

LEGO-Compiler: Enhancing Neural Compilation Through Translation Composability

Shuoming Zhang^{1,3}, Jiacheng Zhao^{1,3}, Chunwei Xia², Zheng Wang², Yunji Chen^{1,3}, Xiaobing Feng^{1,3,4}, and Huimin Cui^{*1,3}

¹SKLP, Institute of Computing Technology, CAS
{zhangshuoming21s,zhaojiacheng,cyj,fxb,cuihm}@ict.ac.cn

²University of Leeds, UK
{C.Xia, Z.Wang5}@leeds.ac.uk

³University of Chinese Academy of Sciences, Beijing, China

⁴Zhongguancun Laboratory, Beijing, China

Abstract

Large language models (LLMs) have the potential to revolutionize how we design and implement compilers and code translation tools. However, existing LLMs struggle to handle long and complex programs. We introduce LEGO-Compiler, a novel neural compilation system that leverages LLMs to translate high-level languages into assembly code. Our approach centers on three key innovations: LEGO translation, which decomposes the input program into manageable blocks; breaking down the complex compilation process into smaller, simpler verifiable steps by organizing it as a verifiable LLM workflow by external tests; and a feedback mechanism for self-correction. Supported by formal proofs of translation composability, LEGO-Compiler demonstrates high accuracy on multiple datasets, including over 99% on ExeBench and 97.9% on industrial-grade AnsiBench. Additionally, LEGO-Compiler has also achieved near one order-of-magnitude improvement on compilable code size scalability. This work opens new avenues for applying LLMs to system-level tasks, complementing traditional compiler technologies.

1 Introduction

The rapid development of Large Language Models (LLMs) has led to an expansion of their applications and effectiveness across various domains [42, 37, 39, 64]. One important area where LLMs have shown impressive results is code translation, including tasks such as code generation from natural languages [61] and transformation between programming languages [60]. In code translation, LLMs have demonstrated remarkable accuracy and readability, often surpassing manually crafted translators.

While LLMs have shown promising results in translating between high-level programming languages [44, 45, 48] and in decompilation tasks [18, 6, 5], their application to translating from high-level languages to low-level assembly languages remains a relatively unexplored area, which is traditionally dominated by handcrafted compilers. However, compilers require significant engineering effort and are tailored to specific languages/architectures. It is interesting to explore whether LLMs can be used to automate the compilation process, and if so, to what extent they can achieve this. Although earlier investigations [3, 21] have shown low translation accuracy, recent work [63] has shown that LLMs finetuned with compiler-generated bilingual corpora can outperform advanced

*Corresponding Author

LLMs in C-x86 compilation tasks and can achieve up to 91% behavioral accuracy. However, an in-depth understanding of LLMs’ capabilities in this domain is still lacking. As compilation is typically divided into two main aspects: translation and optimization. This work focuses on exploring and answering about LLM capabilities in the translation aspect of compilation.

LLMs are pre-trained on vastly large code corpora. some are monolingual, and some may be bilingual (where LLMs can learn the translation rule between two languages). However, most of these LLMs do not disclose their training datasets, so their capabilities can only be assessed through empirical testing. We primarily find that current LLMs learn the neural compilation process from directly compiler-generated bilingual corpora, which is an intuitive way to construct a pretraining dataset and teach LLMs to compile. However, we also found that assembly code directly generated by compilers is hard for LLMs to learn due to several challenges. These include the presence of semantically opaque labels, symbols or numeric values that LLMs struggle to translate accurately, and the need to handle symbol renaming for identifiers with the same name in different scopes, etc. Although style migration or modifications to existing compilers can be made, these approaches still rely on an existing compiler to perform the neural compilation job, which doesn’t outperform existing designs.

Our work takes a different approach where we do not require bilingual corpora, as a result, we don’t necessarily rely on an existing compiler. **We guide LLMs to transform high-level code into assembly code step-by-step.** To achieve this, we first propose an adaptive compilation-knowledge guided LLM workflow, which involves a series of steps with verifications to ensure stepwise correctness, including control flow annotation, struct annotation, renaming transformation, variable mapping transformation and final assembly generation. By splitting the complex compilation task into smaller, manageable steps, we significantly reduce the task complexity in each step by forcing LLMs to focus on certain easier step and achieve substantial improvements in overall accuracy.

More importantly, the scalability of current code translation is an important and challenging problem. Although advanced LLMs already have hundreds of thousands tokens context limit, they can not merely compile a code with 2.6k tokens in CoreMark [19], which is just a 200-LOC function. The major challenges within this significant LLM failure are two-fold: **(1)** the complexity within each expression/statement, and **(2)** the complexity of program structures. **(1)** is continually improved with more advanced LLMs or with proper knowledge guidance and is not our research focus. However, **(2)** is more fundamentally challenging, as a program can be arbitrarily large and complex, and LLMs are not designed to handle such complexity, *direct LLM translation is just not scalable.*

To tackle this scalability problem, we propose **LEGO translation**, which draws inspiration from the modular and composable nature of LEGO blocks to divide-and-conquer it. This method breaks down large programs into manageable, semantically-composable control blocks, analogous to LEGO pieces. These blocks are then independently translated and rebuilt to form a full translation in much larger scale.

We combine the novel LEGO translation method with our proposed neural compilation workflow, and design **LEGO-Compiler**, a scalable, LLM-driven system that leverages the power of LLMs to perform scalable neural compilation tasks. LEGO-Compiler can correctly compile over 99% of the code in ExeBench [4], a large scale dataset with careful unit-testing. We can also correctly compile 97.9% of AnsiBench [33], a collection of well-known ANSI C standard benchmark suites, including CoreMark [19], an industrial-grade codebase that encompasses most common programming language features in C, where we compile all of its 40 functions correctly. Regarding scalability, we have verified that LEGO translation method can significantly scale up the capability of neural code translation performed by LLMs. By ablating LEGO-Compiler methods in AnsiBench evaluation and additional Csmith [59] evaluation, a random C code generator for compiler testing, LEGO translation scales up the available code size for neural compilation by near an order of magnitude.

The main contributions of this work are as follows:

- We propose the novel LEGO translation method to scale up the neural compilation task. By breaking down large programs into manageable, semantically-composable control blocks, the complexity of neural compilation tasks for LLMs is significantly reduced.
- We propose a novel verifiable step-by-step neural compilation workflow that guides LLMs to transform high-level code into low-level assembly. With breakdown steps, we characterize and evaluate the compilation process in LLM’s perspective, and achieve substantial improvements in behavioral accuracy compared to end-to-end translation.

- We provide both theoretical and empirical studies by formally defining the composability in code translation that underpins the LEGO translation method and empirically demonstrating LEGO-Compiler’s effectiveness through extensive evaluations. LEGO-Compiler achieve over 99% accuracy on ExeBench, 97.9% accuracy on AnsiBench. Ablation study also showcases that LEGO translation boosts the scalability of neural-compilable code size by an order of magnitude. The model-independent evaluation process also serves as an important benchmark for LLMs on complex system-level tasks.

2 Related Work

2.1 Code Translation

Recent neural-based **Code Translation** researches can be majorly categorized to two types: learning-based transpilers [44, 45, 56] and pre-trained language models [16, 54, 29, 43, 36, 1]. The former majorly studies the scarcity of parallel corpora [58] and develops unsupervised learning methods to overcome it. The latter using Large Language Models’ vast pretrained knowledge, can also perform code translations well without training [60, 25].

As for **compilation related translations**, [3, 21] preliminarily study on C-x86 and C-LLVM IR translation with limited investigations on the methods. There are also works on the reverse decompilation process [18, 6, 5] and works on code optimizations [9, 10]. The most related work is [63], which achieves state-of-the-art 91% Pass@1 accuracy on the C-x86 task using a finetuned CodeLlama model, where our work surpasses. Besides, their approach relies on compiler-generated bilingual corpora, while our methods can effectively eliminate such dependency by reasoning the steps of how a compiler works.

Finally, **Modular approach** is recognized as the key insight to scale up neural code generation/translation [34, 62, 51, 7]. Our work leverages similar idea of divide-and-conquer to breakdown a large long code into manageable control block parts, then LLMs can translate these parts separately with the aid of necessary context and combine their results into a large, complete and coherent translation.

2.2 Other Related Work

LLM self-repair. Recent research has focused extensively on enhancing LLMs’ self-correction capabilities. Several studies closely related to our work deserve mention. A comprehensive survey by [41] thoroughly examined methods for leveraging feedback to autonomously improve LLM outputs. [53] first uses compiler feedback for better code generation, and [15] establishes the syntax-runtime-functional bug type taxonomy and builds corresponding self-repair pipelines for code. Our work is their natural extensions to neural compilation scenario. While [35] investigated the limitations of self-repair mechanisms in code generation, our findings diverge significantly. Contrary to their conclusions, we discovered that self-repair serves as a highly effective solution in the neural compilation process, particularly when incorporating syntax feedback and runtime feedback.

In-context learning and Chain-of-Thoughts. LLMs are able to in-context learn via inference alone by providing few shots of demonstration then predicting on new inputs [31, 13]. Thus customized Chain-of-Thoughts [55, 8] can guide LLMs to perform complicated reasoning [50, 46], which is the cornerstone of our work. More specifically, [23] reveals the degradation of LLMs’ performance for long context, and validate the effectiveness on using Chain-of-Thoughts to mitigate. We found similar results in code translation/compilation tasks. However, our proposed **LEGO translations** method can significantly mitigate such degradation as it turns a long context direct translation into multiple composable, shorter ones that LLMs can handle.

Generation Scalability and Long Context Learning. Except for code translation, many LLM-based methods suffer scalability problems since larger inputs are not well trained like the smaller ones, which makes general methods to extend LLMs long-context capability challenging. For example, in order to coherently generate long passages of text, [49] proposes a multi-staged keyword-first progressive method to improve it significantly, where our work shares a similar insight. [24] introduces a self-route method to dynamically choose the usage of RAG or fully in-context, balancing the cost and performance in long-context scenario, which inspires us to use a similar dynamic approach.

