Structured Chain-of-Thought Prompting Enhances Code Generation with Large Language Models

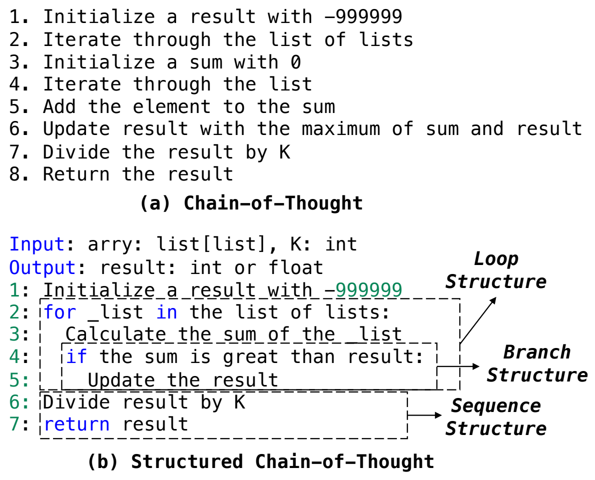
Anonymous ACL submission

Abstract

Chain-of-Thought (CoT) prompting with Large Language Models (LLMs) has shown impressive abilities in code generation. However, the accuracy of CoT prompting still can not satisfy practical applications. For example, gpt-3.5-turbo with CoT prompting only achieves 53.29% Pass@1 in HumanEval. This paper proposes the Structured Chain-of-Thought (SCoT) and presents SCoT prompting. Our motivation is source code contains rich structural information. Intuitively, structured intermediate reasoning steps make for structured source code. SCoT prompting teaches LLMs to generate a SCoT and then output the code. A SCoT is a series of intermediate reasoning steps built with program structures. By explicitly generating program structures, LLMs' programming abilities are further unlocked, *i.e.,* learning to think about how to solve requirements using the programming logic. We apply SCoT prompting to two LLMs (*i.e.,* OpenAI gpt-3.5-turbo and code-davinci-002) and evaluate it on three benchmarks (*i.e.,* HumanEval, MBPP, and MBCPP). SCoT prompting outperforms CoT prompting by up to 13.79% in Pass@1. SCoT prompting is robust to examples and achieves substantial improvements. The human evaluation also shows human developers prefer programs from SCoT prompting.

1 Introduction

Large Language Models (LLMs) have recently shown impressive abilities in code generation, such as OpenAI gpt-3.5-turbo (OpenAI, 2023). The performance of LLMs heavily relies on the prompt (Zhao et al., 2021). Chain-of-Thought (CoT) prompting (Wei et al., 2022) is the state-of-the-art (SOTA) prompting technique. By prompting some demonstration examples, CoT prompting teaches LLMs to generate a CoT and then output the code. A CoT is several intermediate natural language reasoning steps that describe how to write the code.



```
1. Initialize a result with -999999
2. Iterate through the list of lists
3. Initialize a sum with 0
4. Iterate through the list
5. Add the element to the sum
6. Update result with the maximum of sum and result
7. Divide the result by K
8. Return the result
```

(a) Chain-of-Thought

```
Input: arry: list[list], K: int
Output: result: int or float
1: Initialize a result with -999999
2: for _list in the list of lists:
3:   Calculate the sum of the _list
4:   if the sum is great than result:
5:     Update the result
6: Divide result by K
7: return result
```

(b) Structured Chain-of-Thought

Figure 1: The comparison of a Chain-of-Thought and our Structured Chain-of-Thought for a requirement.

Figure 1 (a) shows a CoT on code generation. However, CoT prompting brings limited improvements in code generation. For example, it only improves gpt-3.5-turbo by 0.82 points in Pass@1 upon a real-world benchmark (Chen et al., 2021).

The source code contains rich structural information (Zhang et al., 2019; Wang et al., 2020; Peng et al., 2021), *e.g.,* sequence, branch, and loop structures. Intuitively, intermediate reasoning steps leading to the structured code should also be structured. In other words, using program structures to build intermediate reasoning steps facilitates programming. This phenomenon has been discovered in fields such as programming education and is also known as *programming thinking* (Eckerdal et al., 2005). However, a CoT only linearly organizes intermediate steps and ignores the importance of program structures.

To alleviate the above knowledge gap, we propose a Structured CoT (SCoT) for code generation. A SCoT is a series of intermediate reasoning steps built with program structures. Figure 1 (b) shows an example of SCoT. Compared to the CoT, our SCoT has two advantages. (1) By explicitly generating program structures, LLMs' programming abilities are further unlocked, *i.e.,* learning to think about how to solve requirements using the programming logic. (2) In code generation, a SCoT is more suitable to describe intermediate steps than a CoT. As shown in Figure 1 (b), the SCoT uses a loop structure to accurately describe an iteration in line 2. However, in the CoT, the scopes of two iterations in lines 2 and 4 are ambiguous. The above advantages further liberate the coding abilities of LLMs, significantly improving their accuracy in code generation.

