Lexeme Level Verification for Language Model Guided Program Synthesis

First Author[1] , **Second Author**[2] , **Third Author**[2,3] , **Fourth Author**[4]
[1]First Affiliation
[2]Second Affiliation
[3]Third Affiliation
[4]Fourth Affiliation
{first, second}@example.com, third@other.example.com, fourth@example.com

Abstract

Large Language Models (LLMs) exhibit strong performance for program synthesis tasks including text-to-SQL, up to 86.6% accuracy on the Spider 1.0 benchmark, but purely neural techniques remain prone to hallucinations. Existing approaches to improve accuracy generally rely on post-hoc verification, that fails to constrain unproductive path exploration, or constrained decoding techniques that rely on learned distributions leaning towards correct programs. In this work, we study the mistakes LLMs make when synthesizing SQL queries. We observe that the mistakes can be grouped into **¡x¿** error categories, **¡y¿** of which are amenable to symbolic fixes. Importantly, we find that *constrained decoding alone is insufficient* to address **¡z¿** of these categories. Our findings provide interpretable insight into where LLMs fail and suggest promising, neuro-symbolic directions for improving LLM guided program synthesis.

1 Introduction

[*Symbolic techniques could be used to address LLM's weakness for program synthesis tasks: LLMs make good guesses on average, but lack correctness guarantees.*] The advent of Large Language Models (LLMs) has introduced a novel way of tackling program synthesis as they are increasingly used as program generators to synthesize working code. In contrast to other program synthesis techniques, LLMs are scalable to larger and more complex programs; however, they struggle with self-correction and can get stuck down an incorrect path that, ultimately, will not yield correct code. Conventional program synthesis techniques use symbolic reasoning to produce correct code but are limited in scalability. We hypothesize that an integration of insights and techniques from conventional symbolic reasoning techniques can be applied at inference time to guide LLMs to more promising search spaces.

To evaluate the validity of our hypothesis, we focus our efforts on tackling a subspace of the program synthesis domain, the text-to-SQL task, as it is easier to verify and validate a query's correctness. The text-to-SQL task is concerned with generating valid SQL queries that correctly answers a natural language question for a given schema or workflow. For example, for the question ”How many clubs are there?” for a given schema the corresponding gold SQL query is ”SELECT count(*) FROM club” (Yu et al. 2018). We hypothesize that the insights gained from this work will be further applicable in broader code generation tasks.

Our contributions are as follows:

FIXME motivate contributions in the introduction, include problem statement FIXME variants of TSQL and you finetune models based on these and does it make a difference FIXME error categorization for lexemes not in top-k FIXME in theory if we used TSQL instead of SQL then we could argue more symbolic techniques to fix TSQL that would be harder to fix for SQL

- We present a lexeme level verification framework that incrementally corrects the LLM and guides decoding towards more productive generation paths.
- We show that constrained decoding alone does not reduce LLM search space enough to meaningfully improve program synthesis for realistic SQL queries.
- We present empirical evidence supporting the integration of symbolic techniques to aid program synthesis, suggesting that the combination of techniques could improve LLM-based synthesis.
- We give one way to categorize mistakes the LLM makes.
- We show that ¡x¿ mistake categories are amenable to a handful of symbolic repair techniques.
- We show that the combination of constrained decoding and symbolic repair techniques could help fix the majority of mistakes an LLM makes

2 Related Work

Prior research has explored SQL synthesis using classic approaches, such as enumerative search, to produce correct queries (Wang, Cheung, and Bodik 2017; Takenouchi et al. 2021). These works focus on pruning the search space either through additional information provided by the user, such as the constants that are allowed to appear in a query (Wang, Cheung, and Bodik 2017; Takenouchi et al. 2021), or through pruning unproductive paths (Wang, Cheung, and Bodik 2017; Takenouchi et al. 2021). They often include soundness and bounded completeness guarantees, but struggle to scale to larger, more complex queries.

Other research instead foregoes guarantees for empirical results. Many recent works focus on leveraging Large Language Models (LLMs) to produce SQL queries (Gao et al.