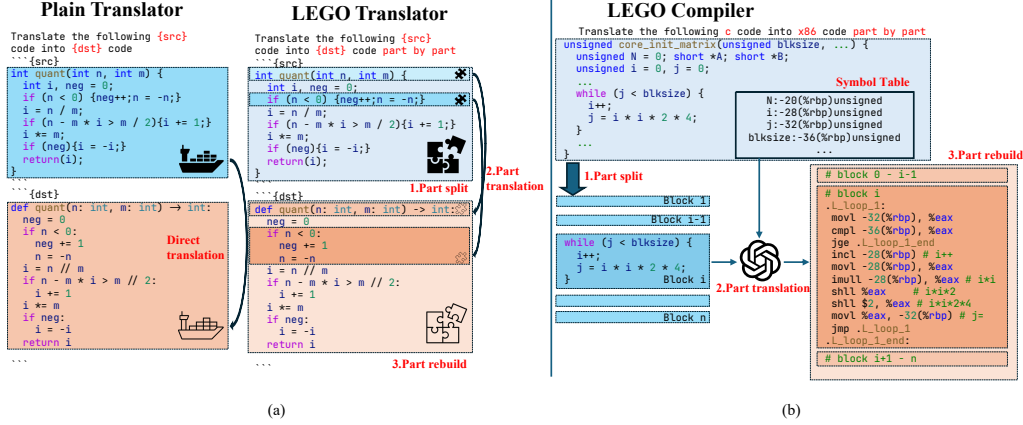


Figure 1: **a.** Plain translation vs LEGO translation, by splitting the program into smaller composable control blocks(parts), translating each part becomes an easier task, and rebuilding each translated partial result will form a full translation. **b.** LEGO compiler, a special case for LEGO translation, to translate each part correctly, a symbol table needs to be maintained first and provided during translation.

3 Methods

3.1 Problem Definition

Before introducing our method, we first define the neural compilation problem. Neural compilation can be viewed as a specialized version of code translation problem, as defined in Definition 1, with the goal of translating high-level programming language as the *src* language (such as C) into low-level assembly language as the *dst* language (such as x86, ARM, or RISC-V). Unlike general code translation, compilation needs to handle more low-level details, such as memory layout and calling convention, while ensuring the functional correctness of the translated result.

Definition 1. There are two programming languages: \mathcal{L}_{src} and \mathcal{L}_{dst} , each is an infinite set of valid program strings. There exists a unary relation \rightarrow from \mathcal{L}_{src} to \mathcal{L}_{dst} . The problem is to perform a translator function $T: \forall x \in \mathcal{L}_{src}, (\exists u \in \mathcal{L}_{dst}, x \rightarrow u) \rightarrow (x \rightarrow T(x)), T(x) \equiv x$ semantically.

3.2 LEGO Translation: Core Method

As depicted in (a) in Figure 1, previous neural code translation methods typically convert entire programs at the function or file level. While this approach may be effective for smaller programs, it struggles with larger programs due to significant accuracy degradation. These methods translate code at a coarse granularity, making it challenging to translate very long functions using LLMs. This limitation is stark: taking neural compilation as an example, even state-of-the-art LLMs [1, 38], despite possessing context windows potentially spanning hundreds of thousands of tokens (e.g., 128k-200k), demonstrably fail to correctly compile a C function exceeding just 2.6k tokens using direct translation. They could also perform code-snippet level translation, but they lack guidelines and necessary information to compose the code-snippet level results together, and there is also no clear formal proof of the composability of code. Despite these limitations, we observe an inherently composable nature in code. In the context of neural compilation, we propose the following insights to enhance translation scalability:

- **Fine-grained translation:** Instead of translating an entire program at once, focus on translating smaller code snippets accurately. By ensuring each part is correctly translated, they can be combined to form a semantically equivalent complete translation.
- **Contextual Awareness:** Effective translation of smaller code snippets requires understanding their contextual positioning within the code. This includes recognizing the relationship with preceding and succeeding snippets to maintain semantic coherence.

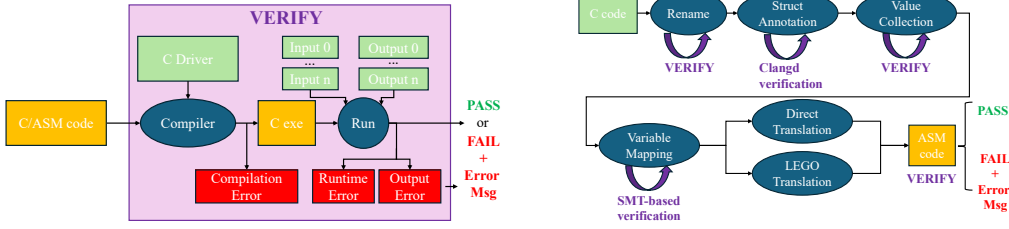


Figure 2: Neural compilation workflow in **LEGO-Compiler**. Left figure shows the behavioral verification process with unit-tests. Right figure shows the detailed steps in the workflow, some step is *residual* that may be skipped as performing such step may be unnecessary for certain input program.

- **Symbol Handling:** Accurate translation necessitates careful management of symbols (like variable scopes, types, and memory locations) and program constructs within each block to ensure correct mapping to the target architecture’s semantics and preserve functionality.

Inspired by [52], where this process is similar to the destruct and rebuild process of a LEGO toy, we named the fine-grained translation technique as **LEGO translation** and our system built upon it as **LEGO-Compiler**. As depicted in (b) in Figure 1, LEGO translation first breaks down large programs into manageable, self-contained blocks analogous to LEGO pieces (**Part split**). Then these blocks are independently translated (**Part translation**) and finally recombined, enabling scalable and accurate translation of complex programs (**Part rebuild**). All these methods rely on an inherent nature in programming languages, the composability in control block level, which reflects the linearization process in compiler design [57], where tree-structured control flow can be linearized, and therefore, composable. We have proved the widely applicable composability of programming languages using a constructive approach in Appendix A.

3.3 A Verifiable, Stepwise Neural Compilation Workflow

Directly translating complex high-level code to low-level assembly poses significant challenges for LLMs. To address this, we structure the neural compilation process as a **stepwise workflow**. This approach decomposes the overall task into a sequence of distinct, more manageable sub-tasks, each designed to be handled effectively by an LLM guided by specific prompts and context.

A key principle integrated throughout this workflow is **verifiability**. As depicted in Figure 2, by breaking down the complex, hard-to-verify compilation process into smaller steps, we create opportunities to validate the intermediate results of many stages before proceeding. This significantly enhances the reliability of the entire process and contributes to understanding the capabilities and limitations of LLMs in compilation tasks. Verification techniques employed vary depending on certain step and may include static source code analysis (e.g., comparing ASTs), cross-referencing intermediate calculations against compilation-based frontend tools, ensuring behavioral equivalence through execution testing, and potentially applying formal methods like SMT-based checks for specific properties like memory safety.

As depicted in Figure 2, the workflow proceeds through the following major steps:

- **Variable Renaming:** An initial source-level transformation ensures all variable identifiers within the compilation scope (e.g., a function) have unique names, resolving potential ambiguities from name shadowing and simplifying subsequent mapping. *This renaming step is verifiable* by executing the original and renamed source code on test cases to ensure behavioral equivalence.
- **Type and Layout Analysis:** This stage focuses on understanding the program’s data structures. The LLM performs a structured reasoning process for compound types (structs, unions, arrays) to determine their memory layout (size, alignment, member offsets) based on target architecture conventions and constituent basic types. *The correctness of this analysis is verifiable* by cross-referencing the inferred type sizes and offsets against outputs from standard development tools like Clangd [27] or IntelliSense [30].

- **Variable Mapping and Allocation:** This step identifies all variable instances and determines the correspondence between high-level variables and their low-level assembly representations. Global variables are mapped to labeled memory, while local variables are assigned stack offsets relative to a base pointer. Access to compound type elements uses calculated offsets adhering to conventions like the System V ABI [47]. *Verification for this stage can involve checks* using techniques like SMT-based verification [11] to detect potential memory allocation issues (e.g., overlaps, out-of-bounds accesses) based on the derived allocation plan.
- **Part Split (Control Flow Decomposition):** Leveraging the LEGO translation principle, this step decomposes the input function into smaller, manageable control blocks. It analyzes the program’s control flow graph (CFG) and uses an LLM-driven process to perform an adaptive algorithm in Algorithm 2 to decide where to split, aiming for semantically coherent units suitable for independent translation. *The structural integrity of the split is verifiable* by ensuring that the CFG formed by recombining the split blocks (before translation) is isomorphic to the original function’s CFG, or more simply, immediate recombination.
- **LEGO Part Translation:** Each control block generated by the Part Split step is translated independently by the LLM into assembly code. The LLM receives the source code for the block along with relevant context derived from previous steps, such as the established variable mappings and type layouts.
- **Part Rebuild and Final Verification:** The translated assembly blocks are reassembled according to the original control flow structure, typically, for two adjacent code blocks, the assembling operation is just concatenation as assembly language is linearized. *The functional correctness of the final, combined assembly code generated by the entire workflow (Split, Translate, Rebuild) is verified* through behavioral equivalence checks against the original source code, implemented using unit-tests.

Finally, LEGO-Compiler integrates a **self-correction loop with error feedback** as a final quality check after full translation. This mechanism detects residual errors using the assembler (semantic errors), runtime execution/debuggers (runtime errors), and behavioral testing via unit tests (behavioral errors). Diagnostic information is fed back to the LLM to iteratively refine the generated assembly. This self-correction process is crucial for enhancing the robustness and accuracy of the LLM-based compilation system. As LLMs are non-deterministic and may exhibit trivial errors, where the feedback loop is essential for correcting these errors.

4 Experiments

4.1 Experimental Setup

Major parameters we have tested are listed below: note that not all combinations of experimental settings are tested due to resource constraints.

