# Developing Programming Languages for Ease of Use with LLM's

Zilong Li
The University of Colorado
Boulder, Colorado, USA
zili1126@colorado.edu

Scott McCall
The University of Colorado
Boulder, Colorado, USA
damc8609@colorado.edu

## Abstract

In this paper, we discuss the importance of prompting in large language models, and how performance on certain tasks greatly varies based on the quality and manner of the prompt itself, which can make interacting with these models difficult. Addressing these limitations is important due to the increased mainstream usage of large language models such as ChatGPT in a wide variety of contexts. The difficulty from improving the performance of these models comes from the billions of parameters that are used in these models, and the idea that fine-tuning them would be impractical for most users and impossible for models that are closed-source. Prompting is a good approach to retrieve satisfactory responses from large languages models, and it is user-friendly since using the natural language is enough. However, many prompting methods are associated with certain templates and the call of languages models' APIs. Our goal is to develop a programming language that can interact with an LLM such as ChatGPT in a way such that constraints and other conditions can be added to the prompt such that it provides a layer of abstraction to the user that makes it easier to interact with the LLM in a way that provides higher quality results.

**ACM Reference Format:**
Zilong Li and Scott McCall. 2023. Developing Programming Languages for Ease of Use with LLM's. In *Proceedings of* . ACM, New York, NY, USA, 8 pages.

## 1 Introduction

Large language models (LLMs) are complex neural networks that perform natural language processing to provide a text-based interface that takes user input and provides some sort of output accordingly. While these models have limitations with the tasks they are able to perform, sufficiently prompting these models in specific ways is what can allow them to perform tasks that they have not seen before. However, as one would expect, the quality of the output is inherently tied to the quality of the input. Any sort of convoluted or ambiguous query can yield poor results, and as a result, practicality of LLM's can entirely depend on the quality of the queries being presented to the LLM.

Over time, usage of LLMs such as ChatGPT have become much more mainstream and commonplace in a variety of tasks, including but not limited to developing code, mathematical computations, fact checking, and text generation. As a result, being able to obtain more reliable and accurate results would be beneficial to a wide variety of individuals for different use cases, and would accelerate completion of the relevant tasks.

Large language models are inherently very complex, requiring a lot of training data and having billions of individual parameters. For certain cases, these parameters can be fine-tuned using sufficient training data for specific tasks, but this is often impractical or impossible for the user depending on the LLM. This is why prompting LLMs has become the preferred method for performance on most tasks in daily use. However, due to this inherent complexity of LLMs, formalizing a prompting method that reliably increases accuracy is not a trivial task, which is why it is the subject of this research.

In this paper, we will outline the formalization of a programming language used to interact with LLMs. More specifically, this language was designed with built-in features that enable the kind of prompting proven to be effective by providing templates to achieve reframe prompting. Our goal was to essentially conform the model output such that we could demonstrate an increase in accuracy on certain tasks using specific prompting methods compared to traditional prompts based on certain test data sets.

The paper will go into detail in the following sections:

- Introduction of Reframe Prompting Templates into a Programming Language
- Formalization of the Dynamics and Statics of the Prompting-Based Language


, ,

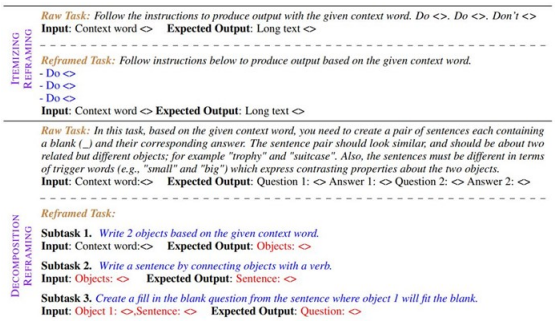
## 2 Related Work

Many scholars have explored the method to make programming languages interact with LLMs. They have meaningful approaches which inspire us.

Openprompt [1] is a kind of language which is designed to improve the behavior of the language model by using prompting methods. Based on their investigation of prompts towards language models, they manually designed many templates and frameworks which can be used for better prompts. Users can write better prompts for classification tasks with their language. Users can also process the response of language model such as using python function strip() to change the format of response.

Semantic Kernel [2] is another language designed by the Microsoft. It is used for the interaction between language models and programming language codes. Users can write language with variables and the semantic kernel will let the language model fill in the variables. It also supports the function call. Users can call external functions before and after the call of language model. This increases the function of languages models.

