Leveraging LLMs for Formal Grammar Generation in Programming Competition Testing

Anonymous ACL submission

Abstract

Test cases are crucial for ensuring the program's correctness and evaluating performance in programming. The high diversity of test cases within constraints is necessary to distinguish between correct and incorrect answers. Automated source code generation is currently a popular area due to the inefficiency of manually generating test cases. Recent attempts involve generating conditional cases from problem descriptions using deep-learning models that learn from source code. However, this task requires a combination of complex skills such as extracting syntactic and logical constraints for a given test case from a problem, and generating test cases that satisfy the constraints. In this work, we introduce a modified context-free grammar that explicitly represents the syntactical and logical constraints embedded within programming problems. Our innovative framework for automated test case generation separates restriction extraction from test case generation, simplifying the task for the model. Our experimental results show that, compared to current methods, our framework produces test cases that are more precise and effective. All the codes in this paper are available in `https://anonymous.4open.science/r/input_spec_generation-35A9`.

1 Introduction

Automated test case generation (ATCG) is a growing area of interest within software engineering, particularly as advancements in deep learning continue to reshape our technological framework. The intersection of machine learning and software development has catalyzed the emergence of innovative tools that boost programming productivity and refine software quality through intelligent code suggestions. These tools, including ATCG, are engineered to assist developers by predicting and auto-generating segments of code that align with user requirements and established coding standards. ATCG, in particular, plays a crucial role in validating the functionality and reliability of these machine-generated code suggestions.

ATCG plays a crucial role in validating the functionality and reliability of these machine-generated code suggestions. By enabling automatic production of diverse test scenarios, ATCG ensures that the software behaves as expected under a wide range of conditions, thus mitigating the risk of defects in production. However, it is vital to underscore the inherent limitations of this approach. While ATCG significantly aids in detecting flaws and ensuring code robustness, passing these generated test cases does not conclusively guarantee the correctness of the program. Recently, Liu et al. (2023) reported that there were incorrect codes in the current program, synthesis benchmarks is sufficiently verified due to the need for test cases. To generate more test cases, they employed the famous large language model (LLM) ChatGPT by OpenAI to produce the initial test cases by providing several prompts, such as 'generate difficult inputs' or 'generate corner-case inputs' Moreover, type-aware mutations apply to the generated test cases to cover all the corner cases. Also, Li et al. (2023) uses the test cases to explore the fault in a given program, hence pointing out the importance of high-quality test cases. MuTAP (Dakhel et al., 2024) introduces generating effectively using pre-trained LLMs and mutation testing by augmenting the prompts with the initial test cases of a program along with the surviving mutants of a program under test.

In this research, we concentrate on the competitive programming field, This presents substantial demands for ATCG technologies to score solutions and efficiently differentiate between correct and incorrect solutions submitted by various programmers to obtain high-quality assessments, which solely depend on the quality of the test cases. To address the importance and the need for high-quality test cases, we introduce a pioneering neural translation task that effectively converts natural language specifications or descriptions of a problem into formal grammar. Our innovative approach not only captures the structural and semantic intricacies of the problem specification but also eliminates the need for additional test case generation. By directly translating to formal grammar, our model focuses exclusively on refining its natural language comprehension during the learning phase, enhancing both its accuracy and efficiency. This practical application of neural translation in test case generation holds significant potential for improving the quality and efficiency of software testing. We rely on formal grammar-based sampling algorithms to generate test cases. Although, to represent a collection of organized texts, context-free grammar (CFG) is frequently employed rather than the other powerful grammars such as CSG (context-sensitive grammar) because CFGs enable more straightforward and more efficient parsing algorithms, essential for practical compiler and interpreter design. However, more information is needed to represent a test case structure.

Our contribution extends to introducing context-free grammars with counters (CCFGs), meticulously designed to represent the syntactic and logical constraints that arise in competitive programming problem descriptions. For the execution of our neural translation task, we utilize the pre-trained CodeT5 model, which was selected for its demonstrated proficiency in code comprehension and generation. The effectiveness of our approach is rigorously validated through experiments conducted on the publicly accessible DeepMind's CodeContests dataset (Li et al., 2022b). Figure 1 illustrates our approach, depicting the seamless transition from problem description to the generation of test cases. Our contributions to the field of competitive programming not only advance the state-of-the-art in neural translation tasks but also set new benchmarks for the formulation and solution of coding problems in both competitive and educational settings.

2 Related work

2.1 Automatic Test Case Generation

A review of the existing work done has indicated that the use of ATCG has shown significant improvement in the generation of grammar-based test case generation. Reports show that the tools for automatically generating the test cases are becoming one of the standard practices in large software organizations. The use of ATCG can enhance the efficiency and the ad-hoc in the software engineering field (Brunetto et al., 2021). However, one of the key challenges is the navigation of the large input space, and all the existing works struggle or find difficulty in generating the high quality and well-structured test case (Olsthoorn, 2022). Typically, Salman (2020) studied the process of generation of test cases using test case specifications by generating the scripts in C# for testing the generated test cases are valid by focusing on the extraction of the feature vectors present in the specifications. Alternatively, a deep learning approach takes an input specification as input and trains the model to get an accurate test case using a neural network. Previously, A3Test has used the existing knowledge from an assertion generation task to the test case generation task (Alagarsamy et al., 2023). Similarly, Wang et al. (2022) have relied on the specifications from the natural language to generate the test cases by extracting the constraints and thus reducing the manual errors for generating the test cases.