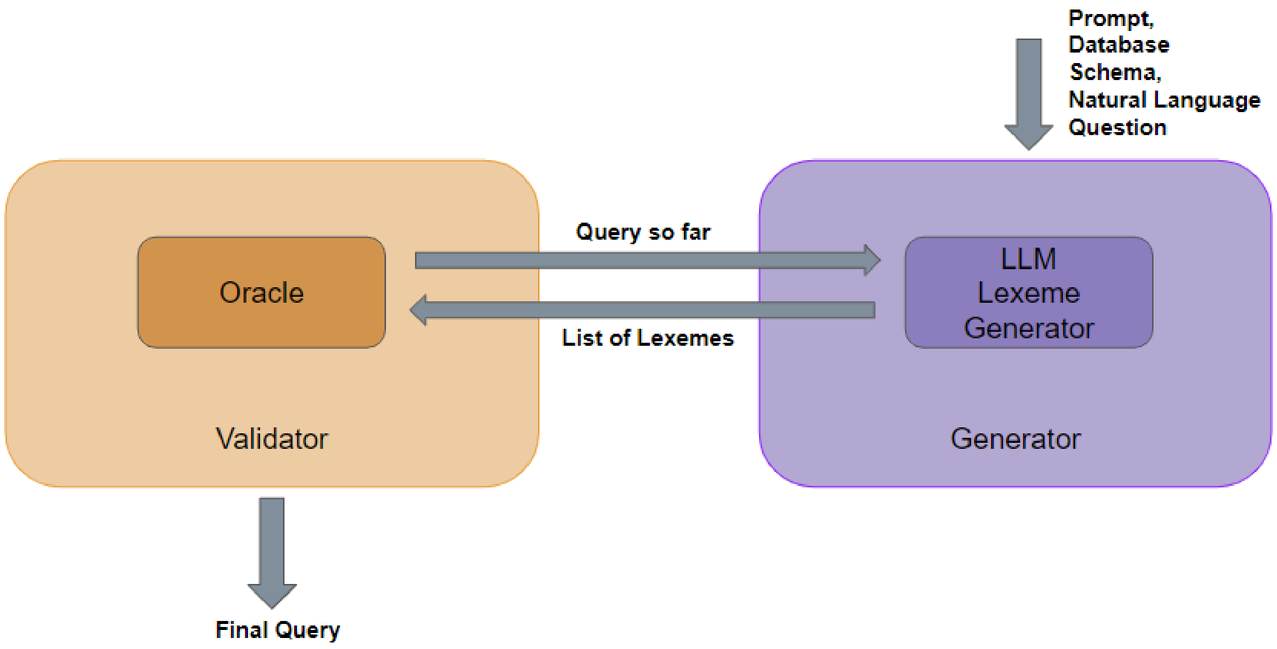

Figure 1: The overall system architecture. The Language Model Generator is prompted to generate a TSQL query given a database schema and natural language question. The generator returns a list of lexemes to the Validator which incrementally validates the most probable lexeme against the gold one and corrects the generator if there is a mismatch.

2024). They have performed well empirically, reaching test suite accuracies of up to 86.6% (Gao et al. 2024) against the Spider SQL dataset (Yu et al. 2018). FIXME MiniSeek 91.2 but no reference

Some prior works build off the strengths of both approaches, using probabilistic models to guide search while verifying the correctness of produced candidates. These prior works either require white-box access to probabilistic models (Lee et al. 2018; Menon et al. 2013), or treat models as program candidate oracles (Jha et al. 2023), or apply a hybrid approach that combines both methods (Li, Parsert, and Polgreen 2024). However, all these works use probabilistic models to generate full candidate programs before evaluating them. This leaves a large search space for the probabilistic model to categorize.

In contrast, we observe that LLMs are inherently next token predictors, rather than full sequence predictors. We leverage this insight to guide LLM search using lexeme level guidance which provides early guidance for the LLM to avoid unproductive search paths.