A more powerful languages we encountered when researching implementations of languages that interact with LLMs is called Language Model Querying Language (LMQL). [3] LMQL provides a methodology for interacting with LLMs in such a way that the user can introduce constraints as variables as an attempt to guide the output of the model. However, this approach does nothing to the query body of the prompt. Sometimes, a chain of queries can be more powerful than a single query. [4] The chain template is shown below:




Upon further research into prompting techniques, we discovered a methodology of prompting using templates for reframing prompts, in which either individual tasks are decomposed into subtasks or individual instructions are itemized, which was demonstrated to improve the performance of the queries and the quality of the output. [5] These techniques will be the basis of the prompting methodology we try to incorporate and formalize as an aspect of our programming language in order to improve the output of LLM queries.

… [variable1] …… [variable2] …… [variable3] …

where (variable1 … or variable3 …) and variable2 …

LMQL has a problem in its "or" operator. The graph above is an demonstration of this problem. Assume that we have three variables and their corresponding constraints in a query. The first and final constraints are connected by an "or" operator and they together appear in a conjunction with the constraint of variable2. If the first constraint is not satisfied, we can go to evaluate others, since we have an "or" operator here. However, if the final constraint is not satisfied, we need to go back to the first one to keep it satisfied. This backtrace will make the language unstable, because it may change the value of variable2 which is dependent on the previous context. This may result that the variable2 does not satisfies the constraint.

In this paper, we try to expand the LMQL language by introducing task decomposition [6] and limiting the usage of "or" operator to one single variable.

In order to evaluate the performance of certain prompts compared to modified versions that use reframing techniques, we decided to adopt evaluation criteria seen in *Prompting is Programming: A Query Language for Large Language Models.* [**?** ] These include:

1. Expressiveness: Can we easily implement common and advanced prompting techniques with simple and concise query logic, especially in the case of interactive prompting?
2. Performance: Can it be used to effectively lower the required number of model queries and thereby reduce the computational or API-related cost of using LMs?
3. Accuracy: Does its constrained decoding affect the task accuracy of LMs when evaluated on standard benchmarks?

These evaluation methods are the methods that will primarily be applied to evaluate our research and the performance of the modified language. They provide a framework for assessing how well prompting methods and languages perform, which is critical for development and optimization. We can quantitatively assess performance based on number of queries, and accuracy based on the percentage of tasks performed correctly. Expressiveness will be evaluated qualitatively based on the complexity of created queries in our developed language.

## 3 Language Semantics

For our language semantics, we provide a framework similar to that introduced in LMQL. Specifically, we allow a user to construct sentences out of strings and variables, and then construct a trace out of these sentences, which can be decoded using various language models. The formalization of these semantics is shown below:

**variable** v::= Var
**sentence** s::= String | Empty | [Var] | {Var} | $s_1; s_2$
**trace** u::= $\epsilon$ | us
**scope** $\sigma$
**decode function** u $\longrightarrow$ String
**execution terminate ok** then return u
**ternary expression**

$$\frac{}{<\text{String},u,\sigma>\longmapsto<\text{Empty},u\text{String},\sigma>} \qquad \frac{}{<\text{Empty},u,\sigma>\longmapsto u\ ok}$$

$$\frac{}{<[\text{Var}],u,\sigma>\longmapsto<\text{Empty},u\text{decode}(u),\sigma[\text{Var}]=\text{decode}(u)>}$$

$$\frac{}{<\{\text{Var}\},u,\sigma>\longmapsto<\text{Empty},u\sigma[\text{Var}],\sigma>} \qquad \frac{<s_1,u,\sigma>\mapsto<\text{Empty},u',\sigma'>}{<s_1;s_2,u,\sigma>\longmapsto<s_2,u',\sigma'>}$$

We define two kinds of variables with different semantic meanings. A variable with square brackets like [Var] means that we need to input the previous context to the language model and assign the response to the variable in its scope. When we encounter a variable with curly braces like {Var}, we do not need to call the language model, but we need to retrieve its value from the current scope. Empty means that there is nothing to execute. Usually, it represents the end of execution.

We define a trace here, which is the record of the execution. It starts with empty. After the execution of a sentence, our languages attaches the result of execution to the trace.

The decode function is the call of language model. Given a trace of execution, it will return a string which we can make use of for a variable.

The sequence of the query body is executed in a left-to-right order. When the previous sentence is executed, the result is attached to the trace and then our language goes to execute the next sentence. When all of sentences have been executed, the remaining query body is empty. Here, we stop the execution and print the trace to the user.

To formalize this algorithm, we define a ternary operation $< s, u, \sigma >$. The first component of this operation is the sentence that we are currently executing. The second one is the trace which keeps the history of execution and it can also be used as the input to the language model. The final one is the scope of the current task. It stores and map variable names to values. If [Var] is encountered during the execution, the scope may be altered.

## 4 Completeness Semantics

For our language, it is required that each query should be well-formed. In this case, "well-formed" means that we should not call the variable which is not defined or which is not in the current scope.