2.2 Natural Language to Formal Grammar

Many approaches have been made to convert the natural language to the grammar. For example, Kate et al. (2005) implemented a method for inducing transformation rules that map natural-language sentences into a formal query or command language. Recently, research of Chen et al. (2023) has shown that semantic regex's can better support complex data extraction tasks than standard regular expressions and the use of the regular expressions or had significantly outperformed the existing tools, including the state-of-the-art neural networks and program synthesis tools. However, previous works also mentioned the drawback of using the regular expression as the study shows that the conversion of the natural language to the regex fails to generate complex regex's. Ye et al. (2020) solved these issues by introducing the semantic parser that can be trained purely from the weak supervision based on the correctness of the synthesized regex. Similarly, Hahn et al. (2022) mentioned that the language models have the capability to translate the natural language to the formal specifications while maintaining the important keywords as well as outperforming the state of the art of using the regular expressions, without a particular need for domain-specific reasoning.

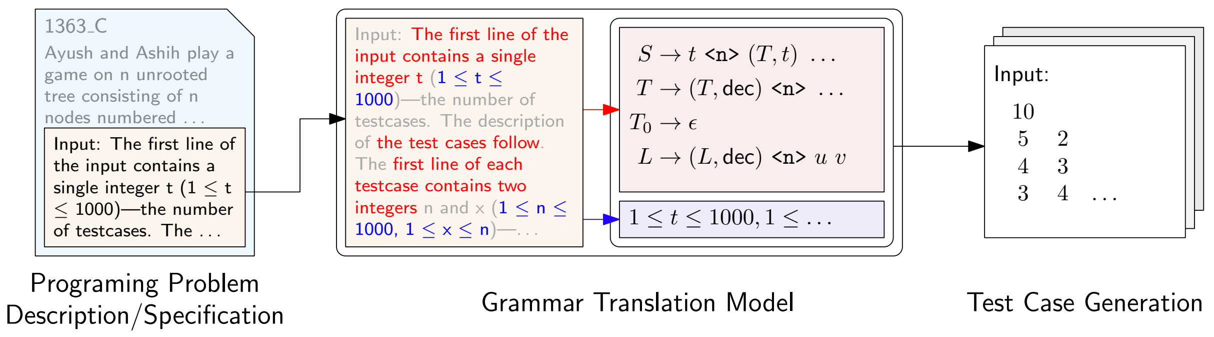
1363_C
Ayush and Ashih play a game on n unrooted tree consisting of n nodes numbered ...
Input: The first line of the input contains a single integer t (1 ≤ t ≤ 1000)—the number of testcases. The ...
Programing Problem Description/Specification
Input: The first line of the input contains a single integer t (1 ≤ t ≤ 1000)—the number of testcases. The description of the test cases follow. The first line of each testcase contains two integers n and x (1 ≤ n ≤ 1000, 1 ≤ x ≤ n)—...
S → t <n> (T, t) ...
T → (T, dec) <n> ...
T_0 → ε
L → (L, dec) <n> u v
1 ≤ t ≤ 1000, 1 ≤ ...
Grammar Translation Model
Input:
10
5 2
4 3
3 4 ...
Test Case Generation

Figure 1: Overview of the proposed framework for generating test cases for competitive programming problems: The deep learning model translates specifications into CCFGs while preserving their meaning. Subsequently, the CCFGs are utilized to generate test cases for the problem.

2.3 LLMs for Program Understanding and Generation

Recently, there have been a plethora of pre-trained language models trained on codes proposed for various types of programming language understanding and generation tasks. Feng et al. (2020) proposed CodeBERT, a RoBERTa-based model pre-trained on multiple programming languages with masked language modeling. Guo et al. (2021) introduced GraphCodeBERT, which is strengthened from CodeBERT by incorporating data flow information in the pre-training stage. Jiang et al. (2021) introduced TreeBERT, a tree-based pre-trained model that utilizes the extracted tree structure by encoding an abstract syntax tree as a set of composition paths. TreeBERT is trained by two novel objectives: tree-masked language modeling and node order prediction. Rozière et al. (2021) investigated another programming-language-oriented pre-training objective based on the de-obfuscation of identifier names in source code.

Recently, Ahmad et al. (2021) proposed PLBART—Program and Language BART—which learns the interaction between program codes and natural language descriptions by leveraging the idea of denoising auto-encoder that uses a bidirectional encoder and an auto-regressive decoder. Wang et al. (2021) introduced CodeT5, which leverages the code-specific characteristics in the pre-training stage by employing the new objectives such as masked random token prediction, masked identifier prediction, and identifier prediction objectives. Ma et al. (2021) proposed a pre-trained multilingual encoder-decoder model that regards the decoder as the task layer of off-the-shelf pre-trained encoders in order to take advantage of both the large-scale monolingual data and bilingual data. Guo et al. (2022) proposed UniXcoder, a unified cross-modal pre-trained model for programming language that utilizes mask attention matrices with prefix adapters to control the behavior of the model and leverage cross-modal contents such as an abstract syntax tree, and enhances the code representation by retaining all the structural information from the tree.

3 Methodology

In this section, we present a formal definition of *context-free grammars with counters (CCFGs)*, specialized formal grammar designed to accurately represent the semantics of input specifications in the context of competitive programming. For symbols that occur in each input specification, (1) *variables* are symbols used to represent values that vary depending on the test case, and (2) *terminals* are the symbols that are not variables, such as white spaces or newline symbols.

Example 3.1 (Input Specification). *The first line contains an integer* m $(1 \leq m \leq 100)$. *The second line contains* m *integers* $x_1, \ldots, x_m$ $(0 \leq x_i \leq 9)$.

Consider the input specification in Example 3.1. Then, symbols m and x are variables of the specification, and $1 \leq m \leq 100$ and $0 \leq x_i \leq 9$ are constraints of these variables.

Let V be a finite set of *variables* and denote the set $V \times \mathbb{N}_0$ by $V_{\mathbb{N}_0}$. A V*-assignment* is a partial function $\alpha : V_{\mathbb{N}_0} \to \mathbb{N}_0$, and $\mathcal{A}_V$ denotes the set of all V-assignments. Intuitively, when a V-assignment α is fixed, each variable $X \in V$ denotes an integer array whose indices is counters. In this case, the value of $(X, i) \in V_{\mathbb{N}_0}$ is $\alpha(X, i)$. We denote $(X, i) \in V_{\mathbb{N}_0}$ by $X[i]$ to reflect this intuition. If $\alpha(X[i]) = n$, we say $X[i] \mapsto n \in \alpha$. The following example describes a relation between a test case and assignments.

