IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 9 October 2023, accepted 25 October 2023, date of publication 1 November 2023, date of current version 8 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3329071

RESEARCH ARTICLE

Enabling Generative AI to Produce SQL Statements: A Framework for the Auto-Generation of Knowledge Based on EBNF Context-Free Grammars

CHRISTOPHER TROY, SEAN STURLEY, (Member, IEEE), JOSE M. ALCARAZ-CALERO, (Senior Member, IEEE), AND QI WANG
School of Computing, Engineering and Physical Sciences, University of the West of Scotland, PA1 2BE Paisley, U.K.

Corresponding author: Christopher Troy (christopher.troy@uws.ac.uk)

This work was supported in part by the Carnegie Trust for the Universities of Scotland under Grant PHD010695; and in part by the European Commission through Autonomous Trust, Security and Privacy Management Framework for IoT (ARCADIAN-IoT) under Grant H2020-SU-DS-2018-2019-2020/101020259.

ABSTRACT The motivation of this paper is to be able to generate high-quality (Structured Query Language) SQL language sentences in terms of syntax and semantics so that they are intended to achieve a concrete predefined and well-known aim. For example, generating SQL sentences that are capable of detecting a cyber-attack from a set of metrics available in a database table. Two solutions are needed to achieve so, a tool that enables and performs the syntactically valid generation of SQL sentences and an (Artificial intelligence) AI algorithm able to guide the semantics of such generations to the achievement of the best sentences for the intended purpose. The main contribution of this manuscript is the first of these solutions. To be concrete, this paper proposes a tool to enable and generate syntactic-valid language sentences. The tool can deal with any language defined as an ANTLR4 EBNF (Extended Backus-Naur Form) grammar. The paper also provides a methodology to help achieve an EBNF grammar suitable for addressing concerns related to ambiguity and recursion as a direct result of the generation process. The paper further implements a prototype utilizing ANTLR4's recognizer and its Augmented Transition Network for language generation using EBNF grammars. In-depth design and logic implementation are provided, showcasing areas of interest for AI integration. The achieved prototype showed an ability to easily generate syntactically valid SQL statements at various depths, with observable problems becoming more apparent during the exponential recursive growth. Our mitigation controls for such scenarios proved to be successful and were able to complete the recursion whilst also moving the push-down automata forward until query completion. Experimental validation was performed against a SQL EBNF grammar feeding the generated SQL statement into an SQL parser to validate the syntax.

INDEX TERMS ANTLR4, ATN, automata, EBNF grammar, generative AI, parser generator, SQL.

I. INTRODUCTION

Generative Artificial Intelligence (AI) is the usage of AI to generate new content, i.e. data, information, and knowledge, for specific domains. Some good examples of generative AI for written text are OpenAI's GPT4 [1], ChatGPT [2], and Google's BARD [3]. For art generation, there are DALL-E [4], and Stable-Diffusion [5], to name a few. Also, SQL (Structured Query Language) is the most widely adopted language for structuring and manipulating data, information, and knowledge around the world [6].

SQL plays a critical role in data structuring making tasks such as data analysis, data modification, and storage possible. The motivation behind our primary research is twofold. On the one hand, there is an objective for generating high-quality SQL language sentences that are intended to achieve a concrete

The associate editor coordinating the review of this manuscript and approving it for publication was Pasquale De Meo.

