Large language models: compilers for the 4^{th} generation of programming languages?

Francisco S. Marcondes
ALGORITMI Research Centre/LASI, University of Minho, Braga, Portugal

José João Almeida
ALGORITMI Research Centre/LASI, University of Minho, Braga, Portugal

Paulo Novais
ALGORITMI Research Centre/LASI, University of Minho, Braga, Portugal

Abstract

This paper explores the possibility of large language models as a fourth generation programming language compiler. This is based on the idea that large language models are able to translate a natural language specification into a program written in a particular programming language. In other words, just as high-level languages provided an additional language abstraction to assembly code, large language models can provide an additional language abstraction to high-level languages. This interpretation allows large language models to be thought of through the lens of compiler theory, leading to insightful conclusions.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Computing methodologies → Natural language processing; Software and its engineering → Compilers

Keywords and phrases programming language, compiler, large language model

Digital Object Identifier 10.4230/OASIcs.SLATE.2023.6

Category Short Paper

Acknowledgements This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020

1 Introduction

As the title suggests, this is a speculative paper discussing whether large language models could be considered a higher level of programming language in relation to current high-level languages. In short, assembly language (2^{nd} generation) replaced punch-card programming (1^{st} generation) by introducing mnemonics. These allowed larger and more complex programs to be created in less time. High level languages (3^{rd} generation) in turn replaced assembly language by introducing structured English constraints.

The hypothesis explored in this paper is that large language models could be a 4^{th} generation language, replacing high-level languages by allowing natural language specifications. The aim is then to discuss the strengths and weaknesses of large language models running as *natural to high-level language* compilers. Other natural language processing tasks that would be associated with an interpreter are therefore beyond the scope of this paper. The question that arises is whether large language models can provide an additional level of abstraction for programming [10] similar to the high-level languages provided for assembly language.

As a disclaimer, this paper uses ChatGPT as a basis for discussion, but does not consider ChatGPT to be a compiler. A 4^{th} generation compiler, as well as a 3^{rd} generation compiler, is expected to produce executable code as a result, not intermediate code as ChatGPT does. Note that there are fine-tuned solutions for programming, such as the *GitHub co-pilot*, but due to its current paywall *ChatGPT 3.5* is used in this paper. Although ChatGPT is used to support this discussion, a proper setup should be designed for proper use of the large


© Francisco S. Marcondes, José João Almeida and Paulo Novais;
licensed under Creative Commons License CC-BY 4.0



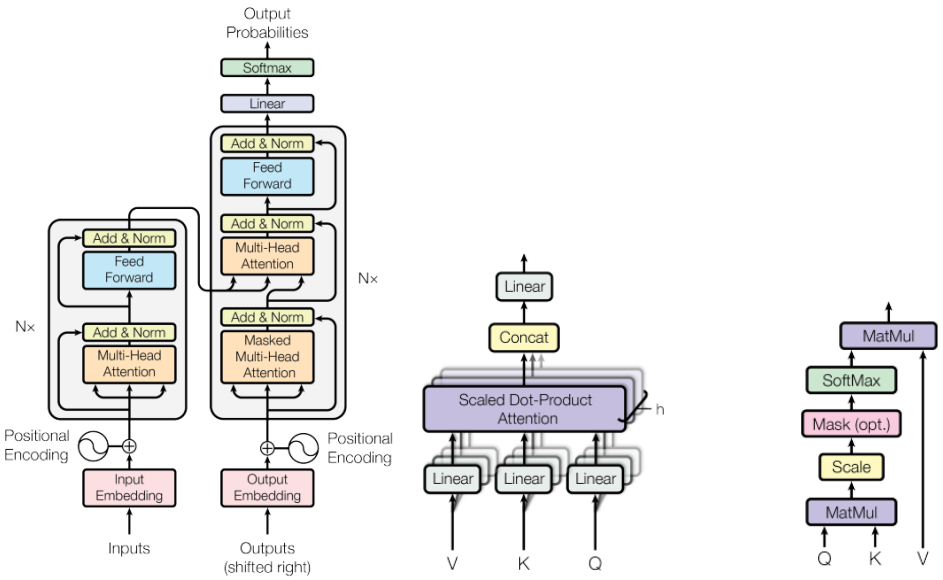
language model as a compiler. In this setup, a programmer is not expected to interfere with the lower-level code any more than a programmer is expected to modify the assembly code generated by a high-level language compiler.