Specifically, a SCoT consists of two parts. The first part is an Input-Output (IO) structure. By generating an IO structure, LLMs define the entry and exit of the code, which clarifies requirements and facilitates the following solving process. The second part is a rough problem-solving process. Because any code or algorithm can be composed of three basic structures, *i.e.,* sequence, branch, and loop structures (Böhm and Jacopini, 1966). We teach LLMs to generate the solving process based on three basic program structures. It ensures that our SCoT can show problem-solving processes for various programs. Because LLMs' training data contains lots of code data, we think they can generate the above program structures.

Based on the SCoT, we present SCoT prompting. By prompting several demonstration examples, it teaches LLMs to generate an SCoT and then implement the code. We apply SCoT prompting to two popular LLMs (*i.e.,* OpenAI gpt-3.5-turbo and code-devinci-002 (Chen et al., 2021)). We compare SCoT prompting to CoT prompting on three representative benchmarks (*i.e.,* HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and MBCPP (Athiwaratkun et al., 2023)). We use test cases to measure the correctness of generated programs and report the Pass@k ($k \in [1, 3, 5]$). In terms of Pass@1, SCoT prompting outperforms CoT prompting by up to 13.79% in HumanEval, 12.31% in MBPP, and 6.63% in MBCPP. The improvements are stable in different LLMs and programming languages. The human evaluation also shows that human developers prefer programs generated by SCoT prompting. We also discuss the robustness of SCoT prompting to demonstration examples. Results show that SCoT prompting does not depend on specific examples or writing styles.

Our contributions are as follows.

- We propose a Structured Chain-of-Thought (SCoT), which uses program structures to build intermediate reasoning steps toward the structured code.
- We propose SCoT prompting for code generation. It prompts large language models to generate an SCoT and then implement the code.
- Qualitative and quantitative experiments show the superiority of SCoT prompting. We also discuss the robustness of SCoT prompting.

2 Methodology

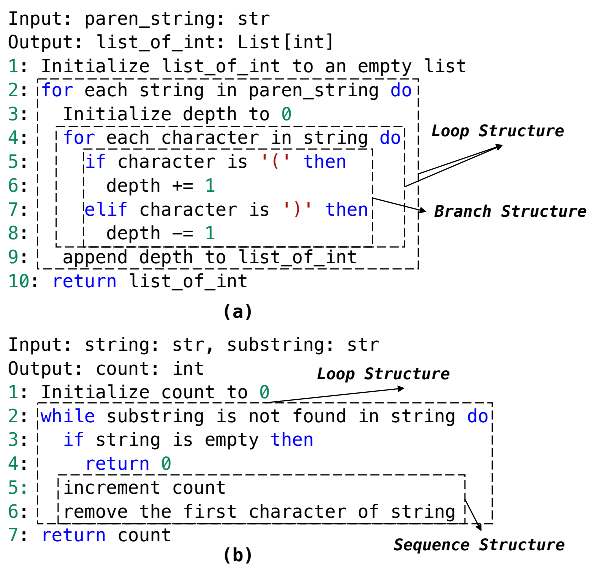
In this section, we describe the proposed Structured Chain-of-Thought (SCoT) and SCoT prompting.

2.1 Structured Chain-of-Thought

Chain-of-Thought (CoT) (Wei et al., 2022) is initially designed for natural language generation (*e.g.,* commonsense reasoning (Talmor et al., 2019)). A CoT consists of several intermediate natural language reasoning steps that sequentially describe how to solve a problem step by step. However, the CoT brings slight improvements in code generation. For example, CoT prompting only improves gpt-3.5-turbo by 0.82 points in Pass@1 on HumanEval.

In this paper, we propose a Structured CoT. Our motivation is that the goal of code generation is the highly structured code. The code solves a problem through special program structures, *e.g.,* sequence, branch, and loop structures. Imagine a human developer's thought process of solving a requirement - `reading text from a given file`. The developer tends to use program structures to come up with an initial idea, *e.g.,* `if the given file exists: read text from the file; else: raise an error;`. Program structures are beneficial to describe the problem-solving process and facilitate the following code implementation. Thus, our SCoT introduces program structures to build intermediate reasoning steps.