- **Models:** We select a variety of state-of-the-art LLMs from different vendors, including OpenAI’s newest GPT-4.1 and its mini version [40], Anthropic’s Claude-3.5-sonnet [1] and Claude-3.7-sonnet [2], Deepseek’s Deepseek-V3 and its newest 0324 version [12], and Google’s Gemini-2.0-flash and Gemini-2.5-pro [20]. We select models pairwise to illustrate the model-side improvement on the neural compilation task.
- **Benchmarks:** We majorly test on ExeBench [4], a large-scale dataset of executable C programs, additionally we use AnsiBench [33] and Csmith-generated programs [59] as case studies. Technically, we use ExeBench’s Real-Executable subset, initially containing over 40k cases, after data cleaning and removing cases uncompileable by an oracle compiler, we finally obtain a 17,121 cases testset of ExeBench, which is too large for full scale evaluation. We further filter two subsets of ExeBench for evaluation, we filter a hard-cases subset of 1,996 samples based on the number of basic blocks and instructions within these blocks using the LLVM toolchain [28], shown in Figure 5, and the other is a randomly selected subset with 2000 cases, which is used for comparison and ablation studies.
- **Temperature:** 0.0-1.0, with 0.2 step increments
- **Architecture:** x86_64, arm-v8a, riscv64, majorly on x86

Table 1: ExeBench (17,121 cases) experimental results with method-level ablation on C-x86 neural compilation task compared to previous state-of-the-art [63], the base model performs similarly, but with our proposed methods, we achieve substantial accuracy improvement on the full dataset. Except DS-V3-0324 model, other models are evaluated on a subset of 2000 cases due to budget constraints, DS-V3-0324 is evaluated on both datasets and performs nearly identical (<0.2%).

Model	Direct	CoT workflow	LEGO translation
DS-V3-0324-full	91.589%	94.708%	99.375%
Zhang et.al [63]	91.718%	-	-
DS-V3-0324	91.60%	94.60%	99.20%
DS-V3-1226	88.7%	94.55%	98.30%
GPT-4.1	90.20%	95.10%	99.50%
GPT-4.1-mini	78.90%	91.30%	97.70%
Claude-3.7-sonnet	95.20%	97.75%	99.70%
Claude-3.5-sonnet	92.40%	95.25%	99.15%
Gemini-2.5-pro	94.45%	97.25%	99.65%
Gemini-2.0-flash	73.10%	84.60%	92.60%

Table 2: ExeBench Hard Subset (1996 cases) evaluation: The applied filters are based on nums of basic blocks(10), and max(80)/all(200) instructions within these blocks analyzed by LLVM toolchain [28], Detailed characterization of ExeBench and its hard subset is on Figure 5.

Model	Direct	CoT workflow	LEGO translation
GPT-4.1	87.43%	94.54%	98.70%
Claude-3.7-sonnet	92.59%	97.14%	99.20%
Deepseek-V3-0324	85.92%	91.38%	97.60%
Gemini-2.5-pro	94.29%	97.09%	99.10%

4.2 ExeBench Evaluation

Recall Figure 2, we evaluate ExeBench through the following setup:

1. Translate the C program to assembly (to generate hypothesis), where we have three different methods: direct translation(as baseline), stepwise workflow translation and LEGO translation. Note that some steps in the workflow is necessary for the LEGO translation method, as a global context is needed.
2. Assemble and link the hypothesis assembly to create an executable.
3. Run the executable through 10 different IO test cases provided by ExeBench.
4. Consider the translation *successful* if it passes all test cases.
5. If a translation fails (at any former step), apply self-fixing with the collected error feedback to the LLM, will try **k** rounds. We set **k** to 5 during the evaluation.
6. Consider the translation *failed* if it doesn’t pass after all configured attempts.

Table 1 and Table 2 summarize the empirical results of our LEGO-Compiler on ExeBench targeting x86-64 architecture. We establish carefully crafted 1-shot prompts to guide LLMs for each translation method. With our proposed methods, all models achieve substantial improvements, where newest models’ accuracy on the whole dataset/hard subset reaches averagely 99.56% and 98.65%. Claude-3.7-sonnet and Gemini-2.5-pro models are the best performing models during our evaluation, achieving over 99% accuracy on the hard subset. We analyze the ablated results as follows:

- **Step-by-step workflow** improves the translation majorly related to complex data structures and many variable assignments, where direct translation may fails to handle such complexity all together.

Table 3: ExeBench’s hard subset evaluations with different architectures with Gemini-2.5-pro.

Architecture	Model	Direct	CoT workflow	LEGO translation
x86_64	Gemini-2.5-pro	94.29%	97.09%	99.10%
arm64	Gemini-2.5-pro	87.64%	92.33%	96.74%
riscv64	Gemini-2.5-pro	84.22%	89.88%	94.64%

- **LEGO translation** majorly improves lengthy code translation with multiple control statements, where the model struggles to keep track of the context and the correct label usage without our divide-and-conquer methodology.
- **Self-correction** fixes most trivial errors related to architecture-specific knowledge, and improves at all methods as it is orthogonal. Two major types of observed errors are: 1) misuse of instruction operands, like ‘cmp’ instructions cannot compare two immediate values or two memory values; 2) mnemonics-related, like access global variables or values stored in data section, LLMs need to generated %rip relative addressing operands instead of direct label usage. Taking Deepseek-V3-0324 as an example, 1) and 2) account for 26 and 40 failed cases in its 172 failed cases during CoT workflow evaluation.

Except x86 evaluation, we also evaluate LEGO-Compiler on arm64 natively on Apple M1 chip and riscv64 through Spike simulator. Due to time and budget constraints, we only evaluate the hard subset as it contains more challenging cases. As depicted in Table 3, the evaluation results are similar to x86, where our LEGO-Compiler powered by Gemini-2.5-pro achieves similar improvements with our proposed methods. The globally lower accuracy may due to lower pretrained knowledge of these assembly languages in LLMs, or insufficient prompt engineering efforts as we have not thoroughly tested prompts usage on these architectures. It is also an interesting work to automate the prompt engineering process for different architectures to inject compilation-related knowledge to the translation process, which we leave for future work.

Another finding is observed through pairwise comparison of models, where we find clear improvement of newer/larger models over older/smaller models. We analyze the reasons as two-fold: First, more advanced new models are pretrained with more compilation-related knowledge, which helps the translation of certain expressions and statements. Second, newer models are more capable of reasoning, which is critical for the workflow translation and LEGO translation methods.

To sum up, the empirical results of our LEGO-Compiler system is promising, we prove a training-free approach to use LLMs as neural compilers, which can successfully translate averagely 99.56% of ExeBench testset and 98.65% of its hard subset across advanced LLMs from 4 state-of-the-art vendors. The model-independent evaluation process also establishes a challenging benchmark for LLMs, which requires 3 key capabilities: 1) mathematical reasoning and long-context reasoning 2) code/assembly understanding and translation, 3) error localization and correction.

4.3 AnsiBench: more real-world codebase evaluation

We conduct additional real-world codebases evaluation, we use AnsiBench [33], a collection of well-known ANSI C standard benchmark suites [22, 14, 19], benchmarking a wide variety of systems and compilers, including a number of classic, industry-standard benchmarks as well as some selected programs that can be used as benchmarks.

We evaluate the whole AnsiBench collection with our LEGO-Compiler, powered by Claude-3.7-Sonnet, our best-performing model in previous evaluation. We list the details of every function we compiled in Figure 3, totally we have 96 functions in total, except for few utility functions which are easy to compile, most of them represent real-world codebase complexity. We ablate the translation methods we applied to showcase both the effectiveness of CoT-like workflow and LEGO translation. In total, we pass 94 out of 96 cases in AnsiBench across 7 different codebases, including Whetstone, Dhrystone, Hint(one failure), Linpack, Tripforce(one failure), Stream and CoreMark. When measured by token count, LEGO translation method significantly improves the translation scalability of real-world code by near an order of magnitude as illustrated in Figure 3.

There are majorly three types of errors where the first two types are where LEGO translation is superior.

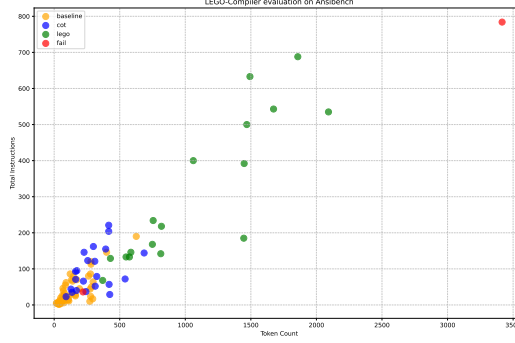


Figure 3: AnsiBench evaluation results with Claude-3.7-Sonnet. The **token count** only computes the input length of C code, and typically, the output assembly will be 3-6 times larger in token size.

- Lengthy code input with over a thousand token size (typically), where the output size is truncated due to limited model output length. Besides, the coarse-grained translation itself is prone to bugs. LEGO translation method can significantly reduce such errors, the case in which LEGO translation also fails is the main function of Hint benchmark, which is even more complex than the main function of CoreMark depicted in Figure 6. We analyze its failure, where the LLM-reasoning step of the stack allocation fails to generate a correct mapping. Despite this, LEGO translation handles all the other lengthy code correctly as it successfully reduces the translation complexity to control-block level.
- Long context forgetting problem [26], where the model can not match the current processing assembly with the source code faithfully, LEGO translation method, on the other hand, can handle these cases efficiently with less unnecessary contexts that may cause these ‘random’ errors. Besides, finer-grained translation also gives LLMs more attention to faithful translation of operations, the order of operations and implicit conversions.
- Insufficient pretraining in LLMs, where LLMs lack certain knowledge to perform certain expression/statement translation or other architecture-specific details. For example, the other error in AnsiBench, the `generate_password` function in TripForce, where the translation fails to translate the multiline strings correctly. Feedback correction can mitigate such failures. Besides, a clear model-level improvement is observed by all model pairs, and we can be positive about these failures because as LLMs advance with more pretrained knowledge, their performance will improve as well.

4.4 Csmith: randomly generated programs evaluation

Except for AnsiBench evaluation. We further perform evaluations on randomly generate programs with sufficient complexity. We use Csmith [59], a random generator of C programs which is widely used for finding compiler bugs using differential testing as the test oracle. Typically, Csmith examines compilers with random programs with corner case features and numbers, testing the robustness of compilers. Code examples generated from Csmith are illustrated in Figure 7.

We follow similar ablation strategy in AnsiBench evaluation. As depicted in Figure 8, randomly generated programs by Csmith are very hard for both baseline and CoT-only methods to translate. In a randomly generated test suite of 40 cases generated by Csmith, LEGO translation successfully compiles 25 cases, while baseline translation can only compiles 4 cases, and 13 cases for CoT workflow. Besides, the complexity of cases passed by LEGO translation method are significantly larger than others, characterized by token count, basic block count and total instructions, LEGO translation scales in code size and complexity by near an order of magnitude.