Furthermore, a single prompt is not expected to be sufficient to produce industrial-scale software, nor is a single specification sufficient to achieve the same end. A typical requirement analysis takes several months and results in thousands of mutable requirements whose complexity is managed by interactive and incrementally throughout the project [14]. There is no reason to believe that this will change with the replacement of compiler technology. Therefore, industrial-scale software would require several mutable prompts in order to properly specify an industry level software product.

It is also worth noting that this proposal is not related to no-code or low-code initiatives. In short, these initiatives aim to provide a different way of doing traditional programming, but without providing the higher level language abstraction that large language models do. Despite several papers are addressing large language models for code synthesis, it was not possible to find any discussing it from the compiling perspective [16]. It was also not possible to find papers referring to 4^{th} or next generation compilers linked with large language models; most are linked with model-driven [12] or domain-specific [17] approaches.

2 Theoretical Foundations

Current large language models are based on Transformers (see figure 1). Perhaps the core element in Transformers is the *attention mechanism* [5]. In brief, given an input embedding, the attention mechanism reweights the embeddings for each token in the input to match the context of the input sentence. For example, let `bank` be a token in the input sentence whose sense is equidistant from `river` and `money` in the embedding model. Multiplying the embedding values of `bank` by the embeddings of the other words in the input sentence is expected to bias the `bank` token towards the appropriate sense.



(a) Overall Architecture. **(b)** Multi-head attention. **(c)** Scaled dot-product attention.

Figure 1 Transformers *cf.* [13]. Note that (c) is in (b), which is in (a).

The attention mechanism is the underlying principle of *prompt engineering*. In a nutshell, prompt engineering is concerned with designing a prompt that, when queried by a large language model, returns the best possible answer. As a rule of thumb, a prompt *cf.* [9] consists of: a) instruction; b) context; c) input data; and d) output indicator. Prompt engineering is therefore less about acting out a natural language conversation, and more about describing specific instructions aimed at an output.

This leads to the almost straightforward conclusion that large language models can be understood as a natural *language processor* (see [1]). A compiler, or translator, is a type of language processor that converts sentences from a usually high-level source language to an often low-level target language. The compiled sentences are interpretable by the target machine, which behaves accordingly. Considering that Transformers was built with the goal of natural language translation in mind [13], a relationship between Transformers and compilers can be suggested, with the source language being a natural language and the target language being a high-level language.

3 Proof of Concept

For reference, consider the introductory programming class problem:

> *The soldiers of the queen of hearts have a problem: once again the queen has sent them to fetch cookies for tea. Five of the soldiers went to get the cookies and returned. Since only one of the soldiers can enter the tea room with the cookies, they need to choose one of them. The problem is that the queen is greedy and has very little patience. Either they quickly figure out which soldier brought the most cookies or they will lose their heads. Your task is to write a program to find the answer.* (1)

The above problem has been prompted in ChatGPT as is. The result is shown in the figure 2a. Recall that a prompt *cf.* [9] is composed of: a) instruction; b) context; c) input data; and d) output indicator. These are the elements being required by the model. ChatGPT does not always give the same output for the same prompt, this is due to the Q, K, V (stands for query, key and value) weight matrices used by the attention mechanism (see figure 1). On a second run of this prompt in a different ChatGPT instance, the language model adopted the previously requested parameters and returned the source code shown in 2b. Therefore, in order to reduce variation, it is necessary to be as specific as possible on prompt building.

Note that the code in figure 2b is well-structured Python source code, capable of running on a Python machine. An immediate assumption is that by providing a formal specification, the resulting code would be enhanced. Prompt (1) is then rewritten as prompt (2) using simple set notation and submitted to ChatGPT, resulting in the code shown in figure 3. Note that the code in figure 3 is not necessarily better or more readable than the code in figure 2b. Such an assumption is therefore not necessarily true in the domain of large language models.

> *Let* $S = \{s_1, ..., s_n | n = 5\}$ *be a set of soldiers,* $C = \{c_1, ..., c_n | n = 5\}$ *be the number of cookies brought by each soldier, and* $f : S \to C$ *is a function that returns the number of cookies of each soldier. The soldier with the maximum number of cookies is given by* $max(\{f(s) | \forall s \in S\}) \vdash min(s)$*. Write a program to implement this algorithm.* (2)

The formal assumption is probably based on the idea that since natural language is ambiguous, some mathematical notation is necessary. This is true in so far as it is not possible to go through a process of clarification by asking questions, which is not the case in ChatGPT, as shown in figure 2a. This does not mean that formal expressions should be avoided, but, as

Sure, I can help you with that! Can you please provide more details on the input and output format expected for the program? For example, how are the number of cookies brought by each soldier represented in the input? And what should the output of the program be?