© 2023 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

predefined, and well-known aim. For example, a sentence that is intended to detect distributed denial of service attacks by making use of database tables with data related to measured metrics related to the communications and resources of the host being analyzed. This aim required the addressing of two different solutions: a tool capable of generating syntactically valid SQL sentences and an AI algorithm capable of driving the generation of such queries to the best possible quality in the semantics of the sentence to achieve the intended aim. This manuscript is focused on the first of these two solutions. Furthermore, the current literature and its sparsity highlight evident gaps surrounding tools and techniques to achieve the automatic generation of sentences without any initial input text. It is worth differentiating between the ample availability of language parser tools intended for processing an initial sentence and the scarcity of available tools for the automatic generation of sentences without an initial input for transformation. The state of the art is even more scarce when searching for automatic generation of sentences without said initial input that are capable of being guided by AI algorithms which is the main contribution of this manuscript. The approach taken to achieve the generation of SQL sentences is the usage of automata. Automata are created from a given grammar which constitutes all the rules of the language. There are plenty of well-established tools to create such automata for the purpose of parsing a sentence in a given language. They are called parser generators which accept an EBNF (Extended Backus-Naur Form) [7] grammar and produce both parser and lexer for validating the input stream using syntax analysis aided by parse tree traversal. However, there is a significant lack of tools available to perform the creation of the automata for the purpose of generating sentences in a given language based on its grammar which is our intended aim.

Within the field of computational linguistics, many formalisms exist for processing natural language. Much of the underpinning theory associated with the aforementioned automata is heavily dependent on the language and its complexity. For simpler language parsing tasks, DFA (Deterministic finite automata) [8] are utilized which is optimal for handling regular languages. A DFA can provide basic pattern matching through regular expressions. However, for complex languages which have nested structures, such complexity requires a more capable formalism. One such implementation for dealing with complex language can be through the implementation of an RTN (Recursive Transition Network) [9], which is a graph theoretical diagram that aids in processing language containing recursive structures. However, our work is influenced through the usage of ANTLR4 [10]. It is a parser generator that makes use of a similar implementation, called an ATN (Augmented Transition Network). An ATN is a non-deterministic push-down automaton that represents the relationships surrounding the syntax and semantics of a language through its states, rules, and transitions. All rules defined within a CFG (Context-Free Grammar) are analyzed by ANTLR4 providing an ATN graph for each production rule. Furthermore, the augmented capabilities of an ATN allow for the handling of recursive structures, just like an RTN. An ANTLR4-based ATN can implement semantic actions to reduce ambiguity through mitigations such as parse tree pruning. To summarise it all, the ATN builds upon the previous theoretical model of the RTN through the usage of augmented features, which provide features very much aligned with the overall aim of this research.

The main problem surrounding ANTLR4's ATN is by way of its intended usage. It is primarily used to aid the parsing algorithm when parsing input using predictive logic which improves its traversal and decision- making efficiency. In contrast with our proposal where no initial input is required, it has motivated our research with an aim to design and develop a prototype using an alternative approach to ATN utilization to enable the generation of sentences of any grammar defined by ANTLR4's EBNF, and at the same time to allow AI to influence the decision-making process whilst traversing the graph representation of the CFG production rules. This will help us find elements of commonality shared between both humans and machines using the SQL language as the expresser. Additionally, it provides a positive leverage where data does not need to be processed outside of the database relating to the associated computing network and storage requirements.

A prototype with these capabilities would yield many advantages for conducting different experiments through use-case scenarios based on data manipulation and knowledge discovery. For example, the generation of novel metrics to detect patterns of network attacks in IoT networks.

Our research aims to describe the design of a methodology and a pragmatic framework to promote the integration of an intermediary prototype using push- down Automatas based on ANTLR4's Augmented Transitional Network (ATN) alongside EBNF grammars.

This will provide state-based graph generations constituting syntactically valid SQL sentences using randomized decision selection as the temporary medium driving decision choice.

The way in which SQL and many other languages are represented within ANTLR4's ATN makes the manipulation of its traversal process possible. This observation highlights opportunities and ways to hypothesize and test solutions such as the initial randomized or future implemented weight-based state decisions for generating syntactically correct SQL. The central principle is to provide a novel prototype that can enable generative SQL, with gained insights highlighting ways in which AI can be correctly interfaced. This research can be explored for a wide range of use cases in ICT (Information and Communication Technology) and vertical businesses, where automatically generated SQL statements could be utilized for database optimisation and or knowledge discovery revealing novel insights into vast ranges of sparse data distributions surpassing human capabilities for recognition and extrapolation. For instance, the proposal provided in this paper is intending to conduct future research by utilizing these insights to implement a cybersecurity use case scenario by generating novel SQL metrics to detect cyber-attacks within IoT (Internet of Things) networks.

