }
   });
}

// Function to check if the rabbit got
    the diamond
function checkRabbitGotDiamond() {
   if (rabbit.position.x ===
       diamond.position.x
   && rabbit.position.y ===
       diamond.position.y)
   {
      rabbit.hasDiamond = true;
      console.log("Rabbit collected the
          diamond!");
      // Win logic here
   }
}
```

Subsequent to code generation, the following phase involves transforming it into a series of straightforward subject-verb-object phrases, adhering to the allowed vocabulary and established rules. These rules break the actions into default behaviors and interactions using if/when statements.
The JavaScript code implements the game logic for the interaction between the rabbit and the foxes through functions including moveRabbit(), moveFoxes(), checkFoxesCaughtRabbit(), and checkRabbitGotDiamond(). These functions define the actions and conditions inside the game, including the rabbit's random movement, the foxes' pursuing the rabbit, and the win/loss conditions dependent upon the rabbit acquiring the diamond or being captured by a fox.
To convert this into NL, the fundamental actions and interactions have been simplified into straightforward subject-verb-object phrases and presented as conditional interactions. The movement Conditions for the rabbit and foxes are articulated as follows: ”The rabbit moves

randomly" and "The foxes move towards the rabbit." The winning and defeat criteria—determined by whether the foxes catch the rabbit or the rabbit obtains the diamond—are articulated as follows: "When the rabbit touches the diamond, the player wins." and "When a fox catches the rabbit, the game is over" By following this two-step process, we achieve the final output. *"Final Output: There is a rabbit. There are 3 foxes. There are 2 diamonds. The rabbit moves randomly. The foxes move towards the rabbit. When a fox catches the rabbit, the game is over. When the rabbit touches the diamond, the player wins."*

- A high success rate in effectively transforming intricate descriptions into JavaScript code, demonstrating the AI's aptitude for logic-based programming.
- enhanced readability and clarity in the retranslated sentences, with decreased ambiguity.

The transformation of ambiguous and complex natural language into code and then back into refined language proved effective in clarifying the original intent and reducing linguistic ambiguities. This dual transformation approach leverages the structured nature of programming languages to impose clarity and precision that natural language typically lacks. Despite the improvements made, some challenges remain: The AI sometimes struggled with sentences that contained parts of the sentences, which could not be directly translated into code.

VII. Conclusion and Future Work

This paper presents a method for converting free-form natural language sentences into a sequence of unambiguous, simplified sentences that can subsequently be translated into machine-executable code. The utilization of LLMs has shown promise in addressing the inherent difficulties brought about by verbosity, ambiguities, complexity, and possible errors. Our approach in methodology I, which combines aspects of Question Answering, Sentence Reframing, and Sentence Decomposition has demonstrated a notable capacity to handle a wide variety of linguistic patterns and semantic complexities. More than 68% of the 1000 problematic and non-problematic sentences were correctly converted by the proposed method.

In our Methodology II, we convert complex and ambiguous natural language input into JavaScript code. This code then restructures the original input into a series of clear, unambiguous sentences. This approach has demonstrated an improvement in the success rate. 76% of the user input sentences were accurately transformed into a series of unambiguous sentences using methodology I and methodology II.

There are areas for improvement, particularly in understanding complex conditional relationships and refining the LLM methodologies, aiming to reduce the incidence of hallucinations. Future work includes reducing LLM hallucinations, adapting transformation rules dynamically, expanding to other platforms, incorporating user feedback, testing scalability on diverse datasets, and developing interactive educational tools to foster computational thinking. Additionally, they draw attention to how AI-powered systems have the potential to greatly enhance our comprehension and interpretation of words with unclear structures, which is an important area of study in the field of natural language programming.

ACKNOWLEDGEMENT

This research is supported in part by NSF grant 2101021.

References

[1] N. K. Yeole and M. S. Hsiao, "Bridging natural language and code by transforming free-form sentences into sequence of unambiguous sentences with large language model," in *eLmL 2024*. IARIA, 2024, pp. 4–10, retrieved: September, 2024.

[2] C. Yang, Y. Liu, and C. Yin, "Recent advances in intelligent source code generation: A survey on natural language based studies," *Entropy*, vol. 23, no. 9, p. 1174, 2021.