With regards to ChopChop, A. our insights are complimentary to ChopChop B. they observed that once an LLM made a mistake...it was too late to go back and fix it....”The type system implemented in our semantic pruner will prevent the assignment to divisors, but it will still allow line 2.” so LLM commit to its solution so it is useful to mitigate these types of errors before it is too late C. constrained decoding is known to skew the underlying distribution D. The pruners in ChopChop seem a bit unrealistic to create but I guess this is theoretical So ChopChop enforces semantic validity when generating queries...in theory...but this requires you to create pruners...and you still run into the issue of an LLM makes a mistake and you reach a point where its too late to correct it...so other symbolic fixes? I mean you could still use this...ChopChop still wouldn’t work because the candidates aren’t in top-k...you would need to have the model learn some more patterns/ clause constraints

3 Our Approach

Our overall system is shown in Figure 1.

3.1 Language Model (LM) Generator

Consider $\mathcal{L}$ to be an (autoregressive) Language Model created for a sequence generation task that has a predefined vocabulary Σ, consisting of some k tokens which are used as part of the tokenization. At each timestep $i \in [1, n]$, the model generates a sequence of these tokens, $(t_1, t_2, \ldots, t_n)$ based on an autoregressive decoding process where the token with the highest probability is selected. This is usually part of a heuristic or sampling based search procedure on the set of k tokens to predict the next most probable token. Now, consider the LM to be finetuned on a new DSL and the tokens in the vocabulary are not changed. This means that the LM is now capable of producing the probability distribution over the same tokens but conforming to the changed ordering of our DSL *i.e.* if an original ordering of tokens $w \in \Sigma$ exists, then a new ordering of tokens $w^* \in \Sigma$ should also be produced by the LM. A $\mathcal{L}$ generates tokens and concatenates them by taking the token with the highest probability and generates a new token and continues until a whitespace token is generated or a max heap size is reached.

Example 3.1. *Consider the original token sequence $w =$ SELECT, *, FROM, books, drawn from SQL queries present in the model's training data. Using our DSL, this sequence is transformed into $w^* =$ PROJECT, *, FROM, books. In this transformation, the only substituted token is PROJECT, and we therefore expect the model's vocabulary to assign a non-zero probability p to this token.*

For commonly occurring words, we can fully expect the new word to also exist in the vocabulary, however, if the word does not exist we know that some n-gram combination of the word does exist. When for example the token, REGENRATE $\notin \Sigma$ then we can form the same word by some combination like $\{$RE, GEN, E, RATE$\} \in \Sigma$. In our approach, we use QWEN-7B (Bai et al. 2023), which uses an open-source fast Byte Pair Encoding (BPE) for tokenization and len$(\Sigma) \in 151642$.

Since we use the language model as an autoregressive next token predictor, we would also like to harness the power of the probabilistic nature in which the tokens are ranked. Usually, since the top-1 token is taken as the next prediction, there is an uncertainty that whether this next token is part of the valid query that satisfies the question that is requested by the user. Hence, we would also like to store, some top-m tokens at each time step and perform a beam search when on these token sets when there is a mismatch in the token requested and the query validated.

Language Model Validator. With each generated lexeme, we validate the generated lexeme by evaluating it against the expected lexeme of the gold query. If the most probable lexeme does not match the gold lexeme, we consider it to be a mistake. We compare the generated lexeme with the gold lexeme while considering issues that arise with aliasing for column names by only considering the column name. For example $T1.BehaviorMonitoring$ and $BehaviorMonitoring$ would both be considered correct lexemes compared to the gold lexeme $BehaviorMonitoring$. If the top most probable lexeme does not match the gold lexeme, we guide the Language Model by correcting it with the gold lexeme and continue generating. We incrementally validate each generated lexeme to guide the Language Model towards more productive generation paths rather than allow it to fully generate a query before evaluating its correctness.

Our system architecture is outlined in Algorithm 1.

4 Experimental Setup

4.1 Implementation

The system is implemented in Python using a Supervised Finetuned QWEN-7B model as the Language Model generator with $k = 10$ beam widths. It is finetuned on the Spider 1.0 dataset. During fine-tuning, we transform each question in the Spider dataset into a natural language prompt and translate each target query to our DSL (TSQL). Specific prompting details are provided in . We consider a lexeme to be complete if the next generated token is whitespace.