(a) First response.

Sure, I can help you with that!

Assuming you have a list of the number of cookies brought back by each soldier, here's a Python program to find the soldier who brought the most cookies:

python Copy code

```
cookies = [4, 7, 2, 9, 5] # list of cookies brought back by each soldier

max_cookies = max(cookies) # find the maximum number of cookies
max_index = cookies.index(max_cookies) # find the index of the soldier who b

print(f"Soldier {max_index+1} brought the most cookies: {max_cookies} cookie
```

In this program, we first find the maximum number of cookies using the `max()` function, which returns the largest item in a list. We then use the `index()` method to find the index of the soldier who brought the maximum number of cookies. Note that we add 1 to the index because soldiers are usually numbered starting from 1 instead of 0.

Finally, we print out the result using an f-string, which is a convenient way to format strings in Python.

I hope this helps!

(b) Second response.

Figure 2 ChatGPT's responses for the same prompt on different instances.

python Copy code

```
def find_soldier_with_max_cookies(soldiers, cookies):
    max_cookies = max(cookies)  # Find the maximum number of cookies
    max_cookie_soldiers = [soldier for soldier, cookie in zip(soldiers, cooki
    return min(max_cookie_soldiers)  # Find the soldier with the minimum inde

# Example usage
soldiers = ["s1", "s2", "s3", "s4", "s5"]
cookies = [4, 2, 4, 3, 1]

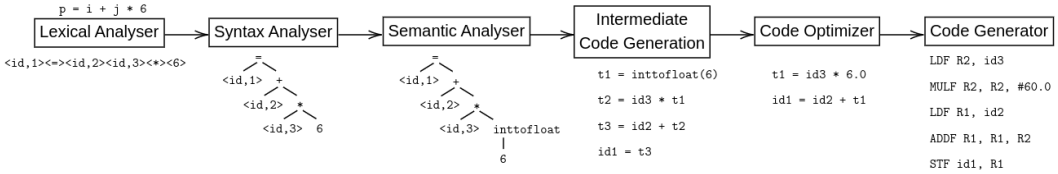
result = find_soldier_with_max_cookies(soldiers, cookies)
print("Soldier with maximum cookies:", result)
```

Figure 3 Source code produced by ChatGPT for the formal specification on prompt (2) with circumventing text removed.

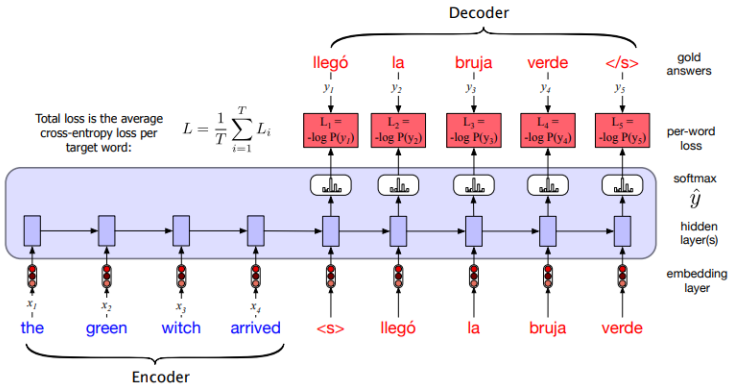
suggested after figure 3, that it is necessary to understand how to use them in this context. Also, that (2) may not be the best way to provide formal specification on ChatGPT.

4 Insights on a 4^{th} Generation Compiler

At this point it is possible to suggest that ChatGPT (*i.e.* large language models) can be considered as a translation device. Figure 4 shows a bird's eye view of a compiler as defined by Aho *et al.* in [1] and a translator as defined by Jurafsky *et al.* in [5]. The word "translator" is accepted as a synonym for "compiler", so the former can be considered a deterministic translator and the latter a probabilistic one.



(a) A compiler instance as presented in [1].



(b) A translator instance as presented in [5].

Figure 4 Deterministic and probabilistic translator instances.

It can also be argued that the deterministic translator is primarily concerned with syntax (i.e. the structure of sentences based on the Chomsky hierarchy) and the probabilistic translator with semantics (i.e. the relationship between words based on the distributional hypothesis). Therefore, it is not a case of replacement, but of composing these two translation strategies. As a result, if a source is generated with syntactic errors, the generator would produce an improved source by using the error messages as feedback. Note that the use of examples is an important prompt feature due to the few-shot learning property of large language models [2]. This in turn can be manifested as unit tests [11]. In this sense, not only syntactic errors but also behavioural errors eventually introduced by the generator can be automatically corrected. Note also that, not all information may be provided in a prompt. There are two situations to consider. One is the prompt within a project, from which the additional information can be retrieved (*e.g.* elements of a class, UML blueprints, *etc.*). If the required information is ambiguous or missing, the compiler is expected to prompt the programmer as shown in figure 2a with each interaction improving either the context or the specification. Note that the correct definition of the prompt is the cornerstone of improved generation; a context with more information than necessary is just as harmful as a context with no information at all. A structured as such is depicted on figure 5.