During Csmith evaluation, we also identify several kinds of errors during LEGO-Compiler translation. For example, overflow value assignment is an error which doesn’t occur usually but can be found in compiler testing. Taking `int16_t x = 0x56671485;` as an example, it will trigger errors because LLMs directly generate `movw $0x56671485, x’s` address in x86, which fails to check whether the numerical value (overflows the 16 bit word) can be represented through `movw` instruction.

Another example is, when handling with implicit type conversions, LLMs may not cast the type correctly, this is critical for floating point computation as operations with wrong precision will cause accumulated numerical errors. As a result, LEGO-Compiler in Csmith evaluation only achieves moderate behavioral accuracy of 62.5%.

However, Csmith-generated test cases do not commonly appear in real-world usages. Therefore, LEGO-Compiler is still promising in compiling common programs as evaluated by ExeBench and AnsiBench evaluation, indicating great potentials in the field of neural compilations.

5 Conclusion

We have presented LEGO-Compiler, a novel approach to neural compilation that leverages Large Language Models (LLMs) to translate high-level programming languages into assembly code. Our LEGO translation method breaks down large programs into manageable, self-contained blocks through the composable nature of code, significantly extending the scalability of neural code translation. By incorporating a series of Chain-of-Thought stages guided by classical compiler design and self-correction mechanisms, LEGO-Compiler effectively addresses key challenges in compilation tasks, achieving significant improvements in accuracy and scalability, with both theoretical and empirical studies demonstrating the effectiveness of our approach.

These findings provide important insights into the capabilities and limitations of LLMs in neural compilation tasks. As LLM capabilities continue to improve, approaches like LEGO-Compiler are poised to play an increasingly important role in the future of software development and compilation, complementing and enhancing traditional compiler technologies.

References

- [1] Anthropic. Claude ai. <https://www.anthropic.com>, 2023. Accessed: 2024-09-14.
- [2] Anthropic. Claude 3.7 sonnet and claude code. <https://www.anthropic.com/news/claude-3-7-sonnet>, February 2025. Accessed: 2025-05-04.
- [3] Jordi Armengol-Estapé and Michael FP O’Boyle. Learning c to x86 translation: An experiment in neural compilation. *arXiv preprint arXiv:2108.07639*, 2021.
- [4] Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael FP O’Boyle. Exebench: an ml-scale dataset of executable c functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 50–59, 2022.
- [5] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O’Boyle. Slade: A portable small language model decompiler for optimized assembler. *arXiv preprint arXiv:2305.12520*, 2023.
- [6] Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. Boosting neural networks to decompile optimized binaries. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 508–518, 2022.
- [7] Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.
- [8] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. Navigate through enigmatic labyrinth a survey of chain of thought reasoning: Advances, frontiers and future, 2024.
- [9] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

- [10] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization, 2024.
- [11] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [12] DeepSeek-AI. Deepseek-v3 technical report, 2024.
- [13] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. A survey on in-context learning, 2024.
- [14] Jack J Dongarra, Piotr Luszczek, and Antoine Petit. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.
- [15] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, Yan Liu, Enyu Zhou, Ming Zhang, Yuhao Zhou, Yueming Wu, Rui Zheng, Ming Wen, Rongxiang Weng, Jingang Wang, Xunliang Cai, Tao Gui, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. What’s wrong with your code generated by large language models? an extensive study, 2024.
- [16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.
- [17] Free Software Foundation. *c++filt*. GNU Binutils, 2023. Accessed: [Insert access date here].
- [18] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32, 2019.
- [19] Shay Gal-On and Markus Levy. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.
- [20] Gemini Team. Gemini: A family of highly capable multimodal models, 2024.
- [21] Zifan Carl Guo and William S. Moses. Enabling transformers to understand low-level programs. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2022.
- [22] John L Gustafson and Quinn O Snell. Hint: A new way to measure computer performance. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 2, pages 392–401. IEEE, 1995.
- [23] Mosh Levy, Alon Jacoby, and Yoav Goldberg. Same task, more tokens: the impact of input length on the reasoning performance of large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15339–15353, Bangkok, Thailand, August 2024. Association for Computational Linguistics.
- [24] Zhuowan Li, Cheng Li, Mingyang Zhang, Qiaozhu Mei, and Michael Bendersky. Retrieval augmented generation or long-context llms? a comprehensive study and hybrid approach, 2024.
- [25] J. Liu, F. Zhang, X. Zhang, Z. Yu, L. Wang, Y. Zhang, and B. Guo. hmcodetrans: Human-machine interactive code translation. *IEEE Transactions on Software Engineering*, 50(05):1163–1181, may 2024.
- [26] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.
- [27] LLVM Project. Clangd: C/c++ language server. <https://clangd.llvm.org/>, 2024. Accessed: 2024-09-14.

- [28] LLVM Project. *The LLVM Compiler Infrastructure*. LLVM Foundation, 2024. Version 18.1.8.
- [29] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.
- [30] Microsoft Corporation. Intellisense in visual studio code. <https://code.visualstudio.com/docs/editor/intellisense>, 2024. Accessed: 2024-09-14.
- [31] Sewon Min, Xinxu Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11048–11064, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.
- [32] Christian Munley, Aaron Jarmusch, and Sunita Chandrasekaran. Llm4vv: Developing llm-driven testsuite for compiler validation. *Future Generation Computer Systems*, 2024.
- [33] nfnit. Ansibench: A selection of ansi c benchmarks and programs useful as benchmarks, 2024. Accessed: 2024-11-22.
- [34] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE ’15*, page 585–596. IEEE Press, 2015.
- [35] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*, 2024.
- [36] OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haoming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O’Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov,

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

- [37] OpenAI. Chatgpt: Optimizing language models for dialogue. *OpenAI*, 2023.
- [38] OpenAI. Gpt-4o system card. <https://openai.com/index/gpt-4o-system-card/>, August 2024. Accessed on September 15, 2024.
- [39] OpenAI. Video generation models as world simulators. <https://openai.com/index/video-generation-models-as-world-simulators/>, 2024. Technical Report.
- [40] OpenAI. Introducing GPT-4.1 in the API. <https://openai.com/index/gpt-4-1/>, April 2025. Accessed: 2025-05-04.
- [41] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies. *Transactions of the Association for Computational Linguistics*, 12:484–506, 2024.
- [42] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.
- [43] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2022.
- [44] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.
- [45] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021.
- [46] Mingyang Song, Mao Zheng, and Xuan Luo. Can many-shot in-context learning help long-context llm judges? see more, judge better!, 2024.
- [47] System V ABI. System v application binary interface: Amd64 architecture processor supplement (with lp64 and ilp32 programming models) version 1.0. Technical report, The Santa Cruz Operation, Inc., 2018.
- [48] Marc Szafraniec, Baptiste Rozière, Hugh Leather, Patrick Labatut, François Charton, and Gabriel Synnaeve. Code translation with compiler representations. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

- [49] Bowen Tan, Zichao Yang, Maruan Al-Shedivat, Eric Xing, and Zhiting Hu. Progressive generation of long text with pretrained language models. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4313–4324, Online, June 2021. Association for Computational Linguistics.
- [50] Boshi Wang, Xiang Deng, and Huan Sun. Iteratively prompt pre-trained language models for chain of thought. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2714–2730, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.
- [51] Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation, 2024.
- [52] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. LEGO-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations*, 2024.
- [53] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Findings of the Association for Computational Linguistics: ACL 2022*, pages 9–19, Dublin, Ireland, May 2022. Association for Computational Linguistics.
- [54] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.
- [55] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022.
- [56] Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, et al. Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*, pages 23685–23700. PMLR, 2022.
- [57] Niklaus Wirth, Niklaus Wirth, Niklaus Wirth, Suisse Informaticien, and Niklaus Wirth. *Compiler construction*, volume 1. Addison-Wesley Reading, 1996.
- [58] Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn Rose. Data augmentation for code translation with comparable corpora and multiple references. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13725–13739, Singapore, December 2023. Association for Computational Linguistics.
- [59] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’11*, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.
- [60] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. Exploring and unleashing the power of large language models in automated code translation. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024.
- [61] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet NL2Code: A survey. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada, July 2023. Association for Computational Linguistics.

- [62] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.
- [63] Shuoming Zhang, Jiacheng Zhao, Chunwei Xia, Zheng Wang, Yunji Chen, and Huimin Cui. Introducing compiler semantics into large language models as programming language translators: A case study of C to x86 assembly. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 996–1011, Miami, Florida, USA, November 2024. Association for Computational Linguistics.
- [64] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Measuring github copilot’s impact on productivity. *Commun. ACM*, 67(3):54–63, February 2024.

A Composability of C-like Language Constructs

A.1 Definitions and Language Structure

We define a simplified C-like language structure using the following EBNF-inspired grammar:

```

block: '{' (blockItem)* '}';
blockItem: decl | stmt;
stmt:
    lVal '=' exp ';'           # assignStmt
    | exp ';'                  # exprStmt
    | 'goto' label ';'         # gotoStmt
    | ';'                      # blankStmt
    | block                   # blockStmt
    | IF '(' exp ')' stmt (ELSE stmt)? # ifStmt
    | WHILE '(' exp ')' stmt   # whileStmt
    | FOR '(' stmt exp ';' stmt ')' stmt # forStmt
    | SWITCH '(' stmt ')' stmt # switchStmt
    | BREAK ';'               # breakStmt
    | CONTINUE ';'            # continueStmt
    | RETURN (exp)? ';'       # returnStmt;

```

We derived from the grammar that describes C-like language to form the following definitions. Also for simplicity purposes, we omit the slight differences between **decl**, **stmt** and **exp**.