The following are the set of innovations beyond the state of the art provided in this contribution:

- A generic methodology to aid in the generation of sentences using any language expressed in EBNF format.
- The design and implementation details of the framework for the generation of datasets of syntactically valid SQL queries from zero input.
- The syntactic validation of the generated SQL queries as a way to highlight its use case.

The rest of this manuscript is laid out as follows. Section II describes the state-of-the-art tools and on- ongoing research related to the generation of SQL sentences. Section III will discuss the proposed methodology which will highlight ways in which EBNF Grammars can be modified or bespoke to aid in language generation. Section IV provides insights into the design of the proposed generator through the implementation of ANTLR4 and both ATN components belonging to its recognizer. Section V will provide the results from the proposed prototype, and section VI will provide our concluding remarks.

II. RELATED WORK

Table 1 provides a detailed analysis of recent and past research within the fields of automatic language sentence generation and generative AI, including the combination of them both. The table also contains our work so that the reader can analyze our contribution against the state-of-the-art. The following subsections group the analyses of the state of the art by the approach used to achieve the generation of the sentences.

A. SQL SENTENCE GENERATION

A study by Sugandhika et al. [11] adopted a strategy by using the English language as text input for SQL generation. This method requires human interaction so that an interpretation of the expected query can be generated using a heuristic-based approach. These heuristics aid in grammar identification in order to perform syntax correction, thus adapting the system dynamically to understand the user input. Our research differs with a notable advantage as the authors do not provide support for SQL nesting capabilities. Such capabilities supported in the SQL language allow for more complex query variations. Similarly, Datachat [12] allows the generation of SQL sentences for data analytics using English language sentences. A deeper understanding of results is facilitated by providing automatically generated English explanations of how they were derived.

Another SQL sentence generation approach has been proposed by authors Anisyah et al. [13] which made use of syntax trees. This approach performed an analysis of the syntax tree to identify notable SQL objects. These will then be arranged into SQL queries. We found this approach to be very interesting, and it is one of the benefits a parser provides when building a parse tree for traversal purposes. However, this approach also requires input and is thus not suitable when such input is non-existent.

B. GRAMMAR ASSISTED SENTENCE GENERATION

These tools make use of grammar to give the process of generating sentences using the language rules defined in such grammar. The way to describe the syntactic rules composing a grammar can be defined using different notations. A well-known notation for this purpose is the meta-syntax notation used in EBNF grammars.

One advantage of having a tool capable of generating language sentences based on EBNF grammar is the plethora of available grammar publicly accessible for use. The end results of implementing an EBNF-based generator lie within the ability to generate sentences in multiple languages, thus improving re-usability.

A proposal closely aligned with our on-ongoing research has been proposed by Sargsyan et al. [14] who demonstrated the capabilities of utilizing ANTLR4's automata for code generation of over 120 languages (EBNF grammars). Their proposed prototype uses random selection to determine what paths to traverse across the generation process. The authors Sargsyan et al. improved upon this [15] and were able to implement a weight-based system to help control the traversal process. One advantage of our contribution over Sargsyan et al. [15] is our proposed implementation to deal with recursive mitigation controls.

Additional benefits can arise when utilizing context-free grammar for generating language. One example highlighted by authors Palmas et al. [16] discussed ways of improving dialogue in video games through the generative capabilities that context-free grammars can express alongside sentiment analysis. The authors utilized a tool very much in a proof of concept stage [17]. This allowed the authors to create a context-free grammar to generate language rather than parse one. This would provide ways to generate new and interactive dialogues dependent on user participant reactions to the NPC's (Non Playable Character) actions with sentiment analysis providing better context to ascertain the players' tone. This helps highlight the generative potential CFGs have, so long as the tool developed to provide the generative language is designed correctly.