The execution of the our language is quite expensive in terms of time and money, since we need to call the language model. We should not check if the query is well-formed during the execution, since once a part of a query is not well-formed, the previously executed part is wasted. Instead, we need to make sure that each segment of a query is well-formed before execution. Thus, we define a static checking semantics to confirm the completeness of a query. If a query dose not pass the check, our language will raise an exception and denote the specific part which cannot pass the completeness checking. The definition of completeness is given as follows:

Com($\top$) This fragment of language is complete
Com($\bot$) This fragment of language is incomplete

As a preliminary to being able to evaluate completeness, we need to be able to check if a variable name has been assigned before prior to its usage. As a result, it is necessary for the language to include the following helper function:

**checkbefore function**: [var] $\longrightarrow$ boolean

The function will return true if a variable has been used previously, and false if not.

Now that we have laid the groundwork to sufficiently describe completeness, the semantics for checking whether a language fragment in the query is complete is given as follows:

$$\frac{}{<\text{String},\sigma>\Downarrow\text{Com}(\top)} \quad \frac{}{<\text{Empty},\sigma>\Downarrow\text{Com}(\top)} \quad \frac{}{<[\text{Var}],\sigma>\Downarrow\text{Com}(\top)}$$

$$\frac{\text{checkbefore}([\text{Var}])\Downarrow\textbf{true}}{<\{\text{Var}\},\sigma>\Downarrow\text{Com}(\top)} \quad \frac{\text{checkbefore}([\text{Var}])\Downarrow\textbf{false}}{<\{\text{Var}\},\sigma>\Downarrow\text{Com}(\bot)} \quad \frac{<s_1,\sigma>\Downarrow\text{Com}(\bot)}{<s_1;s_2,\sigma>\Downarrow\text{Com}(\bot)}$$

$$\frac{<s_1,\sigma>\Downarrow\text{Com}(\top) \quad <s_2,\sigma>\Downarrow\text{Com}(\top)}{<s_1;s_2,\sigma>\Downarrow\text{Com}(\top)} \quad \frac{<s_1,\sigma>\Downarrow\text{Com}(\top) \quad <s_2,\sigma>\Downarrow\text{Com}(\bot)}{<s_1;s_2,\sigma>\Downarrow\text{Com}(\bot)}$$

$$\frac{<\{\text{Var}\},\sigma>\Downarrow\text{Com}(\top)}{<\text{Var},\sigma>\Downarrow\text{Com}(\top)} \quad \frac{<\{\text{Var}\},\sigma>\Downarrow\text{Com}(\bot)}{<\text{Var},\sigma>\Downarrow\text{Com}(\bot)}$$

## 5 Constraints

Next, we need to introduce a set of constraints into the language that allow the user to constrain the output of the decoding performed on the model tokens. The constraints that we implement are defined below:

**constraint**: c::= $c_1; c_2$ | type(Var) | equal(Var; Pythonexp)
| stop_at(Var, Pythonexp) | Var in Pythonexp
| $c_1$ and $c_2$ | $c_1$ or $c_2$ | not c

$c_1; c_2$ represents two constraints connected by a semicolon. Both $c_1$ and $c_2$ should be constraints for different variables and both of them need to be satisfied.

type(Var) is a family of constraints. Our language currently supports common Python datatypes of `int`, `float`,

and `string`. Given a specific python datatype, this constraint makes sure that only the response with the corresponding format of datatype can pass it.

equal(Var;Pythonexp) is a constraint which requires the response of the language model to be equal to the value of the Python expression.

stop_at(Var,Pythonexp) indicates that when the language model is producing a sequence of words, if it encounters the value indicated by the corresponding Python expression, it should stop the generation and return the generated sequence.

Var in Pythonexp requires that the response of language model should match part of the Python expression. Our language supports list and string datatypes. That means the value of the variable should be an element of a list or a substring of a string.

$c_1$ and $c_2$ is a constraint for a single variable. It requires that this variable should satisfy both of two constraints.

$c_1$ or $c_2$ is also a constraint for a single variable. It requires that the variable should satisfy at least one of these two constraints.

not c is just the negation of the constraint. It means that this constraint should not be satisfied.

An important note for the above semantics is in regards to the limitation of one variable for the use of "and" and "or" constraints. One of the limitations of LMQL is that when constructing constraints, building constraints out of multiple variables can cause lots of backtracking, and an increase in the number of decoded tokens. This is because, in a compound constraint, the constraint may not be violated until one of the latter variables is unable to satisfy the constraint when it needs to be able to. For example, imagine one has a constraint where we use the OR operator to create a condition on $n$ different variables. For the first $n-1$ variables, none of the initial decodings can possibly violate the constraint when we still have unknown variables left to decode. When we decode the $n$-th variable, if we do not achieve a possible decoding that satisfies the constraint, one has to backtrack and analyze other decodings of previous variables, resulting in a combinatorial explosion. Thus, in pursuit of the goal of reducing the number of decoded tokens to reduce cost, it is reasonable to limit these compound queries to only be with respect to a single variable per constraint.