Algorithm 1 System Algorithm

```
1: Require: Language Model $\mathcal{L}$, beam width $m$, gold query $g$ and some string prompt $\mathcal{P}$, number of mistakes $n$
2: procedure LanguageModelValidator($\mathcal{P}, w, g, n$)
3:     if $g == w$ returns TRUE then
4:         $\mathcal{P} \leftarrow \mathcal{P} \,||\, w$
5:         return $\mathcal{P}, n$
6:     else
7:         $\mathcal{P} \leftarrow \mathcal{P} \,||\, g$
8:         $n \leftarrow n + 1$
9:         return $\mathcal{P}, n$
10:    end if
11: end procedure
12: procedure Orchestrator()
13:     $n \leftarrow 0$
14:     for $i = 1, 2, \ldots, \text{len}(\text{gold\_query})$ do
15:         $w_i \leftarrow \mathcal{L}(\mathcal{P})$
16:         $g_i \leftarrow$ GoldLexeme($g, i$)
17:         $\mathcal{P}, n \leftarrow$ LanguageModelValidator($\mathcal{P}, w_i, g_i, n$)
18:     end for
19:     return $n$
20: end procedure
```

Total Number of Queries	30
Total Number of Mistakes	200
Gold Lexeme Found in Top-k	77
Gold Lexeme Not Found in Top-k	123
Total Not Found errors	117
Gold was SQL keyword	16
Gold was NOT SQL keyword	101

Table 1: This table shows the empirical evaluation results for SQL generation. Mostly schema and aggregation errors. For the fine-tuned model eglym/DR-TEXT2SQL-CodeLlama2-7B

4.2 Results

Our empirical evaluation results are presented in Table 2. Across 30 queries taken from the test split of the Spider 1.0 database, the LLM generated lexemes resulting in 112 mistakes. Out of these mistakes, 70 were found in the top-k predictions. This indicates that the LLM was in the right search space and conveys the need for the integration for constraint based decoding or beam search guidance. This result also shows the promise for LLMs to be used in program synthesis as 62.5% of the time, the LLM generates the correct lexeme and is the right program space. The integration of symbolic techniques could provide insight to the LLM to re-rank and re-prioritize the correct lexemes.

The 42 mistakenly generated lexemes that were not found in the top-k indicate the gap between the LLMs understanding and the correct search space. The integration of simple purely symbolic fixes would be most useful in guiding the program synthesis by providing formal verification and corrections.

SFT model - XiYanSQL-QwenCoder-7B-2504

Total Number of Queries	30
Total Number of Mistakes	276
Total Not Found errors	234
Gold was SQL keyword	51
Gold was NOT SQL keyword	183

Table 2: Results for Qwen2.5-Coder-7B

5 Taxonomy

5.1 Schema based errors

Wrong Naming these are errors like wrong column or table name (the gold is present in the schema but the predicted candidate is not)

Symbolic fix: Easy- if valid schema element in top-k then you use a schema based constrained decoding to select (usually the gold is the top most). If there is no valid schema element, have the LLM pick from or rank valid schema elements

Wrong Values For where conditions - the wrong value is selected for example - "What are the addresses, towns, and county information for all customers who live in the United States?" but the gold query is

```
SELECT $address_line_1$ , $town_city$ ,
county FROM Customers WHERE Country = 'USA
```

The LLM predicts **'United** which makes sense given the phrasing of the question but involves parsing out all the values of the schema to "infer" that "United States" maps to "USA"

Symbolic Fix: Medium - We may be able to infer the correct condition but we need a way to map that question to the actual values in the database

5.2 Structural based errors

These are elements that are a violation of the expected SQL structure (ie clause based errors).

LIMIT errors The LLM doesn't predict a LIMIT at the end of a full query.