Example 3.2. *For the input specification in Example 3.1, consider the following test case.*

```
9
5 4 9 1 2 3 0 4 8
```

Then, the corresponding assignment α is

$$\{m[0] \mapsto 9, x[1] \mapsto 5, x[2] \mapsto 4, \ldots, x[9] \mapsto 8\}.$$

Suppose that we have an assignment α, and we want assign a value $n \in \mathbb{Z}$ to $X[i]$. Then, we have to consider a new assignment α' that reflects this change. For each $X[i] \in V_{\mathbb{N}_0}$ and $n \in \mathbb{N}_0$, we define a function $\mathsf{assign}(\cdot; X[c] \mapsto n)$ over the set of assignments $\mathcal{A}_C$, which maps α into such α'. Formally, the assignment $\alpha' = \mathsf{assign}(\alpha; X[i] \mapsto n)$ is defined as the following assignment: $\alpha'(X[i]) := n$ and $\alpha'(Y[j]) := \alpha(Y[j])$ if $Y \neq X$ or $j \neq i$.

A *(V-) counter* is a number in $\mathbb{N}_0$ or an indexed variable $X[i] \in V_{\mathbb{N}_0}$, in the latter case, a value of the counter is determined when an assignment is given. $C_V = \mathbb{N}_0 \cup V_{\mathbb{N}_0}$ is the set of all counters. A *counter operator* is one of following partial functions $\mathsf{dec}, \mathsf{nop}, \mathsf{set}_j$ and $\mathsf{set}_{X[k]}$, where (1) $\mathsf{dec} : i \in \mathbb{N} \mapsto i - 1$, (2) $\mathsf{nop} : i \in \mathbb{N}_0 \cup V_{\mathbb{N}_0} \mapsto i$, (3) $\mathsf{set}_j : c \in \mathbb{N}_0 \cup V_{\mathbb{N}_0} \mapsto j$ for some $j \in \mathbb{N}$, and (4) $\mathsf{set}_{X[k]} : c \in \mathbb{N}_0 \cup V_{\mathbb{N}_0} \mapsto X[k]$ for some $X[k] \in V_{\mathbb{N}_0}$. We denote the set of all operators of counters by O_V.

Definition 3.1. *Let V be a set of variables and $T \supseteq \mathbb{Z}$ be a set of terminal symbols, respectively. A CCFG is a tuple (N, V, T, P, P_0, S), where (1) N is a finite set of* nonterminals*, (2) $P, P_0 : N \to 2^{((N \times O_V) \cup V \cup T)^+}$ are sets of production rules, (3) $S \in N$ is an* initial nonterminal*. If $\gamma \in P(A)$ or $\gamma' \in P_0(A)$, then we say $A \to \gamma \in P$ or $A_0 \to \gamma' \in P_0$, respectively.*

The following is an example of a CCFG.

Example 3.3. *Let '<s>' and '<n>' be white space and newline symbols, respectively. The following tuple G is a CCFG for the input specification in Example 3.1:*

$$(\{S, A\}, \{m, x\}, \{\texttt{<s>}, \texttt{<n>}\} \cup \mathbb{Z}, P, P_0, S),$$

where

$$P = \{S \to m \cdot \texttt{<n>} \cdot (A, m), \quad A \to (A, \mathsf{dec}) \cdot \texttt{<s>} \cdot x\}$$

and $P_0 = \{A_0 \to \epsilon\}$.

A CCFG rewrites a string over $(N \times C_V) \cup V_{\mathbb{N}_0} \cup T$ according to production rules in P and a current assignment α, until it obtains a string over T. We call this rewriting *derivation*. The right-hand side of each production is a sequence of three types of components: (1) $N \times O_V$, (2) V and (3) T. We define a function $[\![\cdot]\!]_i$ for each $i \in \mathbb{N}_0$ as follows.

1. $[\![(A, o)]\!]_i = (A, o(i))$ for $(A, o) \in N \times O_V$,
2. $[\![X]\!]_i = X[i]$ for $X \in V$.

Note that $[\![\cdot]\!]_i$ converts each component of type (1) and (2) into a component of type $N \times C_V$ and $V_{\mathbb{N}_0}$, respectively. Thus, we can extend $[\![\cdot]\!]_i$ to convert the right-hand side of the production into a string over $(N \times C_V) \cup V_{\mathbb{N}_0} \cup T$, which is used for derivation.

Example 3.4. *Consider the CCFG G in Example 3.3. Then,*

$$[\![(A, \mathsf{dec}) \cdot x]\!]_3 = (A, 2) \cdot x[3].$$

Let u be a string that we want to rewrite, and x be the left-most component in u such that $x \in (N \times C_V) \cup V_{\mathbb{N}_0}$. Then, a derivation step of u is one of the followings according to x.

Nonterminal Production Suppose that $x \in (N \times C_V)$ and $x = (A, c)$. Let

$$i = \begin{cases} c & \text{if } c \in \mathbb{N}_0, \\ \alpha(c) & \text{if } c \in V_{\mathbb{N}_0}. \end{cases}$$

If $i = 0$, choose a production $A \to \gamma \in P$; otherwise, choose a production $A_0 \to \gamma \in P_0$. Then, rewrite x to $[\![\gamma]\!]_i$.