In order to define whether constraints are well-formed before issuing a query to the model, we extend our completeness semantics from before to statically verify constraints. The rules dictating completeness of constraints are shown below. The value of the binary operator is decided by the value of its parameters. The binary operator evaluates to Com($\top$), if and only if both its components evaluate to Com($\top$). It evaluates to Com($\bot$), if one of its component evaluates to Com($\bot$).

$$\frac{< c_1, \sigma > \Downarrow \mathrm{Com}(\top) \quad < c_2, \sigma > \Downarrow \mathrm{Com}(\top)}{< \mathrm{binary}(c_1; c_2), \sigma > \Downarrow \mathrm{Com}(\top)}$$

$$\frac{< c_1, \sigma > \Downarrow \mathrm{Com}(\bot)}{< \mathrm{binary}(c_1; c_2), \sigma > \Downarrow \mathrm{Com}(\bot)}$$

$$\frac{< c_1, \sigma > \Downarrow \mathrm{Com}(\top) \quad < c_2, \sigma > \Downarrow \mathrm{Com}(\bot)}{< \mathrm{binary}(c_1; c_2), \sigma > \Downarrow \mathrm{Com}(\bot)}$$

The unary operator has the same value with its component.

$$\frac{< c, \sigma > \Downarrow \mathrm{Com}}{< \mathrm{unary}(c), \sigma > \Downarrow \mathrm{Com}}$$

In order to make the 'and '/'or' only be applied to the same variable, we also need to define rules to judge the variables of 'and'/'or'. To implement it, we need to define a function var_name() which returns the variable name of a constraint:

For constraints like equal(Var, Pythonexp) | stop_at(Var, Pythonexp) | Var in Pythonexp | type(Var) | not c, var_name() returns the variable name of it.

We have rules for 'and' | 'or':

$$\frac{\mathrm{var\_name}(c_1) = \mathrm{var\_name}(c_2)}{<c_1 \text{ and } c_2,\sigma>\Downarrow \mathrm{Com}(\top)} \qquad \frac{\mathrm{var\_name}(c_1) \neq \mathrm{var\_name}(c_2)}{<c_1 \text{ and } c_2,\sigma>\Downarrow \mathrm{Com}(\bot)}$$

$$\frac{\mathrm{var\_name}(c_1) = \mathrm{var\_name}(c_2)}{<c_1 \text{ or } c_2,\sigma>\Downarrow \mathrm{Com}(\top)} \qquad \frac{\mathrm{var\_name}(c_1) \neq \mathrm{var\_name}(c_2)}{<c_1 \text{ or } c_2,\sigma>\Downarrow \mathrm{Com}(\bot)}$$

'and'/'or' evaluates to Com($\top$), if two subconstraints have the same variable name, and evaluates to Com($\bot$), if two subconstraints do not have the same variable name.

## 6 Dynamic Semantics with Constraints

The previous execution rule in Section 3 is naive, because it does not consider the constraint of variables. Since the designing goal of our language is making it easier for user to interact with the language model and impose constraints on the response of language model, we need to come up with more rules so that the execution of the query body is compatible with constraints. Here, we introduce a new operation with several rules to solve this problem. We call it the execution with constraints.

The difference between the previous naive execution and execution with constraints is lying in the execution of the variable with brackets [Var]. In cases without this type of variable, execution precedes identically to as in the naive case.

To implement the constraint, we introduce a new quaternary operation $< \mathrm{uv}_{1:j}, c, t_i, \sigma >$ which represents the token by token generation of language models. This operation has four components. $\mathrm{uv}_{1:j}$ is the history, in which u is the trace

we introduced before and $v_{1:j}$ is the historically generated sequence of tokens. $j$ can be 0 and $v_{1:0}$ represents that no token has been generated for the current variable. $t_i$ represents the token we are doing analyzing. It has an index $i$ which means it is the $i$th generated alternative token at a step. Depending on which decoding strategy we use, the language model may generate n alternative tokens at the each step. (n is 1 if we use the argmax strategy or it is assigned by the strategy top_k = n.) $\sigma$ is the scope of the current query task. It still stores and maps a variable name to its value.