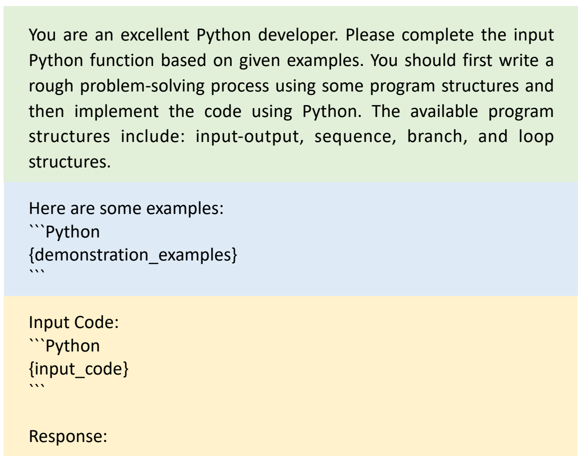
Figure 2 shows two examples of our SCoT. A SCoT consists of two parts. The first part is an Input-Output (IO) structure that shows input-output parameters and their types. An IO structure is required for a program, which indicates the entry and exit. Generating an IO structure is beneficial to clarify requirements and induce the following problem-solving process. The second part



```
Input: paren_string: str
Output: list_of_int: List[int]
Initialize list_of_int to an empty list
for each string in paren_string do
    Initialize depth to 0
    for each character in string do
        if character is '(' then
            depth += 1
        elif character is ')' then
            depth -= 1
    append depth to list_of_int
return list_of_int
```

```
Input: string: str, substring: str
Output: count: int
Initialize count to 0
while substring is not found in string do
    if string is empty then
        return 0
    increment count
    remove the first character of string
return count
```

Figure 2: Examples of SCoT in code generation.



You are an excellent Python developer. Please complete the input Python function based on given examples. You should first write a rough problem-solving process using some program structures and then implement the code using Python. The available program structures include: input-output, sequence, branch, and loop structures.

Here are some examples:
```Python
{demonstration_examples}
```

Input Code:
```Python
{input_code}
```

Response:

Figure 3: The prompt of SCoT prompting.

is a rough problem-solving process, which describes how to solve the requirement. Existing work (Böhm and Jacopini, 1966) proved that any program or algorithm can be composed of three basic structures, *i.e.,* sequence, branch, and loop structures. Thus, we use three basic program structures to build the problem-solving process. Theoretically, our SCoT is capable of describing a problem-solving process for any program. The details of the three basic structures are as follows.

- **Sequence Structure.** The intermediate steps are sequentially placed and all steps are at the same level.
- **Branch Structure.** It starts with a condition and places different intermediate steps for different results of the condition. In this paper, branch structures contain three formats, *i.e.,* `if _`, `if _ else _`, and `if _ elif _ else _`.
- **Loop Structure.** A set of intermediate steps is repeatedly conducted until given conditions are not met. In this paper, loop structures contain two basic formats, including the `for loop` and the `while loop`.

We allow the nesting between different program structures. It enables LLMs to design more complex SCoTs for some tricky requirements. As shown in Figure 2, LLMs flexibly use multiple program structures to generate complex SCoTs.

2.2 SCoT prompting

Based on the above analyses, we expect LLMs to generate a SCoT before outputting the code. A straightforward idea is the fine-tuning. However, it is costly to create a large set of high-quality SCoTs, which is much more complicated than requirement–code pairs used in normal code generation. Inspired by the progress achieved by prompting techniques, we propose SCoT prompting. By inputting a prompt, it teaches LLMs to generate a SCoT and then output the code.

The prompt of SCoT prompting is shown in Figure 3. The prompt starts with several natural language instructions, which tell the code generation task and available program structures. Then, we provide a few demonstration examples that consist of triples: <requirement, SCoT, code>. Finally, the prompt ends with a new requirement.

2.3 Implementation Details

We select a few (*e.g.,* three) <requirement, code> pairs from the training data of benchmarks as example seeds. Then, we manually write the SCoTs for seeds and obtain examples - <requirement, SCoT, code> triples, which are used to make prompts in Figure 3. The examples and prompt templates are available in our supplementary materials. In the future, users can flexibly apply SCoT prompting to more powerful LLMs in a plug-and-play fashion.

3 Experimental Setup

3.1 Benchmarks

Following previous studies (Chen et al., 2021; Nijkamp et al., 2023; Zheng et al., 2023; Chen et al.,

Table 1: Statistics of the datasets in our experiments.

Statistics	HumanEval	MBPP	MBCPP
Language	Python	Python	C++
# Train	–	474	413
# Test	164	500	435
Avg. tests per sample	7.7	3	3

2023), we conduct experiments on three public code generation benchmarks.

HumanEval (Chen et al., 2021) is a Python function-level benchmark, which contains 164 hand-written programming problems. **MBPP** (Austin et al., 2021) is a Python function-level benchmark. It contains 974 programming problems that involve numeric manipulations or standard libraries. **MBCPP** (Athiwaratkun et al., 2023) is a C++ function-level benchmark. It consists of 848 programming problems that are collected by crowd-sourcing. For all benchmarks, each programming problem consists of an English requirement, a function signature, and several test cases for checking generated programs.

The statistics of the benchmarks are shown in Table 1. We randomly pick several training samples to make examples in prompts (Section 2.3). Then, we evaluate SCoT prompting on the test data. Because HumanEval does not contain the train data, we reuse examples from MBPP in HumanEval.