Example 3.5. *Consider the CCFG G in Example 3.3. When $\alpha = \{m[0] \mapsto 9\}$, G rewrites $(A, m[0])$ with $(A, 8) \cdot \texttt{<s>} \cdot x$.*

Variable Sampling We now suppose $x \in V_{\mathbb{N}_0}$ and $x = X[i]$. Then, we randomly choose a number $n \in \mathbb{Z}$ for the indexed variable $X[i]$, and rewrite x with n. Also, we update the current assignment α to $\alpha' = \mathsf{assign}(\alpha; X[i] \mapsto n)$.

For a CCFG G with the initial nonterminal S, if there is a derivation from $(S, 0)$ with an initial assignment $\emptyset$ that yields $w \in T^*$ with a final assignment α, then we say G *generates* w with the assignment α.

3.1 String Sampling and Parsing of CCFGs

Note that G itself does not consider the constraints of an input specification. We can restrict G to generate strings that satisfy the constraints by ensuring that the values of variables sampled during variable sampling do not violate these constraints.

Given a CFG G, one can sample a random string from $L(G)$ in linear time with respect to the length of the sampled string by simulating its derivation procedure. We can directly apply the sampling algorithm for CFGs to CCFGs, utilizing random assignment on variables during the derivation.

However, when the sample string are restricted to meet certain constraints, there are no polynomial-time sampling algorithms for CCFGs, even with simple constraints. The following theorem demonstrates that sampling of CCFGs under equality constraints is NP-hard.

Theorem 3.1. *For a given CCFG $G = (N, V, T, P, P_0, S)$ and a set C of equality constraints on V, let the set $L(G, C)$ be the set strings generated by G under constraints C. Then, determining whether or not $L(G, C) \neq \emptyset$ is NP-hard.*

Proof. We utilize a reduction from the graph coloring problem (Karp, 1972), which is an NP-hard problem. Consider an instance of the graph coloring problem represented as $((V, E), k)$, where $V = \{v_1, v_2, \dots, v_n\}$ denotes the vertices. We then construct a CCFG $G := (\{S\}, V, \mathbb{Z}, \{\}, P_0, S)$, where $P_0 = \{S_0 \to v_1 \cdot v_2 \cdot \cdots \cdot v_n\}$. A set of constraints C consists of $0 \leq v \leq k$ for $v \in V$, and $u \neq v$ for each $(u, v) \in E$. Then, The existence of a k-coloring for G is equivalent to the ability of G to generate a string that satisfies these constraints. □

In practice, we address the computational hardness of the constrained sampling by employing a straightforward Las Vegas approach, which involves sampling variables until they satisfy the constraints.

One can utilize CCFGs as parser for test case validation according to their specifications. For parsing, we search derivations in a depth-first way, and backtracks if the constraint or test case scheme violates the input string. Most of the grammars have linear parsing time with respect to the test-case length in real world, despite of the worst-case time complexity of the parsing algorithm being exponential. This is because most of the problem specifications are unambiguous, and so we can determine which derivation to choose for each step with 1-look ahead (Rosenkrantz and Stearns, 1969). Figure 2 describes a test case generation and validation using CCFGs.

4 Experiments

We evaluate the practical usefulness of CCFGs through experimental validation.

4.1 Dataset

We use the CodeContests dataset, which consists of various programming problems sourced from different competitive platforms (Li et al., 2022a). This dataset includes (1) both correct and incorrect solutions for programs in various programming languages, and (2) public, private and generated test cases for each problems. Additionally, we manually created CCFGs for 1,153 different from their descriptions, using 700 for model training and remaining 453 for evaluation.

4.2 Baselines

Fine-tuned CodeT5 model We fine-tune a pre-trained CodeT5 model using 700 human-labeled problems to generate CCFGs from problem input specifications. For comparative analysis, we utilize test cases from the CodeContests dataset as well as those generated by other LLMs.

Mutation-based fuzzing In competitive algorithm programming, test cases frequently adhere to a specific input format delineated at the end of the problem description. A common scenario involves the first line specifying the number of subsequent inputs, followed by a second line containing integers that match the number indicated on the first line. Alternatively, this could be a string whose length corresponds to the integer on the first line.

For efficient test case generation, it is often practical to maintain the integrity of spaces, string lengths, and newline characters, while varying the values of integers or the characters within the strings to comply with the input specifications. In our methodology, we utilize the public test cases from the CodeContests dataset, where we segment these based on spaces and newline characters. We then randomly select 30% of these tokens for mutation, adapting our approach based on the token type—be it integer, float, or string. This selective mutation process allows us to conduct effective

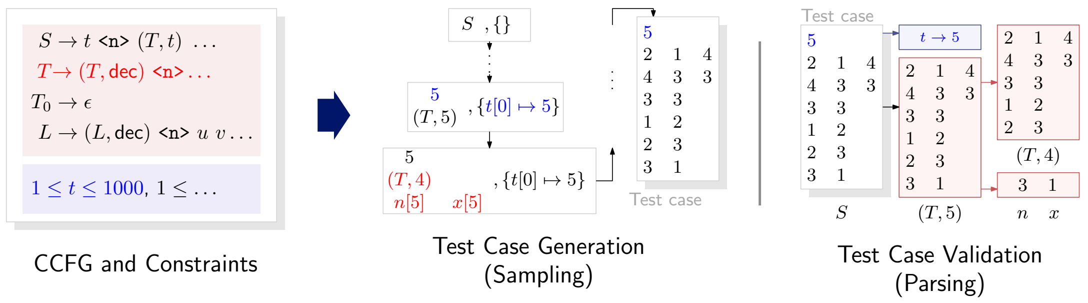

Figure 2: Test case generation and validation process using sampling and parsing algorithms of CCFGs.

fuzzing that still conforms to the original input specifications.

Large language models We employ two renowned LLMs, OpenAI's ChatGPT (version 4.0) and Google's Gemini, to facilitate our understanding and generation of CCFGs. The models utilize the concept of the chain of thought (CoT) (Wei et al., 2022) to enhance the clarity and effectiveness of the generation process. Specifically, we apply these LLMs in two key areas: (1) Developing a baseline neural transition model that converts problem specifications into CCFGs. (2) Creating a test case generation model tailored for competitive programming. To evaluate the efficacy of the test case grammars produced by these LLMs, we conduct performance tests using varying numbers of CoT examples, specifically one and five, to observe how the models' performance scales with increased contextual guidance.