Pires et al. [18] explored a conceptual implementation for code generation using a KDM [19] (Knowledge Discovery Meta-Model) with xUnit [20] software test cases. The authors conceptualised using this meta-data to interface with xUnit which could then generate test-cases. This proposal highlighted XML as a way of expressing a language's structure rather than an EBNF grammar.

C. FUZZER-BASED SENTENCE GENERATION

Wang et al. [21] explored SQL generation for DBMS (Database management Systems) using automata for security testing. They looked beyond syntax generation by including interaction from the response received from the server using a multi-phased fuzzer approach to help manipulate the generation process.

What is interesting about this proposal relates to the interactive behavior that can directly influence the automata.

Rather than the automata being used for constructing the query, they have opted for using it to perform behavior analyses which alters the patterns of the test instances. Although this is an older study, it is intriguing to see ways in which automata can be used for aiding in language generation, rather than being the main actor to generate said language.

D. AI-DRIVEN SENTENCE GENERATION

Parikh et al. [22] used a custom dataset with text consisting of English questions, e.g. "How to select all people in the finance department?". The aim is to create the SQL syntax that performs the purpose indicated in the English language. To achieve so, they make use of the Seq2Seq AI algorithm achieving an accuracy of 69 percent for data retrieval when used by staff with very little to no SQL knowledge. Notice that our intention is to be able to generate SQL without any input thus this approach would not fit for the purpose.

There are several ChatGPT-based [23] research focused on the usage of generative AI for modeling tasks [24] and for modeling software use cases based on system requirements [25], to name a few. They all share the same idea of using a question-based input method to allow the user to provide knowledge as a seed to generate the requested output.

Tang et al. [26] explored the usage of AI for English language generation through the Seq2Seq [27] approach using LSTM (Long short-term memory). This AI architecture makes generating language much more autonomous but is void of any sorts of automata and grammar implementation, but rather showcases how grammar-based approaches are not always the best approach. This type of method was highlighted further by Gao et al. [28] using machine learning techniques and the same Seq2Seq approach to automatically generate protocols such as FTP. Their implementation uses an initial grammar-based fuzzer [14] to obtain input data which is then transformed as features for the model.

Lu et al. [29] has proposed a GAN (Generative Adversarial Network)based Method for Generating SQL Injection Attacks. Their tool is not intended for generating SQL syntax but for augmenting already existing SQL attack samples. Thus, the generation of the new samples requires the original dataset as input using genetic algorithms.

We are trying to avoid many of the studies using a form of "Text to Language" as we want to generate knowledge without any required input. These types of proposals are still very interesting which are evident through each of the authors' contributions which do appear to improve upon the problems known from NLIDB [30] such as the "Empty prompt" problem, uncertain linguistic coverage problem and Linguistic vs. Concept misalignment.

E. META ANALYSIS

We have been selecting the following set of well-known research publishers and have produced a consistent search in said publishers.

Search: "SQL" AND "Generative AI" in "All Metadata" in journal papers and conferences from the last 10 years (2013):

- IEEE via IEEEXplorer -> Provided only 1 paper that has been included
- Elsevier via Scopus -> Provided 3 paper, 4 that has been included.
- Springer via SpringerLink -> Provided 3 journal articles and none of them are generating SQL language.
- Springer via SpringerLink -> Provided 15 conferences and none of them are generating SQL language.

III. PROPOSED METHODOLOGY

A traditional grammar is mainly defined using two types of grammar rules: Lexer rules and Parser rules. They are used to validate if a given input follows the rules of such grammar to determine its lexical and syntactic validation. For example, an SQL grammar allows one to determine if a SQL statement is syntactically valid. However, our purpose is not the traditional one, we aim to generate new knowledge without any previous input using only the grammar as a blueprint to aid in the generation of output complimented by ANTLR4's ATN. Following the same running example, our aim is to generate SQL statements that are both lexical and syntactically valid without any given input stream. Thus, there is no input to validate against and there is also a clear need to create a non-traditional grammar focused on the generation of outputs instead of on the validation of inputs.