3.2 Evaluation Metrics

Following previous code generation studies (Chen et al., 2021; Nijkamp et al., 2023; Zheng et al., 2023; Chen et al., 2023), we use Pass@k as our evaluation metrics. Specifically, we generate $n \geq k$ programs per requirement (in this paper, we use $n = 20$, $k \in [1, 3, 5]$). A requirement is solved if any generated programs pass the corresponding test cases. Then, we count the number of solved requirements c, and calculate the unbiased Pass@k:

$$\text{Pass@}k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{1}$$

We omit some text-similarity-based metrics (*e.g.,* BLEU (Papineni et al., 2002)). The reason is that these metrics are initially designed for natural language generation and are poor in measuring the correctness of programs (Chen et al., 2021).

3.3 Comparison Baselines

We select three mainstream prompting techniques on code generation as comparison baselines.

Zero-shot prompting (Chen et al., 2021) directly feeds the requirement into LLMs without examples. Then, it extracts a generated program from LLMs' outputs. **Few-shot prompting** (Chen et al., 2021) concatenates several randomly selected examples (*i.e.,* <requirement, code> pairs) and the input requirement together, making a prompt. Then, LLMs generate programs based on the prompt. **Chain-of-Thought (CoT) prompting** (Wei et al., 2022) is a variant of few-shot prompting. CoT prompting produces a special prompt consisting of <requirement, CoT, code> triples as examples.

To ensure the fairness of comparison, all baselines and SCoT prompting have the same number of examples (*i.e.,* three examples in our experiments) and example seeds.

We omit some emerging prompting techniques in other fields, such as Least-to-Most (Zhou et al., 2023). Because these approaches are designed for specific tasks (*e.g.,* Arithmetic reasoning) and can not be directly applied to code generation. We also omit some post-processing techniques for code generation (Chen et al., 2023; Zhang et al., 2023). They use LLMs to generate many candidates and then leverage test cases or neural networks to post-process candidates. We think our work and these post-processing techniques are complementary. Users can use our approach to generate programs and then use post-processing techniques to select the final output. We further validate the complementarity in Section 5.2.

3.4 Base Large Language Models

We pick two powerful LLMs on code generation as base models. **code-divinci-002** (Chen et al., 2021) is a powerful language model for code generation, which supports a commercial application - GitHub Copilot (GitHub, 2022). Codex's training data contains both natural language and billions of lines of code. We use OpenAI's APIs to access the latest version with 175 billion parameters. **gpt-3.5-turbo** (OpenAI, 2023) is trained with extensive natural language text and code files. Then, it is trained with reinforcement learning and learns to follow human instructions.

SCoT prompting does not rely on specific LLMs and can be applied to different LLMs in a plus-and-play fashion. In the future, we will apply it to more

Table 2: The Pass@k (%) of SCoT prompting and baselines on three code generation benchmarks. The numbers in red denote SCoT prompting's relative improvements compared to the SOTA baseline - CoT prompting.

Base Model	Prompting Technique	HumanEval			MBPP			MBCPP		
		Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
gpt-3.5-turbo	Zero-shot prompting	49.73	66.07	71.54	37.07	43.54	48.58	47.53	60.09	64.22
	Few-shot prompting	52.47	69.32	74.10	40.00	49.82	53.13	52.58	63.03	66.11
	CoT prompting	53.29	69.76	75.52	41.83	51.04	54.57	53.51	63.84	67.03
	SCoT prompting	**60.64**	**73.53**	**77.32**	**46.98**	**55.31**	**58.36**	**57.06**	**65.70**	**68.70**
Relative Improvement		**13.79%**	**5.40%**	**2.38%**	**12.31%**	**8.37%**	**6.95%**	**6.63%**	**2.91%**	**2.49%**
code-davinci-002	Zero-shot prompting	40.20	61.78	68.11	27.07	43.81	47.93	40.25	54.17	60.65
	Few-shot prompting	42.93	62.96	70.10	33.17	45.72	49.62	44.12	57.65	62.45
	CoT prompting	43.79	63.41	71.56	35.66	46.57	50.11	45.79	58.92	62.56
	SCoT prompting	**49.82**	**66.56**	**75.14**	**38.29**	**50.74**	**53.16**	**48.34**	**60.77**	**64.19**
Relative Improvement		**13.77%**	**4.97%**	**5.00%**	**7.38%**	**8.95%**	**6.09%**	**5.57%**	**3.14%**	**2.61%**

powerful LLMs.

3.5 Sampling Settings

Following previous studies (Chen et al., 2021; Zheng et al., 2023; Nijkamp et al., 2023), we use nucleus sampling (Holtzman et al., 2020) to decode programs from LLMs. All baselines and SCoT prompting generate 20 programs per requirement. The temperature is 0.8 and the top-p is 0.95. For zero-shot and few-shot prompting, the maximum generated length is 300 tokens. For CoT prompting and SCoT prompting, the maximum generated length is 600 tokens. The reason is that CoT prompting and SCoT prompting need to generate intermediate reasoning steps and code. Thus, they require a larger generation length.