4.3 Evaluation Metrics

To evaluate the model-translated CCFGs, we establish the following three distinct metrics: (1) *Sentence accuracy*, compare the model-translated CCFGs with human-labeled ones, (2) *Validity* measures the validity of test cases by using the ground-truth CCFGs, and (3) *Effectiveness* measures the ability to distinguish correct and incorrect source codes.

Sentence accuracy In assessing the performance of our model, we initially focus on the sentence accuracy between the CCFGs generated by the model and those annotated by human experts by comparing them as sentences instead of formal grammars.

Although the names of nonterminals of CCFGs do not inherently influence their semantic function, a straightforward comparison of productions may mistakenly evaluate CCFGs based on the nonterminal names. To address this issue, we standardize the representation of CCFGs before comparing them for equivalence. This normalization involves renaming nonterminals according to the sequence of their initial appearances in the list of productions, thereby mitigating unnecessary discrepancies due to naming conventions. After normalization, we generate top-k lists of productions and their respective outputs for each problem using our model. We then assess the congruence of these results against human-labeled data by evaluating for exact matches.

Validity and generality We define a test case to be *valid* if the test case can be generated by a CCFG that corresponds to the logical input specification of the problem. As the public and private test cases provided in CodeContests dataset are all valid, they can be generated by the ground-truth CCFGs that are manually annotated by human labelers.

Regarding the evaluation of the correctness of generated CCFGs by LLMs, we can also measure the *generality* of a grammar using the ground-truth CCFGs. If a CCFG generated by our model only produces a very restricted subset of the entire set of valid test cases, then the validity score can be really high as it does not reflect the diversity of test cases covered by the grammar.

Obviously, the best way to evaluate the quality of the grammar is to test the language equivalence between the ground-truth CCFG and the generated CCFG. However, it is well-known that the language equivalence between two CFGs (thus two CCFGs as well) is theoretically undecidable (Hopcroft et al., 2007). As an alternative measure to evaluate how much the generated CCFG conforms to the ground-truth CCFG, we randomly sample 10 test cases from the ground-truth CCFG and calculate the number of test cases that can be recognized by the generated CCFG.

Table 1: Overall experimental results. '# Probs.' denotes the number of problems for which test cases exists or successfully generate CCFGs out of a total 453 problems. We use up to 10 sample test cases for each method. 'Problem-based validity' is defined as the ratio of problems whose test cases are all valid to the total number of problems evaluated. 'Test case-based validity' is simply the ratio of valid test cases to all test cases. Note that different numbers of problems are used to measure test case validity for each method to measure problem and test case validity. Also, effectiveness is measured from 442 problems whose incorrect solutions are available.

Category	**Method**	**Validity**			**Effectiveness**
		# Probs.	Problem-Based	Test Case-Based	
CodeContests	Public	451	100.00	100.00 (±0.00)	27.83 (±24.93)
	Private	392	100.00	100.00 (±0.00)	45.78 (±28.36)
	Generated	446	51.12	83.90 (±23.84)	42.38 (±28.06)
Test Case Gen.	Mutation	451	53.44	84.75 (±21.89)	30.65 (±28.84)
	Gemini	453	67.11	81.94 (±34.09)	33.05 (±27.26)
	ChatGPT	453	84.11	**92.93** (±22.66)	43.74 (±28.08)
CCFG-Based	Gemini-1	156	44.23	45.19 (±49.22)	9.10 (±24.14)
	Gemini-5	261	75.86	77.36 (±41.14)	25.77 (±35.00)
	ChatGPT-1	372	86.29	86.83 (±33.37)	39.33 (±36.34)
	ChatGPT-5	375	**92.27**	92.87 (±25.19)	43.67 (±36.93)
	CCFGT5$_1$	209	60.77	60.91 (±48.69)	17.50 (±32.35)
	CCFGT5$_{100}$	444	83.33	84.28 (±35.82)	**47.77** (±36.57)

Effectiveness The primary purpose of test cases in competitive programming is to distinguish between correct and incorrect algorithms. For a given problem p, let A_p be a set of all incorrect algorithms implying that for each $y \in A_p$, there always exists a valid test case x such that $\hat{y}(x) \neq y(x)$, where $\hat{y}(x)$ is the correct output for the test case x. Then, we define the *effectiveness* $E(x, A_p)$ of a test case x with respect to A_p as

$$E(x, A_p) := \frac{|\{y \in A_p \mid y(x) \neq \hat{y}(x)\}|}{|A_p|},$$

which is the ratio of incorrect algorithms in A_p that are *distinguishable* by the test case x to all incorrect algorithms A_p. We then expand this to define the effectiveness of a set X of test case with respect to A_p as

$$E(X, A_p) := \sum_{\substack{x \in X: \\ x \text{ is valid}}} \frac{E(x, A_p)}{|X|} \times 100\ (\%)$$

We determine the correct output $\hat{y}(x)$ for a test case x by executing up to 10 correct algorithms from the dataset and selecting the most frequently occurring output as the correct one. Additionally, for each problem p, we also sample at most 10 incorrect algorithms to create a set designated as A_p. Out of the 453 problems available in the dataset, we use 442 that contain at least one incorrect algorithm to validate the effectiveness of each set of test cases generated by different methods.

4.4 Analysis of Experimental Results

Note that we report the average performance calculated from the individual averages for each problem to ensure that our analysis remains unbiased by the varying number of test cases in each problem.

Overall experimental results Table 1 presents the statistics for test cases, generated by either baseline algorithms or CCFGs. Gemini-k and ChatGPT-k denote the result of CCFGs produced by LLMs employing CoT with k different examples.