It is worth noting that if the complexity of the specification becomes unmanageable, it can be split or simply deleted to start again. Splitting opens up the possibility of tackling

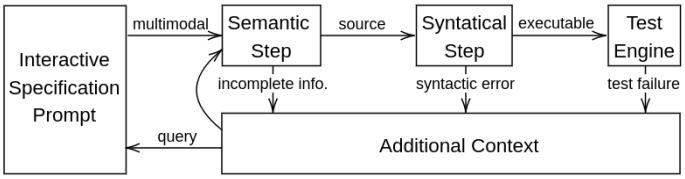


Figure 5 Suggested structure for a 4^{th} generation compiler based on large language models.

increasingly large software. Regardless of the development process used (prescriptive or agile), the overall organization of the project elements ends up being arranged in a tree-like structure. The same structure can be applied to a series of prompts that make up the software, helping to establish the correct context for each prompt. Consider the following Use Case 2.0 [4] partial scenario:

- Use-case: Registering to SLATE
  1. Authentication Flow
     a. The enrollee provides its e-mail, the SuD sends an e-mail with a confirmation link
     b. The SuD receives a confirmation and set the status to "authenticated enrolee"
  2. [authenticated enrolee] Basic Flow for Author Registration
     a. ...

For this discussion, consider step 1a. Note that this step is a self-contained specification and is sufficient to understand the desired behaviour. However, it is not a prompt in the sense that there is much information missing. As a reference, the 4+1 view model [7] suggests the existence of five views: 1) scenario, 2) logic, 3) development, 4) process and 5) physical. This step description only satisfies the scenario view. For example, on which server is this web service expected to run? In addition, a scenario is expected to satisfy the FURPS+ [3]; and this step only satisfies the "F". For example, what would the content of the confirmation email be? Therefore, prompting is not equivalent to a requirement analysis yet, as any development task, derives from it.

Note that steps 1a and 1b form a slice (a coherent part of a use case that can be elaborated into a deployment [4]). Therefore, to produce a deliverable for this slice, a prompt must be written from these two steps. Considering the constraints presented, the 4+1 view model and FURPS+, it becomes clear that only a chat-based structure is not sufficient to express a whole, industrial-scale software (even a slice of it). One possibility of structure to explore is that provided by literate programming, see [6].

In this sense, from a human perspective, it is not a matter of producing a single prompt, but of producing a structured document composed of several prompts addressing different concerns. Given such a document, the compiler would be expected to make two moves. One towards the refinement of each prompt, *i.e.* a chat-based interaction aimed at producing an improved prompt. Another towards the unfolding of additional prompts, *i.e.* the creation of additional slots for prompts to address specific problems. For large software, there could be several specification files, each for a slice. In this perspective, the compiler would act as a specification co-pilot. Note that the context window used by ChatGPT is about 2048 tokens, so managing the context of each prompt is also a feature to be considered.

Another support expected from the compiler is the generation of test cases. Following a behaviour-driven development rationale [11], this leads the programmer to consider several scenarios that improve the resulting program. From the perspective of this paper, this means that the programmer would either refine previous prompts or introduce new prompts.

A possible literate programming early paragraph for the authentication slice could be the one presented in (3). Note that this is not a proper programming prompt yet, submitting it to ChatGPT, an excerpt can be seen in 6, it retrieves several suggestions that illustrate what the refinement and unfolding moves would look like. For example, validating the email with regular expressions would be a refinement on (3); describing the appearance of the HTML form (*e.g.* colour, logo, *etc.*) would be an unfolding prompt.