Definition 2 (Basic Statement). *A basic statement is a statement that does not contain any other statements within its structure. This includes assignStmt, exprStmt, gotoStmt, blankStmt, breakStmt, continueStmt, and returnStmt. We first exclude gotoStmt for the main proof for simplicity.*

Definition 3 (Basic Block). *A basic block is a sequence of consecutive basic statements as defined in Definition 2, in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.*

Definition 4 (Control Block). *A control block is a code snippet that reflects a complete control structure, such as for(;;){}, if(){}[else{}], while(){} , do{}while(), or switch(){case:...}. Each subpart of a control block can be other control blocks or basic blocks as defined in Definition 3.*

Definition 5. *A basic control block is an innermost control block (Definition 4) where each of its subparts contains only basic blocks as defined in Definition 3.*

Definition 6 (Compound Control Block). *A compound control block is a control block (Definition 4) that contains at least one subpart that is not a basic block (Definition 3), but rather another control block as defined in Definition 4.*

Definition 7 (Translation Function and Valid Translations). *Let \mathcal{T} be the set of all valid translation functions from SRC to DST, where SRC is the source language (our C-like language) and DST is the destination language (e.g., x86 assembly).*

Formally, $\mathcal{T} = \{T \mid T : SRC \rightarrow DST\}$ such that for any $T \in \mathcal{T}$ and any $stmt \in SRC$:

1. $T(stmt) \in DST$ 2. $T(stmt)$ preserves the semantics of $stmt$

A translation function $T \in \mathcal{T}$ maps each construct in the source language to one or more constructs in the destination language while preserving the program's behavior.

Definition 8 (Translation Composability). *Let (SRC, \circ) be the source language with concatenation operation \circ , and (DST, \cdot) be the destination language with concatenation operation \cdot . Let \mathcal{T} be the set of valid translation functions as defined in Definition 7.*

Translation composability holds if and only if:

$$\exists T \in \mathcal{T} : \forall P_1, P_2 \in SRC, T(P_1 \circ P_2) \equiv T(P_1) \cdot T(P_2)$$

Where:

- $T : SRC \rightarrow DST$ is a translation function

- \equiv denotes semantic equivalence, preserving both control flow and data flow
- $\circ : SRC \times SRC \rightarrow SRC$ is the concatenation operation in the source language
- $\cdot : DST \times DST \rightarrow DST$ is the concatenation operation in the destination language

A.2 Composability of Basic Statements

Theorem 1 (Composability of Basic Statements). *For any two basic statements $stmt_1$ and $stmt_2$ in SRC , as defined in Definition 2, their translation is composable: $T(stmt_1 \circ stmt_2) \equiv T(stmt_1) \cdot T(stmt_2)$*

Proof. We prove this for all combinations of assignment statements and expression statements. The proof considers control flow preservation, data flow preservation, and independence of translation. Other basic statements (blank, return, etc.) trivially maintain composability as they do not affect control or data flow when composed with other basic statements. \square

Example A.1. *This example illustrates the composability of basic statements as defined in Definition 2 and proved in Theorem 1.*

Consider the following sequence of basic statements:

```
a = b + 3; // stmt_1
b = a - 1; // stmt_2
```

The translation of these statements might look like:

```
T(stmt_1):
    mov eax, [b]
    add eax, 3
    mov [a], eax

T(stmt_2):
    mov eax, [a]
    sub eax, 1
    mov [b], eax
```

These translations are composable because:

1. *Control Flow:* The order of execution is preserved ($stmt_1$ then $stmt_2$).
2. *Data Flow:* The value of 'a' computed in $stmt_1$ is correctly used in $stmt_2$.
3. *Independence:* The translation of $stmt_2$ does not depend on how $stmt_1$ was translated, only on its effect (the value of 'a').

Therefore, $T(stmt_1 \circ stmt_2) \equiv T(stmt_1) \cdot T(stmt_2)$, demonstrating composability.

Example A.1 illustrates that even when statements have data dependencies, their translations remain composable as long as the order of operations is preserved. Similar proof of composability can be made for all stmts within a basic block (Definition 3).

A.3 Composability of Basic Control Structures

Theorem 2 (Composability of Basic Control Structures). *Basic control structures (if-else, for, while, do-while, switch-case), where all their components are basic blocks as defined in Definition 3, are composable under the translation function T as defined in Definition 7.*

Proof. We will prove this for each basic control structure:

1. For Loop:

Let B_{init} , B_{cond} , B_{incr} , and B_{body} be the basic blocks for init, cond, incr, and body respectively.

Translation structure:

```

T(basic_for_loop):
    T(B_init)
loop_start:
    T(B_cond)
    jz loop_end
    T(B_body)
    T(B_incr)
    jmp loop_start
loop_end:

```

1. Control Flow Preservation: The structure of jump instructions preserves the original control flow.
2. Data Flow Preservation: The order of operations within and between blocks is maintained.
3. Composability: $T(\text{basic_for_loop}) \equiv T(B_{init}) \cdot T(B_{cond}) \cdot T(B_{body}) \cdot T(B_{incr})$, where \cdot represents concatenation with appropriate jump instructions.

Therefore, the basic for loop is composable under T . Similar proofs can be constructed for other basic control structures. \square

2. If-Else Statement: Let B_{cond} , B_{then} , and B_{else} be the basic blocks for condition, then-branch, and else-branch respectively.

Translation structure:

```

T(basic_if_else):
    T(B_cond)
    jz else_label
    T(B_then)
    jmp end_label
else_label:
    T(B_else)
end_label:

```

Control flow and data flow preservation follow similarly to the for loop case.

3. While Loop: Let B_{cond} and B_{body} be the basic blocks for condition and body respectively.

Translation structure:

```

T(basic_while):
loop_start:
    T(B_cond)
    jz loop_end
    T(B_body)
    jmp loop_start
loop_end:

```

4. Do-While Loop: Let B_{body} and B_{cond} be the basic blocks for body and condition respectively.

Translation structure:

```

T(basic_do_while):
loop_start:
    T(B_body)
    T(B_cond)
    jnz loop_start

```

5. Switch-Case Statement: Let B_{expr} be the basic block for the switch expression, and B_1, B_2, \dots, B_n be the basic blocks for each case.

Translation structure:

```

T(basic_switch):
    T(B_expr)

```

```

    cmp result, case1_value
    je case1_label
    cmp result, case2_value
    je case2_label
    ...
    jmp default_label
case1_label:
    T(B_1)
    // No break implies fall-through
case2_label:
    T(B_2)
    ...
default_label:
    T(B_n)
end_switch:

```

For all these structures, control flow is preserved by the appropriate use of jump instructions, and data flow is maintained by the sequential execution of basic blocks. The translation of each structure is a composition of its basic block translations, proving composability.

Theorem 3 (Composability of Break and Continue Statements). *Break and continue statements, which are basic statements as per Definition 2, are composable within their respective control structures when proper loop depth tracking is maintained.*

Proof. Let $loop_depth$ be a counter maintained during translation to track nested loop levels.

1. Break Statement: Translation structure:

```

T(break):
    jmp loop_end_label_depth

```

Where $loop_end_label_depth$ corresponds to the end of the current loop at depth $loop_depth$.

2. Continue Statement: Translation structure:

```

T(continue):
    jmp loop_continue_label_depth

```

Where $loop_continue_label_depth$ corresponds to the continuation point of the current loop at depth $loop_depth$.

Control flow is preserved by jumping to the appropriate label based on the current loop depth. Data flow is trivially preserved as these statements do not modify data.

The composability of these statements within their containing loops is maintained because: a) They generate a single jump instruction that integrates with the loop's control flow. b) The loop depth tracking ensures the jump targets the correct loop level in nested structures. \square

A.4 Composability of Complex Structures

Definition 9 (Composable Control Block). *A composable control block is either:*

- A basic block as defined in Definition 3, or
- A basic control structure as proved in Theorem 2, or
- A sequence of composable control blocks, or
- A control structure whose all subparts are composable control blocks.

Theorem 4 (Composability of Sequential Control Blocks). *A sequence of composable control blocks CB_1, CB_2, \dots, CB_n as defined in Definition 9 is composable under the translation function T .*

Algorithm 1 Iterative Bottom-Up Composability Proof Algorithm

```
procedure PROVECOMPOSABILITY(Program  $P$ )
   $blocks \leftarrow \text{DecomposeIntoOutermostControlBlocks}(P)$  ▷ Initial decomposition
   $to\_process \leftarrow \text{new Deque}()$ 
  for each  $block$  in  $blocks$  do
     $to\_process.\text{PushBack}(block)$  ▷ Initialize processing queue
  end for
  while  $to\_process$  is not empty do
     $current\_block \leftarrow to\_process.\text{PopFront}()$  ▷ Handle first unhandled block
    if  $\text{IsBasicBlock}(current\_block)$  then
      continue ▷ Do nothing
    else if  $\text{IsControlStructure}(current\_block)$  then
       $sub\_blocks \leftarrow \text{SplitControlStructure}(current\_block)$ 
      for each  $sub\_block$  in  $sub\_blocks$  in reverse order do
         $to\_process.\text{PushFront}(sub\_block)$  ▷ Handle sub-blocks in original order
      end for
    else
      return  $P$  is not composable ▷ Unrecognized structure
    end if
  end while
  return  $P$  is composable
end procedure

function SPLITCONTROLSTRUCTURE(Block  $b$ )
  if  $b$  is a For Loop then
    return  $\text{SplitForLoop}(b)$ 
  else if  $b$  is an If-Else structure then
    return  $\text{SplitIfElse}(b)$ 
  else
    return  $\text{SplitOtherControlStructure}(b)$  ▷ Extensible for other structures
  end if
end function

function SPLITFORLOOP(ForLoop  $f$ ) ▷ Decompose for loop into constituent parts
  return [  $f.\text{init}$ ,  $f.\text{ForBodyLabel}$ ,  $f.\text{cmp}$ ,  $\text{ConditionalJump}(f.\text{ForEndLabel})$ ,
   $f.\text{body}$ ,  $f.\text{incr}$ ,  $\text{UnconditionalJump}(f.\text{ForBodyLabel})$ ,  $f.\text{ForEndLabel}$  ]
end function

function SPLITIFELSE(IfElse  $i$ ) ▷ Decompose if-else into constituent parts
  return [  $i.\text{cmp}$ ,  $\text{ConditionalJump}(i.\text{ElseLabel})$ ,  $i.\text{then\_body}$ ,
   $\text{UnconditionalJump}(i.\text{EndIfLabel})$ ,  $i.\text{ElseLabel}$ ,  $i.\text{else\_body}$ ,  $i.\text{EndIfLabel}$  ]
end function
```

Proof. Let CB_1, CB_2, \dots, CB_n be composable control blocks. 1. By Definition 9, each CB_i is composable. 2. Translation structure: $T(CB_1 \circ CB_2 \circ \dots \circ CB_n) \equiv T(CB_1) \cdot T(CB_2) \cdot \dots \cdot T(CB_n)$ where \circ denotes sequential composition in SRC and \cdot denotes concatenation in DST. 3. Control Flow Preservation: The sequential order of control blocks is maintained in the translation. 4. Data Flow Preservation: The order of operations between control blocks is preserved.