CCFGT5$_{100}$ demonstrates the highest number of success rate in CCFG generation, suggesting that a larger training dataset enhances the ability of the model to generate well-formed CCFGs from input specifications—tasks at which other LLMs often fail. However, the relatively low problem-based and test case-based validity of CCFGT5$_{100}$, in comparison to ChatGPT-5, indicates that these input specifications are too complex, resulting in incorrect CCFGs and invalid test cases.

Note that models often fail to generate the correct CCFG for a problem; however, once the correct CCFG is produced, the resulting test cases are always valid. The highest problem-based validity of CCFG-based test case generation using ChatGPT-5 supports this observation, as problem-based validity only counts test cases of a problem as valid if all test cases of that problem are valid.

Despite the substantial number of invalid test cases, which fail to distinguish any incorrect answers according to our metric, the effectiveness of CCFGT5$_{100}$ surpasses all other test cases, including private test cases used on real-world competitive coding platforms. The high effectiveness of test cases generated by CCFGs stems from the ability to produce generate various types of test cases.

Consequently, we conclude that utilizing a CCFG-based approach not only eliminates the need for individual test case validation but also enhance the effectiveness of test cases.

Sentence accuracy The experimental results in Table 2 clearly demonstrate that the CCFGT5 model outperforms LLMs in the task of translating specifications into grammars. This trend persists when only the productions are considered. Such results were expected as CCFGT5 has more opportunities to learn about CCFG owing to its reliance on 700 training data. Conversely, LLMs depend on 1-shot or 5-shot learning. As a result, fine-tuning is imperative for extracting the syntax of valid problem inputs.

Table 2: Experimental results on the translation of NL input specification to CCFGs.

		Exact Match (%)		
Model	Method	CCFG	Cons.	Both
CCFGT5	Greedy	23.84	28.04	17.66
	Beam$_{100}$	**70.20**	**54.30**	**45.03**
Gemini	1-shot	4.86	34.44	2.21
	5-shot	18.32	41.28	10.82
ChatGPT	1-shot	31.13	29.48	11.04
	5-shot	45.92	51.43	26.05
CCFGT5	@1	23.84	28.04	17.66
	@10	79.70	73.28	62.69
	@100	81.90	78.15	65.78

On the contrary, the results of the constraint analysis are unexpected. Large language models almost catch up fine-tuned models even in the 5-shot scenario. It is suggested that this may be attributed to the explicit mention of constraints in parentheses in the specifications. LLM detects this effortlessly with just five examples and uses the strings enclosed in the parentheses as constraints. Based on this analysis, it is anticipated that utilizing rule-based symbolic matching is a more effective method of extracting constraints compared to relying on deep learning-based approaches.

Validity and generality The evaluation results presented in Table 3 demonstrate that ChatGPT-5 produces the most semantically correct grammars, achieving the highest validity scores among the models tested. Meanwhile, the CCFGT5 model with a beam size of 100 not only shows the best generality in grammar generation but also excels in terms of the overall number of CCFGs generated. This indicates that while ChatGPT-5 leads in semantic accuracy, CCFGT5 with a larger beam size offers a broader application scope and a higher output volume.

Table 3: Comparison of generated grammars by LLM-based models in terms of the 'scope' of languages accepted by the grammars.

Model	# CCFGs	Valid.	General.
Gemini-1	156	44.23	40.13
Gemini-5	261	75.86	65.94
ChatGPT-1	372	86.29	65.59
ChatGPT-5	375	**92.27**	79.76
CCFGT5$_{100}$	444	83.33	81.06
CCFGT5$_1$	**209**	60.77	**60.91**

5 Conclusions

We introduced the generation of test cases from problem description by extracting the input specification. We utilized the CCFG to generate the test cases to evaluate the problems correctness in a more efficient way. Our proposed CCFGT5 model shows the most effective way to analyze the correctness and the incorrectness of a program.

In our future work, we plan to implement a robust pseudo-labeling framework to improve the semantic precision of the generated CCFGs, Moreover, we plan to refine our string sampling algorithm by integrating weighted production rules from existing weighted CFGs. This improvement will enable the generation of even more refined and effective test cases, bolstering the utility of our approach in competitive programming and beyond.

6 Limitations

Due to its NP-completeness, our CCFG parsing algorithm exhibits an exponential time complexity with respect to the length of the input string. In comparison, the CYK-algorithm (Sakai, 1961) offers a polynomial-time solution for CFG parsing. It still remains open if it is possible to design a polynomial-time CCFG parsing algorithm.

One notable limitation of our study pertains to the size of the training dataset, which may hinder the model's ability to generalize effectively. The design and implementation of grammar-based pseudo-labeling techniques for model generalization represent a promising avenue for our future research endeavors.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668.

Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3test: Assertion-augmented automated test case generation. *CoRR*, abs/2302.10352.

Matteo Brunetto, Giovanni Denaro, Leonardo Mariani, and Mauro Pezzè. 2021. On introducing automatic test case generation in practice: A success story and lessons learned. *CoRR abs/2106.13736*.

Qiaochu Chen, Arko Banerjee, Çagatay Demiralp, Greg Durrett, and Isil Dillig. 2023. Data extraction via semantic regular expression synthesis. *CoRR*, abs/2305.10401.

Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Inf. Softw. Technol.*, 171:107468.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event*. OpenReview.net.

Christopher Hahn, Frederik Schmitt, Julia J. Tillman, Niklas Metzger, Julian Siber, and Bernd Finkbeiner. 2022. Formal specifications from natural language. *CoRR abs/2206.01962*.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021*, volume 161 of *Proceedings of Machine Learning Research*, pages 54–63. AUAI Press.

Richard M. Karp. 1972. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, pages 85–103. Plenum Press, New York.