Symbolic Fix: Easy - we basically just look at the output and if it matches except the number of rows we just add a LIMIT

Everything else

5.3 Where Conditions

Wrong operator wrong operator (sometimes the LLM will correctly predict bigger/ greater but not >

Symbolic fix: easy for wrong operator - if LLM predicts bigger then just replace it with >

upon further inspection there aren't many of these that can be easily fixable (only about 2 were bigger/greater instead of >) but I'll leave this category in for now

5.4 Aggregate Errors

Wrong aggregate functions like COUNT, MAX, etc

Symbolic fix: hard

5.5 Other

catch all

includes syntactically/ semantically "similar" ie they look right....but aren't

Symbolic fix: hard

6 Conclusion

LLMs are being increasingly used for program synthesis but solely relying on their probabilistic token ranking can lead to incorrect programs. In this work, we investigate these limitations and propose a lexeme level validation methodology to guide the LLM towards semantically valid queries. We find that many errors can be corrected by using a constraint decoding and beam-search guided reasoning technique to rerank the generated lexemes. For errors that cannot be corrected using these methods, purely symbolic techniques can be used to aid program synthesis.

References

Bai, J.; Bai, S.; Chu, Y.; Cui, Z.; Dang, K.; Deng, X.; Fan, Y.; Ge, W.; Han, Y.; Huang, F.; Hui, B.; Ji, L.; Li, M.; Lin, J.; Lin, R.; Liu, D.; Liu, G.; Lu, C.; Lu, K.; Ma, J.; Men, R.; Ren, X.; Ren, X.; Tan, C.; Tan, S.; Tu, J.; Wang, P.; Wang, S.; Wang, W.; Wu, S.; Xu, B.; Xu, J.; Yang, A.; Yang, H.; Yang, J.; Yang, S.; Yao, Y.; Yu, B.; Yuan, H.; Yuan, Z.; Zhang, J.; Zhang, X.; Zhang, Y.; Zhang, Z.; Zhou, C.; Zhou, J.; Zhou, X.; and Zhu, T. 2023. Qwen technical report.

Gao, D.; Wang, H.; Li, Y.; Sun, X.; Qian, Y.; Ding, B.; and Zhou, J. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. *Proc. VLDB Endow.* 17(5):1132–1145.

Jha, S. K.; Jha, S.; Lincoln, P.; Bastian, N. D.; Velasquez, A.; Ewetz, R.; and Neema, S. 2023. Counterexample guided inductive synthesis using large language models and satisfiability solving. In *MILCOM 2023 - 2023 IEEE Military Communications Conference (MILCOM)*, 944–949.

Lee, W.; Heo, K.; Alur, R.; and Naik, M. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, 436–449. New York, NY, USA: Association for Computing Machinery.

Li, Y.; Parsert, J.; and Polgreen, E. 2024. Guiding enumerative program synthesis with large language models. In Gurfinkel, A., and Ganesh, V., eds., *Computer Aided Verification*, 280–301. Cham: Springer Nature Switzerland.

Menon, A. K.; Tamuz, O.; Gulwani, S.; Lampson, B.; and Kalai, A. T. 2013. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, I–187–I–195. JMLR.org.

Takenouchi, K.; Ishio, T.; Okada, J.; and Sakata, Y. 2021. Patsql: efficient synthesis of sql queries from example tables with quick inference of projected columns. *Proc. VLDB Endow.* 14(11):1937–1949.

Type of Error	Rough Distribution	Suggested Symbolic Fix
Schema Mismatch (also join on wrong column errors could also go here)	50-70% of errors	top-k: schema based decoding usually the highest ranked schema element is gold not in top-k: have the LLM rank schema elements
Structural	10-20%	easy: end of a query determine if you need a LIMIT medium: hard: catch all
WHERE		
Aggregates		
Other		

Table 3: Error Types and suggested symbolic fixes

	XGenerationLab/ XiYanSQL-QwenCoder-14B-2504	XGenerationLab/ XiYanSQL-QwenCoder-7B-2504	eglym/ DR-TEXT2SQL-CodeLlama2-7B
Schema (includes join)	214	211	206
Structural	31	45	32
Where (note these numbers are inflated)	22	24	19
Aggregate	20	21	11
Other	30	17	17
Total Errors	318	318	285

Table 4: Different SFT models and general error distributions over 51 queries.

Wang, C.; Cheung, A.; and Bodik, R. 2017. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, 452–466. New York, NY, USA: Association for Computing Machinery.

Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; Zhang, Z.; and Radev, D. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Riloff, E.; Chiang, D.; Hockenmaier, J.; and Tsujii, J., eds., *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 3911–3921. Brussels, Belgium: Association for Computational Linguistics.