Different from the previous semantics, when we encounter a variable with brackets during the execution of the query body, we go to the where clause and search the constraint for this variable. If it exists, we jump from the ternary operation to this quaternary operation. If it does not exist, we continue language execution with the naive execution semantics above.

$$\frac{}{<[\text{var}],u,\sigma>\longmapsto<uv_{1:0},c,t_1,\sigma>}$$

This rule explains the jump described above. When encountering a [var], our language let the language model generate a token based on the context $uv_{1:0}$.

$$\frac{<uv_{1:j},c,t_i,\sigma>\Downarrow\textbf{true} \quad t_i\neq<EOS>}{<uv_{1:j},c,t_i,\sigma>\longmapsto<uv_{1:j+1},c,t_1,\sigma>} \text{ where } v_{1:j+1}=v_{1:j}t_i$$

This rule explains that when the generated sequence together with the current token we are analyzing satisfies the constrain and the current token is not the special token $< EOS >$ (end of sentence), we need to attach the current token to the history and let the language model produce the next token based on the history. Then, we go to analyze the first possible token at the next step.

$$\frac{<uv_{1:j},c,t_i,\sigma>\Downarrow\textbf{true} \quad t_i=<EOS>}{<uv_{1:j},c,t_i,\sigma>\longmapsto<\text{Empty},\text{udecode}(u),\sigma[\text{Var}]=\text{decode}(u)>} \text{ where } \text{decode}(u)=v_{1:j}$$

when the generated sequence together with the current token satisfies the constraint and the current token is the $< EOS >$, our language stops the generation. Then, we assign the history $v_{1:j}$ to the variable in the scope and attach the history to the trace u. We have finished the execution of the variable.

$$\frac{<uv_{1:j},c,t_i,\sigma>\Downarrow\textbf{false} \quad i\leq n}{<uv_{1:j},c,t_i,\sigma>\longmapsto<uv_{1:j},c,t_{i+1},\sigma>}$$

If the generated sequence and the current token do not satisfy the constraint and the index of the current token does not exceed n we assign to the language model, we go on to analyze the next possible token produced at the same step.

$$\frac{<uv_{1:j-1},c,t_{i'},\sigma>\longmapsto<uv_{1:j},c,t_i,\sigma> \quad i=n+1 \quad j>1}{<uv_{1:j},c,t_i,\sigma>\longmapsto<uv_{1:j-1},c,t_{i'+1},\sigma>}$$

If all alternative tokens generated at a step except the first step do not pass the constraint checking, we get stuck at this step. To produce a sequence which can better satisfy the constraint, we need to go back to the previous step and evaluate the next possible token at the previous step.

$$\frac{<uv_{1:j},c,t_{i-1},\sigma>\Downarrow\textbf{true} \quad <uv_{1:j},c,t_i,\sigma>\Downarrow\textbf{false}}{<uv_{1:j},c,t_i,\sigma>\longmapsto<\text{Empty},\text{udecode}(u),\sigma[\text{Var}]=\text{decode}(u)>} \text{ where } \text{decode}(u)=v_{1:j}t_{i-1}$$

$$\frac{<uv_{1:j},c,t_{i-1},\sigma>\Downarrow\textbf{true} \quad i=n+1}{<uv_{1:j},c,t_i,\sigma>\longmapsto<\text{Empty},\text{udecode}(u),\sigma[\text{Var}]=\text{decode}(u)>} \text{ where } \text{decode}(u)=v_{1:j}t_{i-1}$$

These two rules explain that if all alternative tokens at the current step do not satisfy the constraint, but there is a token we have analyzed at the previous step satisfying the constraint, we stop the generation and return the history with this token. This result is the sequence which satisfies the constraint with the highest probability. Then, we assign it to the variable of the current scope.

$$\frac{i>n \quad j=1}{<uv_{1:j},c,t_i,\sigma>\longmapsto c \text{ fail}}$$

We define the failure **fail** for the constraint. When the language model can not generate a token at the first step satisfying the constraint c, we raise an exception and tell the user which constraint cannot be satisfied.

**Validating constraints:** For a sequence of tokens, we define the following rules to decide whether this sequence satisfies the current constraint.