Therefore, the sequence of composable control blocks is itself a composable control block under T . □

Theorem 5 (Composability of Arbitrary Programs). *Any program P that can be decomposed into a sequence of control blocks as defined in Definition 4 is composable under the translation function T if the Iterative Composability Proof algorithm (Figure 1) marks it as composable.*

Proof. The proof follows from the correctness of the Iterative Composability Proof algorithm:

1. The algorithm starts with basic blocks and basic control structures, which are proven composable by Theorem 1 and Theorem 2.
2. It iteratively builds up composability for larger structures:
 - Sequences of composable blocks are proved composable by Theorem 4.
 - Control structures with all composable subparts are marked composable.
3. The process continues until the entire program is marked composable or no further progress can be made.
4. If the entire program is marked composable, it means that $T(P)$ can be expressed as a composition of the translations of its composable parts, preserving both control flow and data flow as per Definition 8.

Therefore, if the algorithm returns that P is composable, then P is indeed composable under the translation function T . \square

Theorem 6 (Composability of Goto Statements). *Goto statements, which are basic statements as per Definition 2, are composable under the translation function T , but arbitrary goto statements can break the structured control flow assumed in the main proof.*

Proof. Let l be a label and $\text{goto } l$ be a goto statement.

Translation structure:

```
T(goto l):
    jmp label_l
```

```
T(l:):
label_l:
```

The goto statement translates to an unconditional jump, preserving control flow. It doesn't directly affect data flow. Composability holds as $T(\text{stmt}_1 \circ \text{goto } l \circ \text{stmt}_2) \equiv T(\text{stmt}_1) \cdot T(\text{goto } l) \cdot T(\text{stmt}_2)$.

However, goto introduces complications:

- Non-local control flow can break the nested structure of control blocks.
- Programs with unrestricted goto usage are difficult to decompose into well-defined control blocks.
- It can lead to unstructured code, complicating reasoning about program behavior.

\square

While goto is provably composable, it's discouraged in modern programming for readability, maintainability, and optimization reasons. Our composability principle is most applicable and valuable in the context of structured programming paradigms.

A.5 Scope and Limitations of the Proof

The proof of composability presented in this paper is based on a simplified model of C-like languages and unoptimized translation. It's important to note several key points about the scope and limitations of this proof:

1. **Simplification and Correctness:** The simplifications made in our language model and translation process do not compromise the validity of the proof. The core of our argument relies on the decomposition of programs into control blocks and the composability of these blocks. The internal structure of basic blocks, while important for actual compilation, does not affect the composability principle we've established.

2. **Unoptimized Translation:** Our proof assumes a straightforward, unoptimized translation process. This assumption is crucial for maintaining the direct correspondence between source code structures and their translations.
3. **Limitations for Complex Language Features:** The composability principle as proved here can be applied to C-like languages, but may not hold for more complex language features. For example:
 - **Exception Handling:** Languages with sophisticated exception handling mechanisms, such as Python, introduce complexities that can break composability. These mechanisms often require:
 - Guarded execution of code blocks.
 - Runtime type information (RTTI) for determining appropriate exception handlers.
 - Non-local control flow that can't be easily decomposed into our model of control blocks.
 - **Coroutines and Generators:** Features that allow for suspending and resuming execution mid-function can introduce state that is not easily captured in our model of control flow.
 - **Reflection and Metaprogramming:** Languages that allow for runtime modification of program structure or behavior can invalidate static composability assumptions.

Although not applicable to some specific language features, it doesn't mean the composability and its derived LEGO translation method is not applicable to the whole programming language, as long as these features are not used in the code, the composability will still stand and the LEGO translation will still work.

4. **Optimizations Across Basic Blocks:** Our proof assumes that the boundaries of control blocks are respected in the translation process. However, many real-world compiler optimizations operate across these boundaries. Examples include:
 - Loop unrolling
 - Function inlining
 - Global value numbering
 - Code motion optimizations

Such optimizations can reorder, eliminate, or combine operations from different control blocks, potentially breaking the composability property as we've defined it. However, some local optimizations, like mem2reg and strength reduction can still be performed, and is observed through our evaluation where the model tends to perform such optimizations.

5. **Applicability:** Despite these limitations, the composability principle proved here is valuable for:
 - The foundation of LEGO translation method, the proof reveals the composable nature of code in at least control block level, which is a major difference than natural languages.
 - The proof process also guided Algorithm 2 in LEGO translation, as proving the composability and making use of the composability share similar algorithms.

In conclusion, while our proof provides a strong foundation for understanding composability in C-like languages with straightforward translation, it's important to recognize its boundaries. More complex language features may require extensions or modifications to this framework to maintain composability guarantees. And optimized code translation usually is not composable.

B Discussions

B.1 Universality of LEGO translation

The LEGO translation method, while initially developed for compilation tasks, demonstrates broader applicability based on fundamental properties of programming languages rather than being specific to compilation. The composability that LEGO translation leverages stems from the well-encapsulated control flow and locality principles inherent in modern programming languages (disregarding constructs like **goto** in C, more limitations are clearly described in Appendix D).

These characteristics are intrinsic to programming languages themselves and have guided modern compiler design. They enable the modular partitioning of large-scale programs in modern software development, allowing for incremental and even parallel compilation of code. We harness these properties and apply them to the context of neural compilation using Large Language Models (LLMs).

It’s important to note that the applicability of LEGO translation extends beyond compilation. It is suitable for various tasks originating from programming languages, such as code translation between different languages. This method significantly enhances the scalability of machine translation tasks for code, providing a powerful tool for handling large and complex codebases.

B.2 Managing Highly Complex Expressions

One of the primary challenges in neural compilation arises when dealing with expressions or statements of high complexity. In such cases, LLMs struggle to accurately evaluate these expressions through next token prediction. To address this, we propose two solutions:

- **External Tool Integration:** We can utilize external parsing tools to generate tree structure information for complex expressions evaluation. This tree structure is then provided to the LLM, offering an explicit traversal order and guiding the evaluation process.
- **Expression Decomposition:** Without relying on external tools, we can design a new pass where the LLM identifies high-complexity expressions and rewrites them as a combination of lower-complexity expressions. This approach ensures that the entire program consists only of expressions within a proper LLM’s evaluation capabilities.

B.3 Computational Cost, Effectiveness, and Future Prospects

While our neural compilation method is primarily a proof of concept, it does incur significantly higher computational costs compared to traditional compilation methods - approximately 10^6 to 10^7 times higher. However, this should be weighed against the substantial human resources required for traditional compiler development.

The key advantage of our approach lies in its potential for rapid adaptation to new instruction set extensions or frontend intrinsics. Through techniques like RAG (Retrieval-Augmented Generation) and in-context learning, our method can be extended to support new architectures or language features. This positions neural compilation as a valuable assistant in the compiler development process. A particularly promising application is in generating end-to-end unit tests for compiler adaptation to new instructions. This could significantly streamline the development and testing phases of compiler updates. Recent research like [32] has shown the ability to use LLMs to generate unit tests during compiler validations.

C Evaluation Details

This section provides more details figures, tables and further explanations about **LEGO-Compiler** design and experiment evaluation.

C.1 LEGO-Compiler: detailed designs

As depicted in Figure 4, LEGO-Compiler is designed to perform a series of steps guided by compiler expert knowledge, just like Chain-of-Thoughts(CoTs). However, not all CoTs are necessary for each input code, so in our design, we have an **analyze-then-think** approach. First, we will perform an analyzing pass to scan the whole program, whose output flags would trigger necessary Chain-of-Thoughts that will be used in the following process. In this example, the code pattern is majorly about double-precision floating point calculations (**numerical**) and complicated expression evaluation (**order**), besides, the code is too long for direct translation method to handle (**long**). Thus, based on the analysis, we applied the following CoTs:

- **Values collection:** A necessary thought, collecting all variables, numerals in a scanning pass, the **numerical** flag will teach the LLM about assembly knowledge to save numerical values.