4 Results and Analyses

4.1 Main Results

Results. Table 2 shows the Pass@k ($k \in [1, 3, 5]$) of different approaches on three benchmarks. These Pass@k values are reproduced in this paper under the same setting. We ensure that the relative improvements are valid.

Analyses. (1) SCoT prompting achieves the best results among all baselines. In terms of Pass@1, SCoT prompting outperforms CoT prompting by up to 13.79% in HumanEval, 12.31% in MBPP, and 6.63% in MBCPP. Pass@1 is a strict metric and it is difficult to improve. The significant improvements show that SCoT prompting is more promising than existing prompting techniques. (2) SCoT prompting is effective in different LLMs and programming languages. In terms of Pass@1, SCoT prompting improves gpt-3.5-turbo by up to 13.79% and code-davinci-002 by up to 13.77%. Besides, SCoT prompting is language-agnostic and brings substantial improvements in Python (*i.e.,* HumanEval and MBPP) and C++ (*i.e.,* MBCPP).

4.2 Robustness of SCoT prompting

As stated in Section 2.3, we select <requirement, code> pairs as example seeds and manually write SCoTs for them, obtaining examples in prompts. In practice, people may write different examples, which makes the performance of SCoT prompting vary. Thus, we explore the robustness of SCoT prompting to examples in two aspects, *i.e.,* seed selection and writing style.

Seed Selection (validate SCoT prompting does not rely on specific seeds). We select three groups of <requirement, code> pairs as seeds and ask an annotator to write SCoTs. Then, we obtain three groups of examples. We measure the performance of SCoT prompting with different groups of examples. **Writing Style** (validate SCoT prompting does not rely on specific writing styles). We hire three annotators to independently write SCoTs for the same example seed, and obtain three groups of examples. Annotator A is a Ph.D. student in computer science. Annotator B is a product manager from an IT company. Annotator C is a developer from an IT company. Then, we measure the performance of SCoT prompting with different examples.

For comparison, we also measure the robustness of CoT prompting in the same settings. We select gpt-3.5-turbo as the base model and conduct evaluations in HumanEval.

Results. The results are shown in Table 3 and 4, respectively.

Analyses. SCoT prompting substantially outperforms CoT prompting when using different example seeds or annotators. It validates that

Table 3: The Pass@k of CoT prompting and SCoT prompting with different example seeds.

Seed	CoT prompting			SCoT prompting		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Seed A	53.29	69.76	75.52	**60.64**	**73.53**	**77.32**
Seed B	52.81	68.97	74.55	**60.27**	**73.11**	**77.16**
Seed C	51.36	67.44	73.62	**59.36**	**72.88**	**76.79**

Table 4: The Pass@k of CoT prompting and SCoT prompting with different annotators.

Annotator	CoT prompting			SCoT prompting		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
Annotator A	53.29	69.76	75.52	**60.64**	**73.53**	**77.32**
Annotator B	51.43	67.92	73.44	**59.48**	**72.16**	**76.44**
Annotator C	52.18	68.45	74.71	**60.02**	**73.15**	**77.24**

SCoT prompting does not depend on specific seeds or writing styles. It also shows that the improvements of SCoT prompting attribute to program structures instead of specific details in examples.

We also notice that there are slight variances in the performance of SCoT prompting with different examples. It is expected for prompting techniques using examples. Similar variances can be found in CoT prompting, and SCoT prompting still outperforms CoT prompting with different examples.

4.3 Human Evaluation

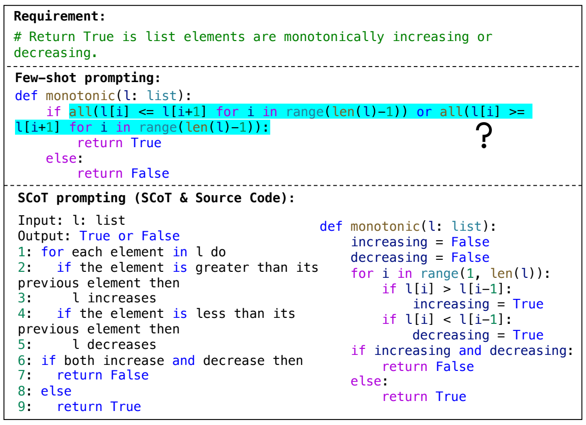
The goal of code generation is to assist developers in coding. Thus, we manually review the programs generated by SCoT prompting and baselines.