$$\frac{}{<uv_{1:j},\text{type}(\text{Var}),t_i,\sigma>\Downarrow\textbf{true}} \text{ if } v_{1:j}t_i \text{ has the same } \textbf{type} \text{ with type}$$

$$\frac{}{<uv_{1:j},\text{type}(\text{Var}),t_i,\sigma>\Downarrow\textbf{false}} \text{ if } v_{1:j}t_i \text{ does not have the same } \textbf{type} \text{ with type}$$

$$\frac{}{<uv_{1:j},\text{equal}(\text{Var};e),t_i,\sigma>\Downarrow\textbf{true}} \text{ if } v_{1:j}t_i \text{ has the same prefix with e}$$

$$\frac{}{<uv_{1:j},\text{equal}(\text{Var};e),t_i,\sigma>\Downarrow\textbf{false}} \text{ if } v_{1:j}t_i \text{ does not have the same prefix with e}$$

$$\frac{}{<uv_{1:j},\text{Var in e},t_i,\sigma>\Downarrow\textbf{true}} \text{ if } v_{1:j}t_i \text{ matches prefix of element in set e or a part of string e}$$

$$\frac{}{<uv_{1:j},\text{Var in e},t_i,\sigma>\Downarrow\textbf{false}} \text{ if } v_{1:j}t_i \text{ does not match prefix of element in set e or a part of string e}$$

$$\frac{<uv_{1:j},c_1,t_i,\sigma>\Downarrow\textbf{true} \quad <uv_{1:j},c_2,t_i,\sigma>\Downarrow\textbf{true}}{<uv_{1:j},c_1 \text{ and } c_2,t_i,\sigma>\Downarrow\textbf{true}}$$

$$\frac{<uv_{1:j},c_1,t_i,\sigma>\Downarrow\textbf{false}}{<uv_{1:j},c_1 \text{ and } c_2,t_i,\sigma>\Downarrow\textbf{false}}$$

$$\frac{<uv_{1:j},c_1,t_i,\sigma>\Downarrow\textbf{true} \quad <uv_{1:j},c_2,t_i,\sigma>\Downarrow\textbf{false}}{<uv_{1:j},c_1 \text{ and } c_2,t_i,\sigma>\Downarrow\textbf{false}}$$

$$\frac{<uv_{1:j},c_1,t_i,\sigma>\Downarrow\textbf{true}}{<uv_{1:j},c_1 \text{ or } c_2,t_i,\sigma>\Downarrow\textbf{true}} \qquad \frac{<uv_{1:j},c_1,t_i,\sigma>\Downarrow\textbf{false} \quad <uv_{1:j},c_2,t_i,\sigma>\Downarrow\textbf{true}}{<uv_{1:j},c_1 \text{ or } c_2,t_i,\sigma>\Downarrow\textbf{true}}$$

$$\frac{<uv_{1:j},c_1,t_i,\sigma>\Downarrow\textbf{false} \quad <uv_{1:j},c_2,t_i,\sigma>\Downarrow\textbf{false}}{<uv_{1:j},c_1 \text{ or } c_2,t_i,\sigma>\Downarrow\textbf{false}}$$

$$\frac{<uv_{1:j},c,t_i,\sigma>\Downarrow\textbf{true}}{<uv_{1:j},\text{not } c,t_i,\sigma>\Downarrow\textbf{false}} \qquad \frac{<uv_{1:j},c,t_i,\sigma>\Downarrow\textbf{false}}{<uv_{1:j},\text{not } c,t_i,\sigma>\Downarrow\textbf{true}}$$

**Notice**: stop_at here is not a normal constraint, it is exactly a flow control. If the stop token is generated, we truncate the generation and return the history. If the stop token is not generated, we continue to generate tokens.

Thus, stop_at has its special rule:

$$\frac{<uv_{1:j},\text{stop\_at}(\text{Var};e),t_i,\sigma>\Downarrow\textbf{false} \quad t_i\neq<EOS>}{<uv_{1:j},\text{stop\_at}(\text{Var};e),t_i,\sigma>\longmapsto<uv_{1:j+1},\text{stop\_at}(\text{Var};e),t_1,\sigma>} \text{ where } v_{1:j+1}=v_{1:j}t_i$$

$$\frac{<uv_{1:j},\text{stop\_at}(\text{Var};e),t_i,\sigma>\Downarrow\textbf{true}}{<uv_{1:j},\text{stop\_at}(\text{Var};e),t_i,\sigma>\longmapsto<\text{Empty},\text{udecode}(u),\sigma[\text{Var}]=\text{decode}(u)>} \text{ where } \text{decode}(u)=v_{1:j}t_i$$

$$\frac{<uv_{1:j},\text{stop\_at}(\text{Var};e),t_i,\sigma>\Downarrow\textbf{false} \quad t_i=<EOS>}{<uv_{1:j},\text{stop\_at}(\text{Var};e),t_i,\sigma>\longmapsto<\text{Empty},\text{udecode}(u),\sigma[\text{Var}]=\text{decode}(u)>} \text{ where } \text{decode}(u)=v_{1:j}$$

$\frac{}{<uv_{1:j}, \text{stop\_at}(Var;e), t_i, \sigma> \Downarrow \textbf{true}}$ if the end of $v_{1:j}t_i$ matches e