[3] D. Baidoo-Anu and L. Owusu Ansah, "Education in the era of generative artificial intelligence (AI): Understanding the potential benefits of chatgpt in promoting teaching and learning," *SSRN Electronic Journal*, January 2023, published by Elsevier BV, retrieved: March, 2024, Available at SSRN: https://ssrn.com/abstract=4337484 or http://dx.doi.org/10.2139/ssrn.4337484.

[4] T. P. Tate, S. Doroudi, D. Ritchie, Y. Xu, and M. W. Uci, "Educational research and AI-generated writing: Confronting the coming tsunami," January 2023, published by Center for Open Science, retrieved: April, 2024. [Online]. Available: https://doi.org/10.35542/osf.io/4mec3

[5] D. Mogil et al., "Generating diverse code explanations using the GPT-3 large language model," in *ICER '22: Proceedings of the 2022 ACM Conference on International Computing Education Research*. Association for Computing Machinery, 08 2022, pp. 37–39.

[6] F. F. Xu, B. Vasilescu, and G. Neubig, "In-IDE code generation from natural language: Promise and challenges," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, mar 2022, retrieved: April, 2024. [Online]. Available: https://doi.org/10.1145/3487569

[7] C. Niklaus, "From complex sentences to a formal semantic representation using syntactic text simplification and open information extraction," Ph.D. dissertation, 03 2022, retrieved: April, 2024. [Online]. Available: https://opus4.kobv.de/opus4-uni-passau/frontdoor/index/index/docId/1054

[8] M. S. Hsiao, "Automated program synthesis from object-oriented natural language for computer games," in *Proceedings of the Controlled Natural Language Conference*, August 2018.

[9] ——, "Multi-phase context vectors for generating feedback for natural-language based programming," in *Controlled Natural Language*, September 2021.

[10] ——, "Automated program synthesis from natural language for domain specific computing applications," Patent 10 843 080, November, 2020.

[11] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024, arXiv.

[12] F. Shi, D. Fried, M. Ghazvininejad, L. Zettlemoyer, and S. I. Wang, "Natural language to code translation with execution," 2022, arXiv.

[13] T. H. M. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys*, vol. 53, no. 3, p. 1–38, Jun. 2020, retrieved: April, 2024. [Online]. Available: http://dx.doi.org/10.1145/3383458

[14] A. Vaswani et al., "Attention is all you need," in *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, pp. 5998–6008, retrieved: April, 2024. [Online]. Available: https://arxiv.org/abs/1706.03762

[15] M. L. Zong and B. Krishnamachari, "A survey on GPT-3," 2022, arXiv, retrieved: April, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:254221221

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics*, 2019, retrieved: April, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:52967399

[17] Y. Liu et al., "Roberta: A robustly optimized BERT pretraining approach," 2019, CoRR, abs/1907.11692, retrieved: April, 2024. [Online]. Available: http://arxiv.org/abs/1907.11692

[18] J. Wei et al., “Chain-of-thought prompting elicits reasoning in large language models,” in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 24 824–24 837.

[19] J. Akanya and C. G. Omachonu, “Meaning and semantic roles of words in context,” *International Journal of English Language and Linguistics Research (IJELLR)*, vol. 7, pp. 1–9, 03 2019.

[20] M. Rosoł, J. S. Gasior, J. Łaba, K. Korzeniewski, and M. Młyńczakl, “Evaluation of the performance of GPT-3.5 and GPT-4 on the polish medical final examination,” *Scientific Reports*, vol. 13, no. 1, p. 20512, 2023, retrieved: April, 2024. [Online]. Available: https://doi.org/10.1038/s41598-023-46995-z

[21] T. Brown et al., “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[22] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, “Training language models to follow instructions with human feedback,” in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 27 730–27 744.

[23] N. Wies, Y. Levine, and A. Shashua, “Sub-task decomposition enables learning in sequence to sequence tasks,” 2023, arXiv.

[24] Z. Ji et al., “Survey of hallucination in natural language generation,” vol. 55, no. 12. Association for Computing Machinery (ACM), Mar. 2023, pp. 1–38, retrieved: April, 2024. [Online]. Available: http://dx.doi.org/10.1145/3571730

[25] GitHub, “About github copilot,” 2024, retrieved: April, 2024. [Online]. Available: https://docs.github.com/en/copilot/about-github-copilot