Rohit J Kate, Yuk Wah Wong, and Raymond J Mooney. 2005. Learning to transform natural to formal languages. In *Proceedings of the 20th national conference on Artificial intelligence-Volume 3*, pages 1062–1068.

Tsz On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 14–26. IEEE.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022a. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal

Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022b. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *CoRR*, abs/2305.01210.

Shuming Ma, Shaohan Huang Li Dong, Dongdong Zhang, Alexandre Muzi, Saksham Singhal, Hany Hassan Awadalla, and Furu Wei Xia Song. 2021. Deltalm: Encoder-decoder pre-training for language generation and translation by augmenting pretrained multilingual encoders. *CoRR abs/2106.13736*.

Mitchell Olsthoorn. 2022. More effective test case generation with multiple tribes of ai. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 286–290.

Daniel J. Rosenkrantz and Richard Edwin Stearns. 1969. Properties of deterministic top down grammars. In *Proceedings of the 1st Annual ACM Symposium on Theory of Computing, May 5-7, 1969, Marina del Rey, CA, USA*, pages 165–180.

Baptiste Rozière, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. *CoRR*, abs/2102.07492.

Itiroo Sakai. 1961. Syntax in universal translation. In *Proceedings of the International Conference on Machine Translation and Applied Language Analysis*.

Alzahraa Salman. 2020. *Test Case Generation from Specifications Using Natural Language Processing*. Ph.D. thesis, KTH.

Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2022. Automatic generation of acceptance test cases from use case specifications: An nlp-based approach. *IEEE Trans. Software Eng. 48(2)*, pages 585–616.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, pages 8696–8708.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Xi Ye, Qiaochu Chen, Xinyu Wang, Isil Dillig, and Greg Durrett. 2020. Sketch-driven regular expression generation from natural language and examples. *Trans. Assoc. Comput. Linguistics 8*, pages 679–694.

A Prompts for Grammar Generation of Large Language Models

A.1 Prompt for translation into grammar: 1-shot

```
You are the best programmar in the world.
You will be asked to determine the grammar and
↪  the constraint of the given specification
↪  by following the general rules and the
↪  structure of the given five examples.
I will first give you the general rules and the
↪  examples of how the grammars and the
↪  constraints are constructed from the
↪  specifications along with the reasons.
After analyzing the general rules, examples and
↪  learning how to generate the grammar and
↪  the constraint from the specification,
I will ask you to generate the grammar and the
↪  constraint for the last specification by
↪  following the rules and the examples.

<Specification> "Constraints\n\n* 1 \\leq N
↪  \\leq 100\n* 1 \\leq A_i \\leq
↪  1000\n\nInput\n\nInput is given from
↪  Standard Input in the following
↪  format:\n\n\nN\nA_1 A_2 \\ldots A_N"
↪  </Specification>
<Reason> "The grammar begins with the start
↪  symbol <S>. Here, 'N' is used as a counter
↪  variable for the array elements, thus it is
↪  denoted as [N] in the grammar to reflect
↪  its role as a counter. If 'N' were not a
↪  counter, it would be represented simply as
↪  'N'. All variables that serve as counters
↪  are similarly denoted with brackets.
↪  Consequently, the grammar is initiated with
↪  '<S>->[N] <n> <T_N>'. The non-terminal
↪  <T_i> is a counter-driven rule for [N],
↪  where 'A_i' represents each array element
↪  separated by a space symbol <s>. The
↪  grammar constructs are laid out as
↪  '<S>->[N] <n> <T_N>', '<T_i>-><T_i-1> <s>
↪  A_i', and '<T_1>->A_1' to systematically
↪  parse the sequence of array elements
↪  following the count [N]." </Reason>
<Grammar> "<S>->[N] <n> <T_N>", "<T_i>-><T_i-1>
↪  <s> A_i", "<T_1>->A_1" </Grammar>
<Constraint> "1<=n<=100", "1<=A_i<=1000"
↪  </Constraint>
<Specification> {{specification}}
↪  </Specification>

Generate the <Grammar> and <Constraint> for the
↪  given last <Specification> by strictly
↪  following the general rules and the
↪  examples provided without changing its
↪  basic structure from the examples in a json
↪  format.
```