$\frac{}{<uv_{1:j}, \text{stop\_at}(Var;e), t_i, \sigma> \Downarrow \textbf{false}}$ if the end of $v_{1:j}t_i$ does not match e

## 7 Reframe Prompting Framework

Next, we can discuss the syntax for defining a task in the language, which is how the model is queried with the advanced reframe prompting technique. The syntax for a task is shown below:

```
argmax / top_k = n
task
"query body"
output as [Var]
where constraint...
end
```

A query of our language has 6 components:

**argmax / top_k:** This is the decoding strategy we assign to the language model. Our language supports two kinds of strategies. If we use the argmax strategy, at each step, the language model only generates one token. If we use the top_k = n strategy, the language model will generate n tokens with the highest possibility at each step so that our language can choose one which satisfies the constraint.

**task:** A task marks the start of a query. It is required for a complete language model query.

**"query body":** "query body" is the main part of a query. It must be written with the double quotation markers. It contains the words we want to say to the language model. Users can also declare variables in the query body to capture the response from language models and retrieve variables' value from the current scope.

**output as:** This part is optional. If we write this clause, the final response of the language model will be assigned to a variable. Then, this variable will be passed to the scope of following queries so that following queries can make use of the result of the first query.

**where:** This clause is optional. With this clause, users can impose constraints on the variables of the query body. When executing the query body, our language tries to make each variable satisfy the constraint respectively. Constraints for each variable should be separated by semicolons.

**end:** end represents the end of a task. It is required for each query.

## 8 Code Implementation + Evaluation

We implement our language with python. For the query, we use the regular expression to parse it. We write three python files to implement the completeness checking, constraint checking and the execution function. Finally, we define an interface function so that users only need to call this function to execute their queries. Our language is lightweight and easy to use.

As mentioned in Section 2, the authors of LMQL proposed three standards to evaluate their language. These standards are expressiveness, performance and accuracy. Since they evaluated their language manually, it is very difficult to evaluate our language on the exact same metrics. Specifically, while they evaluated their language compared to traditional prompting in areas such as arithmetic and understanding dates, these datasets weren't readily available for our usage. Also, our language depends on the API of GPT. It is expensive to evaluate our language on the dataset with a large scale. Many open-source large language models have billions of parameters. We cannot run them on our laptops. Therefore, evaluating our language across large language models is also difficult.

We still try to evaluate the expressiveness our language and we find that it has reached our designing goals.

```
test21 = \
'''
argmax
task
"
1+1=
"
output as [variable1]
where int(variable1)
end

task
"
1+1={variable1}
Is it correct? [variable2]
Repeat: {variable2}
"
where str(variable2)
end
'''
```

```
1+1=
1+1=2

 Is it correct?

Yes, that is correct.  Repeat:

Yes, that is correct.
```

Here is an example of the multitask function of our language and its result. We define two tasks. The final result of the first task is output as a variable and then it will be used by the second tasks. We also impose constraints on these two variables. For the output of our language, the first line is the query body of the first task and the rest lines are the

query of the second task. In the first task, our language successfully captures the response of the language model and this response is also used in the second task. In the second task, we capture the response of language model and then retrieve its value from the current scope. Yes, that is correct appears twice. This shows that our language is well executed.

```
test12 = \
'''
argmax
task
"
Say this is a test: [variable1]
Repeat: {variable2}
"
end
'''
```

```
File D:\python_project\prompting language\complete
     36 variablename = re.search(r'\{[a-zA-Z0-9]+?'
     37 if scope.count(variablename) == 0:
---> 38     raise Exception('Not defined variable:

Exception: Not defined variable: variable2
```

For the completeness rule, we also design an experiment to test it. Here, our language tries to call a variable2 which is not defined by [variable2] before. Since users cannot call an undefined variable in our language, we throw an exception and point to the part which violates the completeness checking rule.

```
test32 = \
'''
argmax
task
"
1+1=[variable1]
"
where str(variable1)
end
'''
```

```
Exception: The language model cannot produce a sequence that satisfies the constraint
```

Our language tries to let the response of language model satisfy the constraint. If the language model cannot generate a sequence which can pass the constraint checking, we throw an exception and point to this unsatisfied constraint. In this example, we try to make the response of language mode to be a string. Since only one token is generated and it is 2, it does not satisfy the constraint. Our language throws an exception and informs the user.

```
test4 = \
'''
top_k = 3
task
"
1+1=[variable1]
"
where int(variable1) and equal(variable1, 1)
end
'''
```

```
query_execution(test4)
```

```
1+1=1
```

By making use of the top_k decoding strategy and constraints, users can control the result of our language to some degree. Here we use the top_k = 3 decoding strategy. That means the language model generates three alternative tokens at each step. In this query, the language model generates 1, 2 and 3 for the variable1. Since we also have the constraint equal(variable1, 1), only the 1 can pass the constraint. Therefore, our language returns 1+1=1.