> *This programme is part of a conference registration platform and is designed to verify that the email provided by a registrant is a valid one. This requires: 1) a web page that allows the registrant to enter their email address; 2) a web service that receives this address and sends the confirmation email; and 3) another web service that generates the confirmation page and waits for the registrant to retrieve it.* (3)

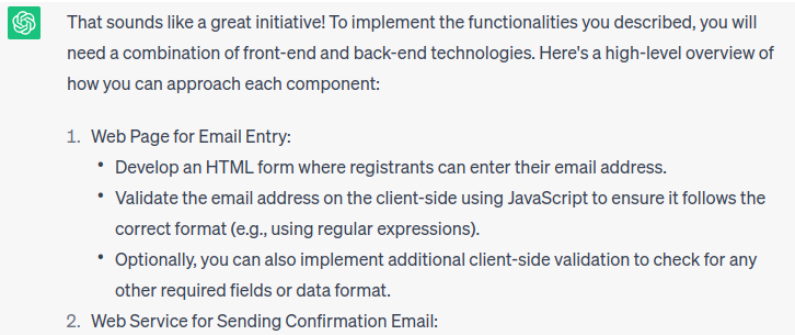


Figure 6 Excerpt of ChatGPT response when prompting (3).

The presented example is based on plain literate programming. Since large language models are currently turning into multimodal, it is also possible to enrich the specification with diagrams and other images [15].

5 Conclusion

This paper introduced the possibility of interpreting large language models as a fourth generation programming language, based on the notion that large language models are capable of translating a natural language specification into well-formed source code. This notion opens up and strengthens a wide range of research areas, including further developments from the compiler perspective to specific prompt engineering techniques for producing programs. The key discussion is that the current fears and suspicions raised by this technology are analogous to those that arose during the transition from the second to the third generation of programming languages. It is therefore a natural phenomenon that should be resolved on its own terms, as far as the proposal presented in this paper is concerned.

Comparing this approach with a current compiler raises the question of the probabilistic nature of large language models (the same prompt produces two outputs). In short, this can be addressed by a fixed, perhaps optimal, internal state with respect to the compilation task. However, the strengths and weaknesses of such an approach are a subject for future work. This is somehow related to explainability, a flourishing area of research that is expected to provide answers to questions about why a model has generated a particular piece of code.

Note that ChatGPT may produce incorrect code, but also that it is not tuned for programming, and it will eventually become outdated. However, it is expected that a 4^th

generation compiler will be able to deal with such problems, just as 3^{rd} generation compilers will be able to deal with code with syntactic problems (perhaps one way, inspired by genetic algorithms, would be to produce a few generations and select the best by a set of parameters). This assertion assumes that the specification has been written properly and correctly, but it is also necessary to consider that intentional and induction errors will still exist, but at a higher level of abstraction. In this sense, if the executable doesn't run as expected, the programmer should concentrate on fixing the prompt as he currently does with the source.

This leads to issues related to the dataset, which will also need to be addressed in future work. Then, considering the compilation task, it would be necessary to understand the composition of the dataset. Also, which setup would perform better: a general large language model or a fine-tuned one? What would be the fine-tuning parameters? Also, when considering prompting, which software engineering tools, methods and principles are appropriate and which are not. Note that multimodal prompting requires a refresh on model-driven development, see [8]. Finally, this seems to be a promising area of research to be explored further.

References

1 Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques and tools*. 2020.

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

3 Robert B Grady and Deborah L Caswell. *Software metrics: establishing a company-wide program*. Prentice-Hall, Inc., 1987.

4 Ivar Jacobson, Ian Spence, and Brian Kerr. Use-case 2.0. *Queue*, 14(1):94–123, 2016.

5 Dan Jurafsky and James H. Martin. *Speech and Language Processing*. draft (https://web.stanford.edu/jurafsky/slp3/), third edition, 2023.

6 Donald Ervin Knuth. Literate programming. *The computer journal*, 27(2):97–111, 1984.

7 Philippe B Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.

8 Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model driven architecture with executable UML*, volume 1. Cambridge University Press, 2004.

9 Elvis Saravia. Prompt engineering guide, 2023. URL: `https://www.promptingguide.ai/`.

10 Robert W Sebesta. *Concepts of programming languages*. Pearson Education, 2019.

11 J.F. Smart and J. Molak. *BDD in Action, Second Edition: Behavior-Driven Development for the Whole Software Lifecycle*. Manning, 2023.

12 Bernhard Thalheim and Hannu Jaakkola. Model-based fifth generation programming. *Information Modelling and Knowledge Bases*, 31:381–400, 2020.

13 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

14 Karl Wiegers. *More about software requirements*. Microsoft Press, 2005.

15 Zhuosheng Zhang, Aston Zhang, Mu Li, Hai Zhao, George Karypis, and Alex Smola. Multimodal chain-of-thought reasoning in language models. *arXiv:2302.00923*, 2023.

16 Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

17 Majd Zohri Yafi. *A Syntactical Reverse Engineering Approach to Fourth Generation Programming Languages Using Formal Methods*. PhD thesis, University of Essex, 2022.