A.2 Prompt for translation into grammar: 5-shot

```
You are the best programmar in the world.
You will be asked to determine the grammar and
↪  the constraint of the given specification
↪  by following the general rules and the
↪  structure of the given five examples.
I will first give you the general rules and the
↪  examples of how the grammars and the
↪  constraints are constructed from the
↪  specifications along with the reasons.
After analyzing the general rules, examples and
↪  learning how to generate the grammar and
↪  the constraint from the specification,
I will ask you to generate the grammar and the
↪  constraint for the last specification by
↪  following the rules and the examples.

<Specification> Constraints\n\n* -1000 ≤ a, b
↪  ≤ 1000\n\nInput\n\nTwo integers a and b
↪  separated by a single space are given in a
↪  line. </Specification>
<Reason> "The grammar construction begins with
↪  the initial non-terminal <S>. It defines
↪  two variables, 'a' and 'b', which are
↪  separated by a space symbol denoted as <s>.
↪  The structure of the grammar is formulated
↪  as '<S>->a <s> b', representing the input
↪  format where 'a' and 'b' are two integers
↪  separated by a space." </Reason>
<Grammar>  "<S>->a <s> b". </Grammar>
<Constraint>  "-1000<=a<=1000",
↪  "-1000<=b<=1000". " </Constraint>
<Specification> "Constraints\n\n* 1 \\leq N
↪  \\leq 100\n* 1 \\leq A_i \\leq
↪  1000\n\nInput\n\nInput is given from
↪  Standard Input in the following
↪  format:\n\n\nN\nA_1 A_2 \\ldots A_N"
↪  </Specification>
<Reason> "The grammar begins with the start
↪  symbol <S>. Here, 'N' is used as a counter
↪  variable for the array elements, thus it is
↪  denoted as [N] in the grammar to reflect
↪  its role as a counter. If 'N' were not a
↪  counter, it would be represented simply as
↪  'N'. All variables that serve as counters
↪  are similarly denoted with brackets.
↪  Consequently, the grammar is initiated with
↪  '<S>->[N] <n> <T_N>'. The non-terminal
↪  <T_i> is a counter-driven rule for [N],
↪  where 'A_i' represents each array element
↪  separated by a space symbol <s>. The
↪  grammar constructs are laid out as
↪  '<S>->[N] <n> <T_N>', '<T_i>-><T_i-1> <s>
↪  A_i', and '<T_1>->A_1' to systematically
↪  parse the sequence of array elements
↪  following the count [N]." </Reason>
<Grammar> "<S>->[N] <n> <T_N>", "<T_i>-><T_i-1>
↪  <s> A_i", "<T_1>->A_1" </Grammar>
<Constraint> "1<=n<=100", "1<=A_i<=1000"
↪  </Constraint>
```

```
<Specification> "Ania has a large integer S.
↪  Its decimal representation has length n and
↪  doesn't contain any leading zeroes. Ania is
↪  allowed to change at most k digits of S.
↪  She wants to do it in such a way that S
↪  still won't contain any leading zeroes and
↪  it'll be minimal possible. What integer
↪  will Ania finish with?\n\nInput\n\nThe
↪  first line contains two integers n and k (1
↪  \u2264 n \u2264 200 000, 0 \u2264 k \u2264
↪  n) \u2014 the number of digits in the
↪  decimal representation of S and the maximum
↪  allowed number of changed digits.\n\nThe
↪  second line contains the integer S. It's
↪  guaranteed that S has exactly n digits and
↪  doesn't contain any leading zeroes."
↪  </Specification>
<Reason> "The grammar begins with the starting
↪  non-terminal <S>. It inlcudes two variables
↪  n and k, but n  serves as the counter
↪  variable so we enclose in [n]. The grammar
↪  is initiated with <S>->[n] <s> k <n> <T_n>.
↪  Since it does not have any leading zeroes
↪  so, <T_i>->[1-9]{1} <B_i-1> is used reflect
↪  this and <B_i> is used to reflect all the
↪  other numbers including zero." </Reason>
<Grammar> "<S>->[n] <s> k <n> <T_n>",
↪  "<T_i>->[1-9]{1} <B_i-1>",
↪  "<T_1>->[1-9]{1}", "<B_i>->[0-9]{1}
↪  <B_i-1>", "<B_1>->[0-9]{1}" </Grammar>
<Constraint> "1<=n<=200000", "0<=k<=n"
↪  </Constraint>
<Specification> "Input\n\nThe first line
↪  contains a single integer n (1 ≤ n ≤ 10^6)
↪  − the length of Dima's sequence.\n\nThe
↪  second line contains string of length n,
↪  consisting of characters \"(\" and \")\"
↪  only." </Specification>
<Reason> "The grammar begins with the starting
↪  non-terminal <S>. It includes one variable,
↪  n, which is used both as a counter and to
↪  specify the length of a sequence.
↪  Consequently, the variable is represented
↪  as [n] to indicate its role as a length
↪  specifier for the sequence of characters,
↪  which consist only of '(' and ')'. This
↪  sequence is described using the regular
↪  expression [()]{n}, meaning a string of n
↪  characters, each of which is either '(' or
↪  ')'. Thus, the grammar is constructed to
↪  reflect this format: "<S>->[n] <n>
↪  [()]{n}"."</Reason>
<Grammar> "<S>->[n] <n> [()]{n}  has the
↪  counter variable [n] that is why  we
↪  changed n to [n] for reflecting the counter
↪  variable because it is used  as the length
↪  in the regex expression" </Grammar>
<Constraint> "1<=n<=10^6" </Constraint>
<Specification> "Input\n\nThe first line of the
↪  input contains two integers n and m (1 ≤ n,
↪  m ≤ 100) − the number of floors in the
↪  house and the number of flats on each floor
↪  respectively.\n\nNext n lines describe the
↪  floors from top to bottom and contain 2·m
↪  characters each. If the i-th window of the
↪  given floor has lights on, then the i-th
↪  character of this line is '1', otherwise it
↪  is '0'." </Specification>
```

```
<Reason> "The formation of the grammar starts
↪  with the start non-terminal <S>, there is
↪  two variables that is n and m and both of
↪  them serves as the counter variables, hence
↪  the variables are converted to [n] and [m]
↪  which then makes <T_n> that is separated by
↪  the new line <n>.  If variable n and m was
↪  not the counter variable we will write as n
↪  and m only not [n] and [m]. We convert all
↪  the variables that serves as a counter
↪  variable also to "[variable]".  Hence,  the
↪  grammar starts with "<S>->[n] <n> [m] <n>
↪  <T_N>". The <T_i>  is the counter
↪  non-terminal of the counter variable [n]
↪  having  <L_2m> counter non-terminal of size
↪  2m which is separated by the new line token
↪  <n>.  The <L_i> has only 0 and 1 of length
↪  one using the regular expression as
↪  [01]{1}, so we represent the length in {}"
↪  </Reason>
<Grammar> "<S>->[n] <s> [m] <n> <T_n>",
↪  "<T_i>-><T_i-1> <n>
↪  <L_2m>","<T_1>-><L_2m>", "<L_i>-><L_i-1>
↪  <s> [01]{1}", "<L_1>->[01]{1}"  </Grammar>
<Constraint> "1<=n,m<=100" </Constraint>

Generate the <Grammar> and <Constraint> for the
↪  given last <Specification> by strictly
↪  following the general rules and the
↪  examples provided without changing its
↪  basic structure from the examples in a json
↪  format.
```

A.3 Prompt for test case generation

```
Generate 10 valid test cases for the following
↪  specification:
{{specification}}
Each test cases should be in one line using "\n"
```