Setup. Following previous studies (Hao et al., 2022; Li et al., 2023), the evaluation metrics are three-fold. (1) *Correctness*: whether the program satisfies the requirement. 0 point: the program is inconsistent with the requirement. 1 point: the program is implemented, but misses some details. 2 points: the program is correctly implemented. (2) *Code Smell*: whether the program contains bad code smells. 0 point: There is a serious code smell. 1 point: some details are not in place. There is a code smell of low severity. 2 points: No obvious code smell. (3) *Maintainability*: whether the implementation is standardized and has good readability. 0 point: the program does not follow a consistent specification, *e.g.,* meaningless names. 1 point: the program implementation meets certain specifications, but can be further refined. 2 points: the program implementation is relatively standardized.

We select gpt-3.5-turbo as the base model and collect 200 generated programs per approach. Finally, we obtain 800 programs for evaluation. We hire 10 developers with 3-5 years of development

Table 5: The results of human evaluation in three aspects. The numbers in red denote SCoT prompting's relative improvements compared to CoT prompting. All the p-values are substantially smaller than 0.05.

Approach	Correctness	Code Smell	Maintainability
Zero-shot prompting	1.012	1.523	1.372
Few-shot prompting	1.119	1.653	1.552
CoT prompting	1.225	1.689	1.616
SCoT prompting	**1.412**	**1.869**	**1.873**
Relative Improvement	**15.27%**	**10.66%**	**15.90%**



```
Requirement:
# Return True is list elements are monotonically increasing or
decreasing.
```

```
Few-shot prompting:
def monotonic(l: list):
    if all(l[i] <= l[i+1] for i in range(len(l)-1)) or all(l[i] >=
l[i+1] for i in range(len(l)-1)):
        return True
    else:
        return False
```

```
SCoT prompting (SCoT & Source Code):
Input: l: list
Output: True or False
1: for each element in l do
2:   if the element is greater than its
previous element then
3:      l increases
4:   if the element is less than its
previous element then
5:      l decreases
6: if both increase and decrease then
7:   return False
8: else
9:   return True
```

```
def monotonic(l: list):
    increasing = False
    decreasing = False
    for i in range(1, len(l)):
        if l[i] > l[i-1]:
            increasing = True
        if l[i] < l[i-1]:
            decreasing = True
    if increasing and decreasing:
        return False
    else:
        return True
```

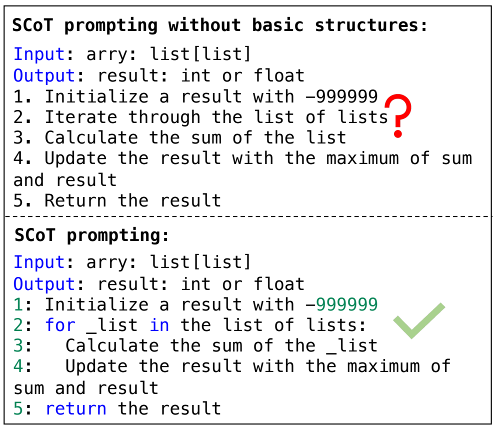
Figure 4: Two programs generated by few-shot prompting and SCoT prompting, respectively.

experience from a crowdsourcing platform to evaluate the programs in the form of a questionnaire. The evaluators include industry employees and academic researchers who are not co-authors of this paper. The 800 programs are divided into 5 groups, with each questionnaire containing one group. The programs are randomly shuffled and anonymously reviewed by evaluators. Each group is evaluated by two evaluators, and the final score is the average of two evaluators' scores. Evaluators are allowed to search the Internet for unfamiliar concepts.

Results. The results of the human evaluation are shown in Table 5.

Analyses. SCoT prompting performs better than baselines in all three aspects. Specifically, SCoT prompting outperforms the SOTA baseline - CoT prompting by 15.27% in correctness, 10.66% in code smell, and 15.90% in maintainability.

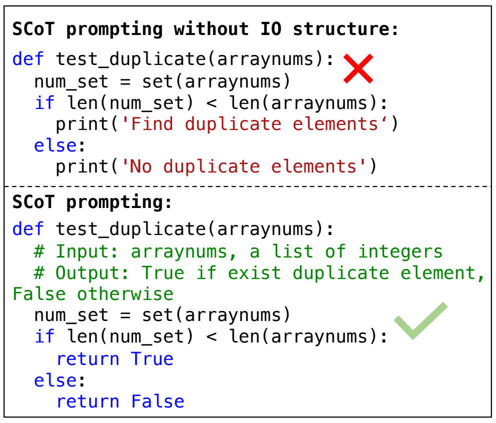
Figure 4 shows two programs generated by SCoT prompting and few-shot prompting, respectively. Both programs pass test cases. But the program from few-shot prompting contains a complex statement highlighted in Figure 4. Developers have to spend lots of effort to understand and maintain this program. In contrast, the program from SCoT prompting has good readability, and



```
SCoT prompting without basic structures:
Input: arry: list[list]
Output: result: int or float
1. Initialize a result with -999999
2. Iterate through the list of lists
3. Calculate the sum of the list
4. Update the result with the maximum of sum
and result
5. Return the result
```

```
SCoT prompting:
Input: arry: list[list]
Output: result: int or float
1: Initialize a result with -999999
2: for _list in the list of lists:
3:   Calculate the sum of the _list
4:   Update the result with the maximum of
sum and result
5: return the result
```