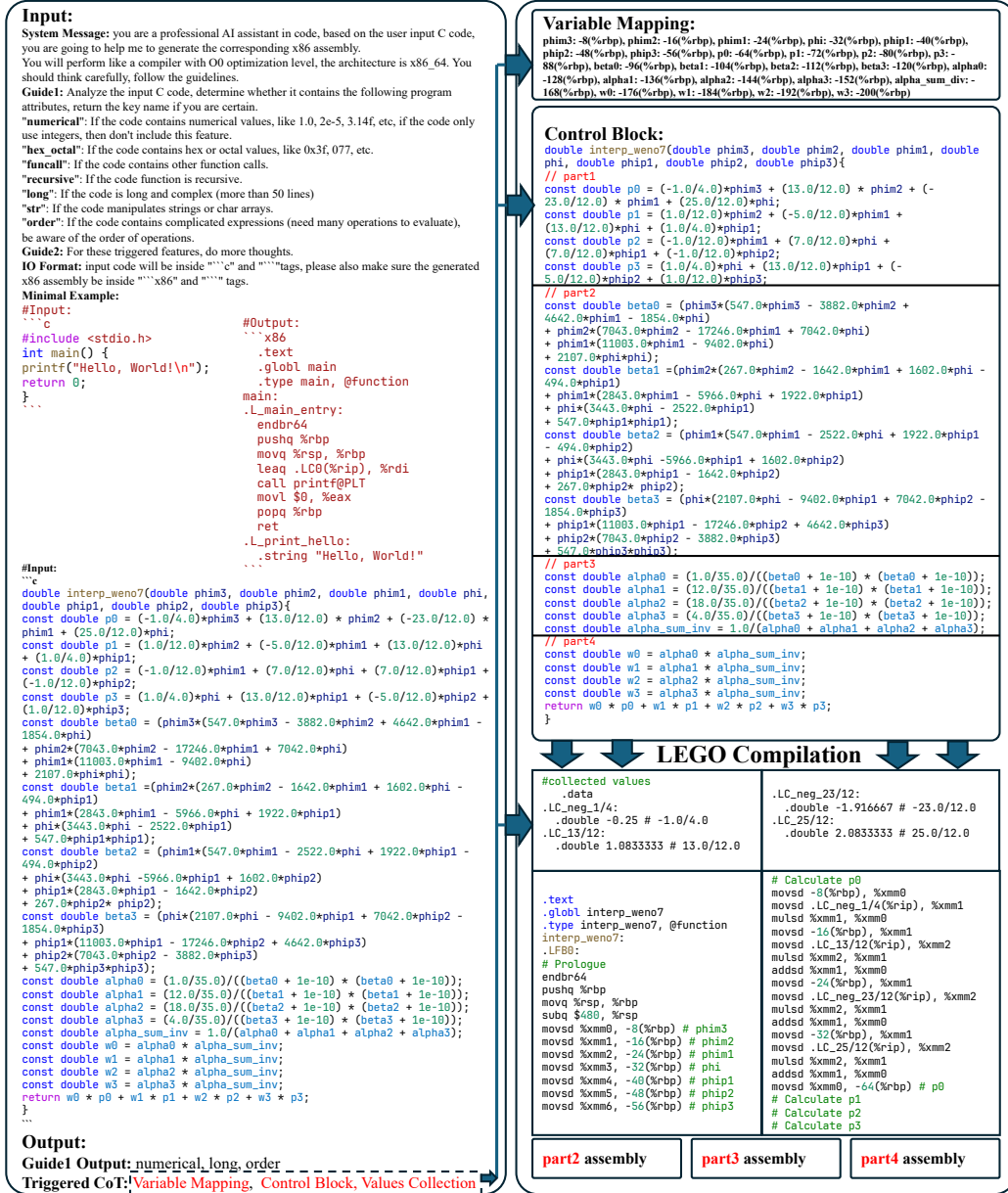


Figure 4: Example workflow for LEGO-Compiler on a full ExeBench example: source code analysis triggers thoughts, including **variable mapping**, splitting **control blocks** and **value collection** illustrated.

Algorithm 2 LLM-driven **Part Split** Algorithm based on Control Blocks

```
procedure SELECTCONTROLBLOCKS(function)
  blocks  $\leftarrow \emptyset$ 
  deque.push_back(function)
  while deque is not empty do
    block  $\leftarrow$  deque.pop_front()
    decision  $\leftarrow$  LLMDecideSplit(block)
    if decision is "keep" then
      blocks.append(block)
    else
      subBlocks  $\leftarrow$  SplitByOutermostControl(block)
      for subBlock in subBlocks in reverse order do
        deque.push_front(subBlock)
      end for
    end if
  end while
  return blocks
end procedure
```

- **Variable mapping:** Another necessary thought, which will base on the scanned variables and their types, and form a variable mapping table (SymbolTable) for later compilation.
- **Control Block:** the LEGO translation methodology is applied triggered by **long**, where the entire code is considered too long and will be split into control-block level code snippets via Algorithm 2, it's noteworthy that the **order** flag from analysis will suggest the LLM to split the program into finer-grained blocks so that they can focus more on the order of operations within each block, in Figure 4, there is just one basic block, the flag suggests LLM to split into 4 sequential parts. Then these parts are translated with the aid of SymbolTable individually. Finally, these compiled results are composed together to form a full LEGO compilation.

Therefore, we apply a *residual* step-by-step workflow. With different input code, the triggered CoTs will be different, and non-triggered CoTs are just skipped with a residual connection.

C.2 ExeBench breakdown

The failure cases observed in ExeBench can be majorly categorized into three types:

- The insufficiency on some language-specific features, for example, lacking the knowledge of certain operations, which can be definitely improved with more data in the next model pretrained or by providing external knowledge to aid its generation.
- The unsuccessful reasoning step in the workflow. This method requires the LLMs to reason arithmetic computation and capture specific code patterns in the code to form intermediate results to aid the generation. If the reasoning process generates incorrectly (rare), the whole workflow will fail. However, the reasoning capabilities required for this method is not high, majorly the addition and multiplication of integer values within 1000(typically). The error-feedback loop can mitigate most of trivial errors during reasoning. Besides, as LLMs keep improving their abilities in reasoning and math, this type of failures will reduce significantly.
- Very long code reasoning and follow-up generation, where LLMs fail to generate a very large output at once. The first reason is the limitation of current LLMs themselves, although advanced LLMs have increased their context limits into hundreds of thousands tokens, their single generation capability is still limited, to either 4096, 8192 or 16384 tokens. The second reason is the difficulty to generate a long, error-prone output(like assembly languages) at once, this is an intrinsic drawback of direct generation method itself, and can be improved greatly with the proposed LEGO translation/compilation method. LEGO translation can reduce the complexity to control block level, or at maximum, statement level, however, if the statement itself is very long and complicated to evaluate (which is very rare during

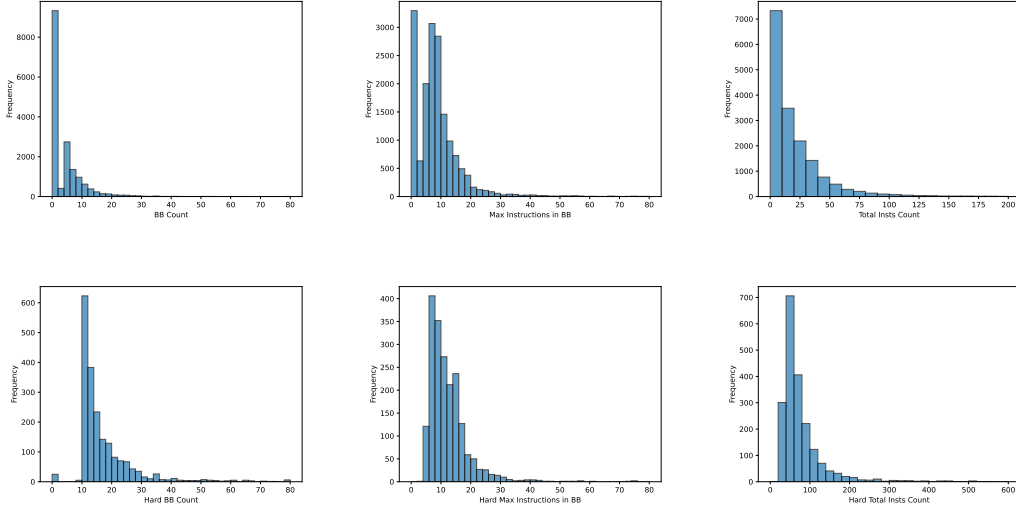


Figure 5: Complexity breakdown of ExeBench and its hard 10% (roughly) subset, we use llvm as the analysis tool, then filter the subset with the following conditions: number of basic blocks(BB) ≥ 10 or max instructions in BB ≥ 80 or total instructions ≥ 200 . Upper figures characterize the overall of Exebench and Lower figures characterize the hard 10% subset.

Table 4: Ablation study: impact of temperature on Pass@1 and Pass@5 performance

Model	Pass@1					Pass@5				
	0.2	0.4	0.6	0.8	1.0	0.2	0.4	0.6	0.8	1.0
GPT-4o	71%	73%	72%	72%	72%	79%	83%	86%	89%	92%
Claude-3.5-Sonnet	87%	91%	93%	88%	89%	91%	92%	96%	94%	96%
DeepseekCoder	89%	88%	86%	87%	88%	92%	92%	92%	93%	92%
GPT-4o-mini	64%	61%	61%	60%	60%	71%	71%	79%	73%	80%
Claude-3-Haiku	79%	76%	78%	72%	73%	82%	84%	85%	86%	86%

evaluation, but potential in modern programming paradigms), our method doesn’t tackle it, which is a limitation in our work. However, more program-rewriting steps can help to mitigate such issues and is verifiable, and we leave it as future work.

C.3 Other evaluation details

Table 4 shows the impact of temperature when using LLMs for neural compilation. LLMs have better Pass@1 accuracy when temperature is low, but higher Pass@5 accuracy when temperature is high. This is as expected, since temperature influences the decoding process, with higher temperature, the results are more diverse, allowing LLMs to jump out of pretraining bias, however, this could also cause more errors by choosing sub-optimal decoding tokens that may cause errors.

D Limitations

Optimization Capabilities: The major focus of LEGO-Compiler is on the translation correctness rather than code optimization. Traditional compilers excel at producing highly optimized code, a capability not yet matched by our neural approach. Although we do find LLMs are capable to optimize the translation process individually and generate optimized assembly code, it is not our main focus. Future work could explore integrating optimization techniques into the neural compilation process.