These tests demonstrate the expressiveness and effectiveness of our language. Our language can check the completeness of query before execution and impose constraints on the response of language models during the execution. It makes the interaction between users and language models easier.

## 9 Limitation and Future work

There are still several limitations in our language. In the future work, we want to fix these limitations and expand functions of our language.

Firstly, as we mention above, we parse the query with regular expressions. In some case, due to the limitation of regular expressions, we may lose the original format of the query body. It sometimes makes the result of a query strange and hard to read. We do not want the input from users to be destroyed during the parsing. In the future, we will try to optimize our regular expressions or adopt other methods so that the original format will not be changed during the execution.

Secondly, our language generates response and checks the constraint token by token. It works for the open-source models in the huggingface database, since many models here have the generate() function so that we can analyze generated tokens at each step. However, it still reduces the speed of our language, since we need to call the large language model at each step. Also, for GPT, it does not have the generate() function. The response of GPT is always a sequence. Thus, we need to limit the number of generated tokens so that we can execute our checking rules. However, because the pricing of GPT is counted by the number of input and output tokens, it will be very expensive if we call the APIs of GPT at each step. In order to save time and money, we need to evaluate constraints on the sequence level. To realize this, we need to come up with more semantic rules in the future and reconstruct our codes to implement these rules.

Additionally, there is a limitation on the function of our language. Our language now only supports a small size of constraints. Even for constraints of our language such as type(Var) and Var in Pythonexp, only some Python datatypes

are supported. To make our language more applicable and flexible, we need to support more constraints in our language.

In regards to more future work that would be beneficial to our language, choosing some datasets to perform quantitative evaluation on compared to traditional prompting would be important so we could compare our implemented method of prompting to traditional prompting. While we would need to find a dataset financially viable to not have excessive query costs, having this quantitative evaluation is important for the demonstrated benefits of our language.

Although this would be difficult to implement, an additional language feature that could be used to reduce query costs would be constraints that get incorporated into the prompt itself. The benefit of this is that the output of the model decoding is more likely to be in accordance with what the user wants on earlier decoding iterations. Instead of just constraining the output of the decoding, having the model understand the constraint could result in less backtracking and better quality results. Currently, our language leaves the query body to be entirely specified by the user, which can result in lower quality results in general if the prompt is not well-formed to the task at hand.

Finally, there are additional reframe prompting techniques that could be beneficial to introduce into the language, as each technique performs better on certain tasks. Given that these reframe prompting techniques can improve performance over traditional prompting methods, having a language that allows multiple methods of advanced prompting would be beneficial for providing the user with a model interface that is applicable in a wide range of scenarios. While we feel that task decomposition was the most important of these techniques, having the added flexibility could improve the user experience interfacing with our language.

## 10 Conclusion

In this work, we identified and addressed the problems of the LMQL language and formalized the execution of a language model query. We introduced our own language model query language and implemented basic completeness checking and constraint checking functions. With our language, users can easily do chain of thought and impose constraints on the response of language models. As the prevalence and widespread usage of large language models increases over time, having superior methods to interacting with these models will ultimately provide for a better user experience. Our language fundamentally depends on decomposing the query body and calling the language model. It works as an abstracted interface between users and language models. The method of our language can potentially be applied to the internal decoding of language models in the future. As a result, it will make the response of language models more proficient and satisfactory than compared with traditional decoding methods.

## References

[1] Ning Ding, Shengding Hu, Weilin Zhao, Yulin Chen, Zhiyuan Liu, Hai-Tao Zheng, and Maosong Sun. 2021. OpenPrompt: An Open-source Framework for Prompt-learning. https://doi.org/10.48550/arXiv.2111.01998

[2] Microsoft Semantic Kernel https://learn.microsoft.com/en-us/semantic-kernel/overview/

[3] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. Proc. ACM Program. Lang. 7, PLDI, Article 186 (June 2023), 24 pages. https://doi.org/10.1145/3591300

[4] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. https://doi.org/10.48550/arXiv.2201.11903

[5] Swaroop Mishra, Daniel Khashabi, Chitta Baral, Yejin Choi, and Hannaneh Hajishirzi. 2022. Reframing Instructional Prompts to GPTk's Language.

[6] Justin Reppert, Ben Rachbach, Charlie George, Luke Stebbing, Jungwon Byun, Maggie Appleton, and Andreas Stuhlmüller. 2023. Iterated Decomposition: Improving Science Q&A by Supervising Reasoning Processes. https://doi.org/10.48550/arXiv.2301.01751