Figure 5: The comparison of SCoT prompting and SCoT prompting without basic structures.



```
SCoT prompting without IO structure:
def test_duplicate(arraynums):
  num_set = set(arraynums)
  if len(num_set) < len(arraynums):
    print('Find duplicate elements')
  else:
    print('No duplicate elements')
```

```
SCoT prompting:
def test_duplicate(arraynums):
  # Input: arraynums, a list of integers
  # Output: True if exist duplicate element,
False otherwise
  num_set = set(arraynums)
  if len(num_set) < len(arraynums):
    return True
  else:
    return False
```

Figure 6: The comparison of SCoT prompting and SCoT prompting without the IO structure.

the SCoT clearly explains the behavior of the code. Developers can further use the SCoT as comments of the program for future maintenance.

4.4 Ablation Study

SCoT prompting introduces three basic structures (*i.e.,* sequence, branch, and loop) and the IO structure. This section is to analyze the contributions of different program structures.

Setup. We select gpt-3.5-turbo as the base model. Then, we conduct an ablation study by independently removing basic structures and the IO structure. When removing basic structures, we use a CoT with an IO structure as the intermediate steps. When removing the IO structure, the SCoT only contains a solving process with basic structures.

Results. The results are shown in Table 6. "w/o" is the abbreviation of without.

Analyses. (1) Three basic structures are beneficial to generate a feasible solving process. After removing basic structures, the performance of SCoT prompting drops obviously. We carefully inspect failed cases and find that LLMs benefit from using basic structures to write a problem-solving process. Figure 5 shows the intermediate steps of SCoT prompting and SCoT prompting without basic structures. SCoT prompting without basic structures uses CoTs, which sequentially describe how to write the code line by line and contain many ambiguities. For example, the scopes of two iterations on lines 2 and 4 are unclear. LLMs are likely to misunderstand the CoT and generate incorrect code. In contrast, SCoT prompting uses three basic structures to describe the solving process. The SCoT is clear and is similar to code, benefiting the following code implementation.

(2) The IO structure benefits the requirement understanding. After deleting the IO structure, the performance of SCoT prompting has a slight decrease. We analyze failed cases and think the IO structure benefits the requirement understanding. Figure 6 shows two programs from SCoT prompting and SCoT prompting without the IO structure. We can see that SCoT prompting without the IO structure wrongly understands the output format and generates an incorrect program. After adding the IO structure, LLMs first reason about the input-output format and correctly return a boolean value.

5 Discussion

5.1 SCoT *vs.* Pseudocode

We notice that the SCoT is similar to the pseudocode. We randomly select 100 generated SCoTs and manually review them. 26% of SCoTs are close to the pseudocode. On one hand, we think the similarity enhances the usability of SCoT prompting. For example, users can quickly know the behavior of a program based on its SCoT. A SCoT also can be inserted into the comment and benefits future maintenance. On the other hand, the majority of SCoTs (74%) are different from the pseudocode because they are more abstract. SCoTs tend to use natural languages to summarize an operation, *e.g.,* `calaulate the sum of list1`. But the pseudocode contains more implementation details, *e.g.,* `sum ← 0; for i in list1: sum ← sum + i;`.

Compared to the pseudocode, we think the SCoT is a better choice for intermediate steps. Because a SCoT naturally decomposes code generation into two steps. LLMs first focus on exploring feasi-

Table 6: The results of ablation study.

Prompting Technique	HumanEval			MBPP			MBCPP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
CoT prompting	53.29	69.76	75.52	41.83	51.04	54.57	53.51	63.84	67.03
SCoT prompting	**60.64**	**73.53**	**77.32**	**46.98**	**55.31**	**58.36**	**57.06**	**65.70**	**68.70**
w/o Basic structures	55.67	70.94	76.13	43.36	53.64	56.57	54.79	64.32	67.77
w/o IO structure	59.65	72.79	77.12	46.13	54.76	57.88	56.61	65.01	68.42

Table 7: The comparison of SCoT-P prompting and SCoT prompting. The numbers in red denote SCoT prompting's relative improvements compared to SCoT-P prompting.