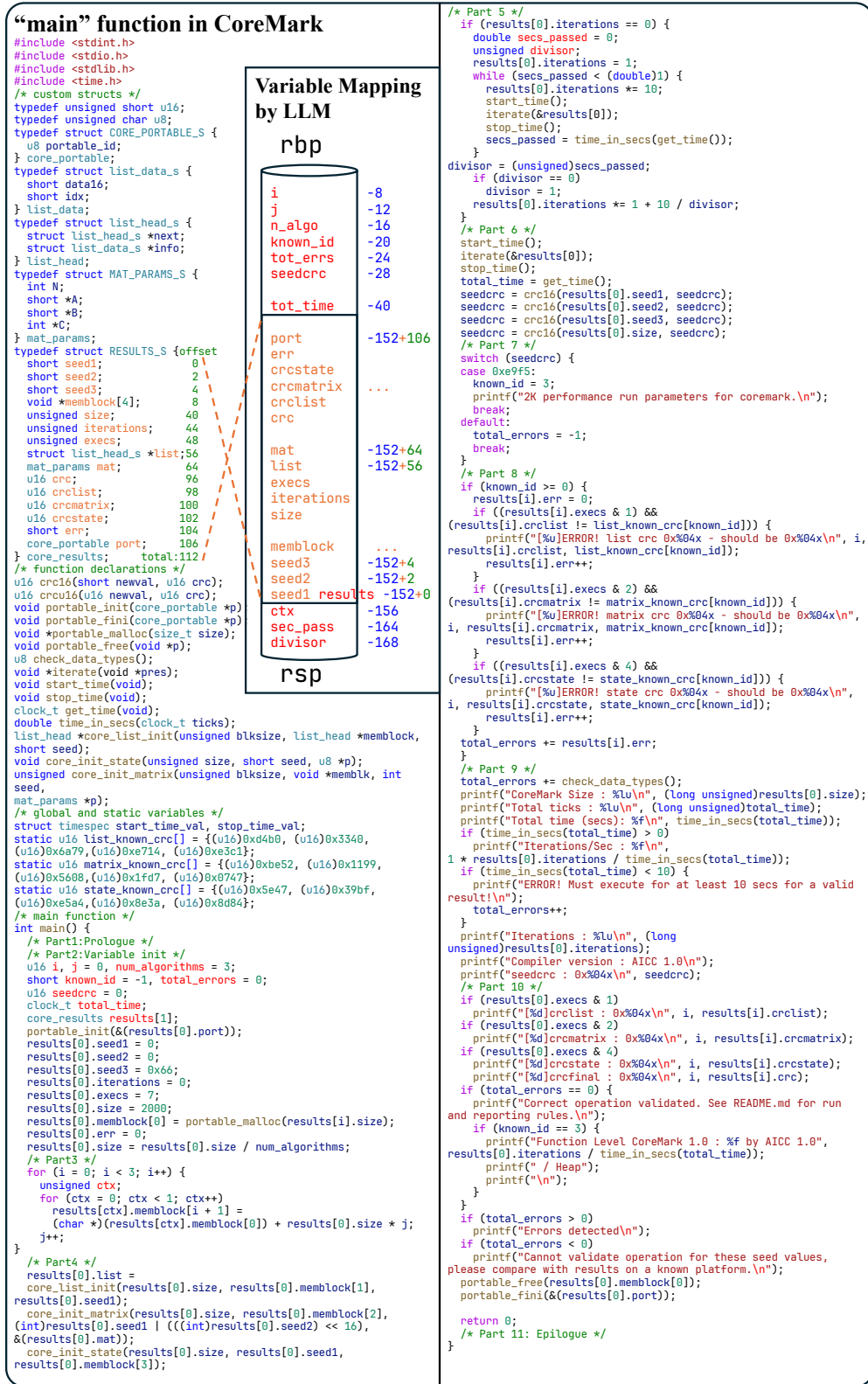


Figure 6: The CoreMark main function, one of the most difficult code we evaluated. In this figure, all CoTs are illustrated in the code annotations in color, as well as the variable mapping process.

```

static uint8_t func_1(void)
{
    int64_t l_2[1];
    int32_t l_3 = 0xF37831E4L;
    int32_t l_6[3];
    int i;
    for (i = 0; i < 1; i++)
        l_2[i] = 0xEC2E0CF5720E83C7LL;
    for (i = 0; i < 3; i++)
        l_6[i] = 0xA8CDA2AEL;
    for (l_3 = 0; (l_3 >= 0); l_3 -= 1)
    {
        int16_t l_4 = (-1L);
        int32_t l_5 = (-1L);
        int i;
        l_5 = ((l_2[l_3] != 1UL) <= l_4);
        l_6[0] = l_4;
    }
    l_6[2] = l_3;
    return l_6[0];
}

struct S0 {
    uint8_t f0;
    int32_t f1;
    uint16_t f2;
};

struct S1 {
    struct S0 f0;
    uint32_t f1;
    struct S0 f2;
    uint16_t f3;
};

static struct S1 func_1(void)
{
    uint32_t l_4 = 0xF054A20AL;
    int32_t l_5 = 0x4B03E386L;
    uint8_t l_6[3];
    struct S1 l_11 = {
        {0x8EL, 0x36DC9922L, 0xC436L},
        4294967295UL,
        {1UL, 0xC3FC0233L, 0xD52AL},
        0x2BBDL
    };
    ...
    return l_11;
}

```

Figure 7: Csmith example code, the major body part of the right hand side code is omitted. This example characterizes the necessity of both the Chain-of-Thought reasoning of structs and stack allocation and the LEGO translation method to overcome the complexity of coarse-grained translation.

Performance Overhead: As noted in the discussion, the computational cost of neural compilation is significantly higher than traditional methods. This limitation may restrict its practical application in scenarios where compilation speed is critical.

Complex Expression Handling: The paper acknowledges challenges in managing highly complex expressions, proposing external tool integration or expression decomposition as potential solutions. This indicates a current limitation in LLMs’ ability to handle intricate code structures independently.

Architecture-Specific Knowledge: While the paper demonstrates success with x86, ARM, and RISC-V architectures, expanding to a broader range of architectures, especially more specialized ones, may require significant additional training or fine-tuning of the LLMs or providing large RAG database to provide such knowledge in the context.

Security and Reliability: The stochastic nature of LLM outputs raises concerns about the consistency and security of the generated assembly code. Ensuring deterministic outputs and preventing potential vulnerabilities introduced by the neural compilation process remains a challenge.

Handling of Language-Specific Features: The paper primarily focuses on C-like language compilation and proves the availability of functionality in neural compilation through both theoretical and empirical results. However, extending the approach to other programming languages can result in more tailored problems, for example:

- **RAII idiom:** Languages with class properties, like C++, have an important programming idiom called **Resource Acquisition Is Initialization (RAII)**, which pose significant challenges for LLMs. For instance, constructor and destructor functions in these languages are implicitly called based on scope. This implicit behavior is difficult for LLMs to accurately model and implement in assembly code.
- **Name Mangling:** Languages like C++ and Rust use name mangling mechanisms for function overloading and template instantiation. This requires special handling of global symbols, such as function names, during compilation, which may be challenging for LLMs to consistently implement without explicit training on these concepts, but this could be solved by using external mangling tools like **c++filt** [17].
- **Dynamic Language Features:** Some language features violate the composability principle that LEGO translation relies on. For example, Python’s exception handling mechanism,

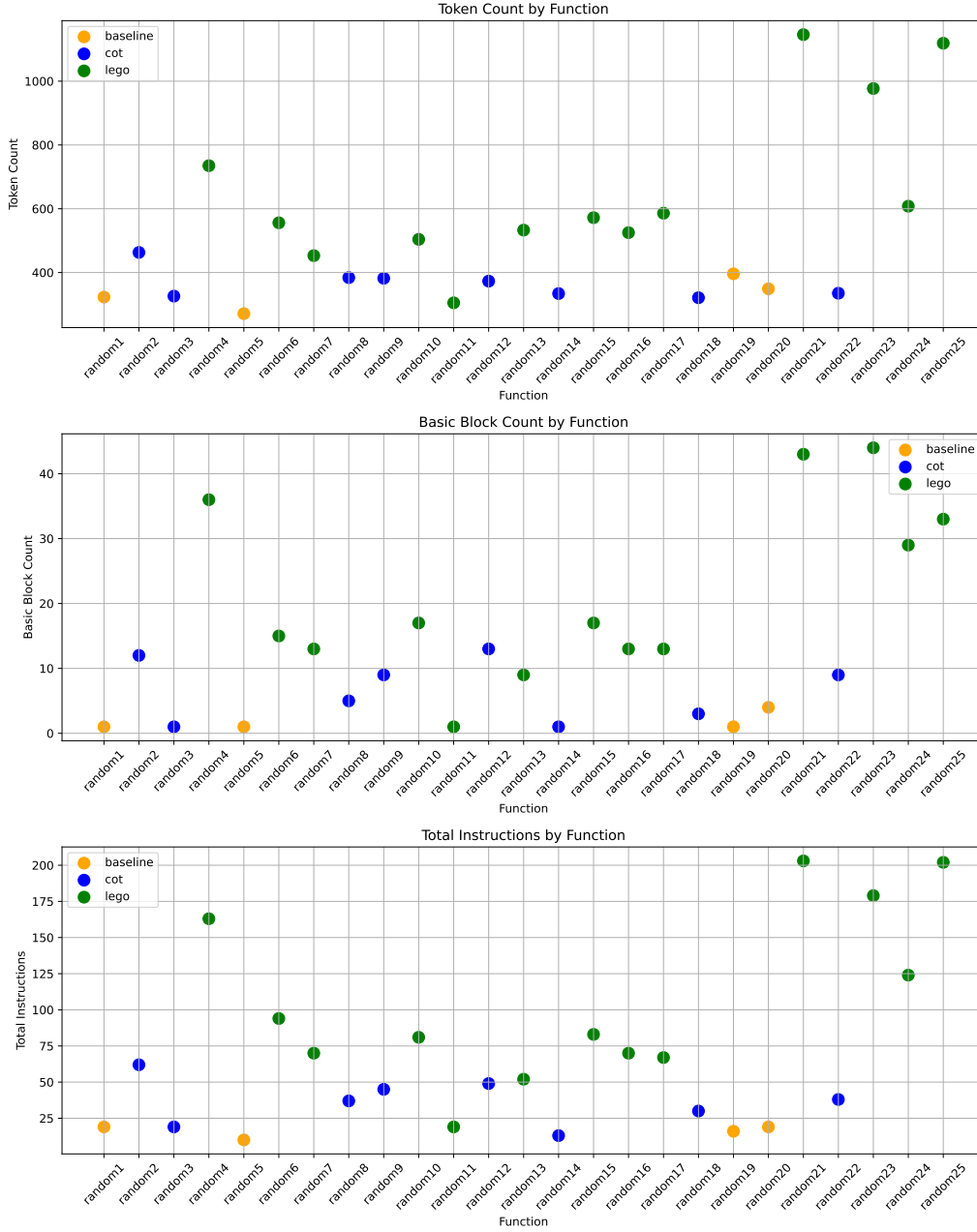


Figure 8: Csmith random generated code statistics, where the practical utility of the LEGO method is show clearly by passing significantly more complex cases.

which can cross scope boundaries, would make the LEGO translation method ineffective for such features.

It's important to note that many of these challenges are not unique to neural compilation. Traditional compilers also struggle with highly dynamic features like exception handling and Run-Time Type Information (RTTI). Languages like Python achieve flexibility by sacrificing native code generation in favour of interpretation or JIT compilation. Therefore, these limitations are not specific to our work but rather inherent to any approach based on static compilation analysis.

The ability to handle these diverse language features represents an area for future research in neural compilation. It may require developing specialized techniques or combining neural methods with traditional compiler approaches to address these complex language-specific challenges.

Scalability to Very Large Codebases: While the LEGO translation method significantly improves scalability, handling entire large-scale software projects or operating systems may still be beyond the current capabilities of this approach. However, It is noteworthy that repository complexity is naturally reduced into files or functions, therefore, LLM-based compilers and translators are potential to translate them with more advanced models and more carefully designed methods.