Approach	HumanEval			MBPP			MBCPP		
	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5	Pass@1	Pass@3	Pass@5
CoT prompting	53.29	69.76	75.52	41.83	51.04	54.57	53.51	63.84	67.03
SCoT-P prompting	55.23	70.33	75.94	43.28	52.16	55.77	54.25	64.09	67.78
SCoT prompting	**60.64**	**73.53**	**77.32**	**46.98**	**55.31**	**58.36**	**57.06**	**65.70**	**68.70**
Relative Improvement	**9.80%**	**4.55%**	**1.82%**	**8.55%**	**6.04%**	**4.64%**	**5.18%**	**2.51%**	**1.36%**

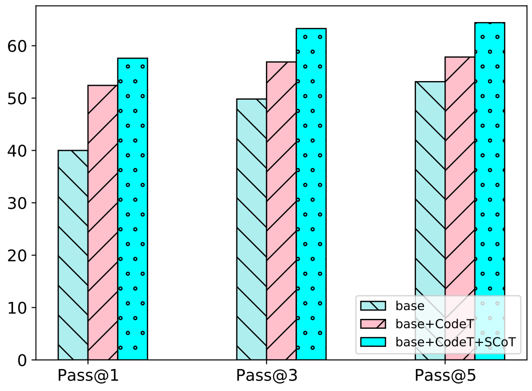


Figure 7: The complementarity between CodeT and SCoT prompting.

ble solutions and then implement the code in a standardized way. To validate this point, we design a variant of SCoT prompting, named SCoT-P prompting, which considers the pseudocode as intermediate steps. We apply SCoT-P prompting and SCoT prompting to gpt-3.5-turbo and measure their accuracy. The results are shown in Table 7. SCoT prompting substantially outperforms SCoT-P prompting on three benchmarks. The improvements show the superiority of our SCoT.

5.2 SCoT prompting *vs.* Post-Processing Techniques

Some recent studies (Chen et al., 2023; Zhang et al., 2023) propose post-processing techniques to enhance code generation. Given a requirement, they first sample many programs from LLMs and then use test cases or neural networks to post-process (*e.g.,* rerank or edit) programs. For example, CodeT (Chen et al., 2023) executes programs on auto-generated test cases. Based on execution results, the programs are reranked.

We do not directly compare SCoT prompting to post-processing techniques. Because SCoT prompting and post-processing techniques have different focuses, and they are complementary. Our work aims to design an effective prompt for LLMs. Post-processing techniques do not care about LLMs and aim to pick the best one from LLMs' multiple outputs. In practice, users can use SCoT prompting to generate many programs and then use post-processing techniques to get a final output.

To verify the complementarity between SCoT prompting and post-processing techniques, we conduct an exploratory experiment. We select gpt-3.5-turbo as a base model and progressively introduce CodeT and SCoT prompting. The results on MBPP are shown in Figure 7. We can see that the performance of the base model is continually improved by adding CodeT and SCoT prompting.

6 Conclusion and Future Work

Large Language Models (LLMs) have shown impressive abilities in code generation, but still can not satisfy practical applications. This paper proposes Structured Chain-of-Thought (SCoT) prompting. SCoT prompting teaches LLMs to generate intermediate reasoning steps using program structures (*e.g.,* sequence, branch, and loop structures) before outputting the code. Extensive experiments show that SCoT prompting significantly outperforms CoT prompting in Pass@k and human evaluation. Besides, SCoT prompting is robust to examples and obtains stable improvements.

In the future, we will explore new prompting techniques for code generation, *e.g.,* tree-based prompting.

7 Limitations and Risks

We identify a few limitations of this work. First, our approach is applicable to LLMs pre-trained with the source code, which may not always be included in all language models (especially small ones). Second, our experiments only cover two programming languages (*e.g.,* Python and C++) and two LLMs. We leave more experiments to future work.

Besides, the goal of our work is to improve the efficiency of human developers writing code, not to replace them. Meanwhile, we should also avoid code generation techniques being used to generate malware software.

References

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. Multi-lingual evaluation of code generation models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Anna Eckerdal, Michael Thuné, and Anders Berglund. 2005. What does it take to learn 'programming thinking'? In *International Computing Education Research Workshop 2005, ICER '05, Seattle, WA, USA, October 1-2, 2005*, pages 135–142. ACM.

GitHub. 2022. Github copilot. `https://github.com/features/copilot`.

Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. Aixbench: A code generation benchmark dataset. *CoRR*, abs/2206.13179.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. Skcoder: A sketch-based approach for automatic code generation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2124–2135. IEEE.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

OpenAI. 2023. gpt-3.5-turbo. `https://platform.openai.com/docs/models/gpt-3-5`.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL.

Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021. Integrating tree path in transformer for code representation. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 9343–9354.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. 2019. Commonsenseqa: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for*

Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), pages 4149–4158. Association for Computational Linguistics.

Wenhan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. 2020. Modular tree network for source code representation learning. *ACM Trans. Softw. Eng. Methodol.*, 29(4):31:1–31:23.

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 *NeurIPS*.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 783–794. IEEE / ACM.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 769–787. Association for Computational Linguistics.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 12697–12706. PMLR.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. 2023. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.