This is what we have coined as generative-friendly grammar. This methodology describes the steps required to achieve a generative-friendly grammar that is suitable to be used to automatically generate syntactically correct new source code. Fig. 1 illustrates the steps of the proposed methodology. It should be noted that although this methodology proposes a generic way to generate the source code of different languages, the use case for using the generated code will equally play an important role in its success. It is a pragmatic methodology that also aims to highlight potential areas of concern. If an EBNF grammar can be bespoke or modified with the proposed methodology while still producing the same language, then it can be considered generative-friendly.

A. GRAMMAR SCOPE REDUCTION

see *(step 1 in Fig. 1)*. Reducing the scope of the grammar facilitates the decrease in complexity of the search space to explore the generation of the source code through the limitation of what can be generated using a subset of the original grammar rules. Grammar can be problematic due to the ambiguity it contains. Although some ambiguity is natural and not necessarily problematic, it can certainly hinder language generation. This can be considered an optimization for use case customization. An example would be an SQL grammar where only DQL (Data Query Language) statements are needed, where the objective is to only generate queries that select data rather than modify it

TABLE 1. Related work comparisons.

	Authors	Language Generated	Generic Language Support	Language Grammar	Weight Driven	AI Model	Architecture	Use Case	Validation approach	Implementation
AI-Driven Sentence Generation	Gao et al [28]	FTP Protocol	No	BooFuzz Fuzzer	Yes	Seq2Seq	3-Layer Stacked LSTM	Protocol Fuzzing	Unit-Test	Prototype
	Lu et al [29]	SQL	No	N/A	No	Genetic Algorithm	Not Available	SQL Injection	Metric	Prototype
	Bajaj et al [25]	English	No	English	N/A	GTP-3	LLVM	Modeling Tasks	Prototype	Prototype
	John et al [24]	English	No	English	N/A	GPT-3	LLVM	Use Case Modeling	Prototype	Prototype
	Tang et al [26]	English	No	NER Toolkit	Yes	Seq2Seq	4-Layer LSTM	Knowledge -Based Question Generation	Metric	Prototype
	Parikh et al. [22]	SQL	No	Spider-Dataset	Yes	RAT-SQL	SmBop	SQL Fuzzing	Unit-Test	Prototype
Fuzzer-Based Sentence Generation	Wang et al [21]	SQL	No	DFA	No	No	EXT-DBFSM	SQL -System Testing	Unit-Test	Prototype
SQL Sentence Generation	Anisyah et al [13]	SQL	No	NLP /SQL	No	No	PC-PATR	Imperative sentence to SQL translation	Prototype	Prototype
	Datachat [12]	SQL	No	NLP	No	No	N/A	Data Analytic	Prototype	Prototype
	Sugandhika et al [11]	SQL	No	NLP /XML	No	N/A	NLID (SQL)	SQL Complexity Reduction using NLP	Questionnaire /Metric	Prototype
Grammar Assisted Sentence Generation	Pires et al [18]	Java	Yes	KDM /XML	No	No	KDM2Xunit	Generic Test-Case Generation	N/A	N/A
	Palmas et al [16]	English	No	CFG	No	No	Expressionist	Interactive dialog generation using CFG alongside sentiment analysis	User -Feedback	Prototype
	Sargsyan et al [14]	C++, Python, Java	Yes	EBNF /ATN	No	No	ATN Graph	Multi -Language Fuzzing	Prototype	Prototype
	Sargsyan et al [15]	All Available ANTLR4 EBNF Grammars	Yes	EBNF /ATN	Yes	No	ATN Graph	Multi -Language Fuzzing	Prototype	Prototype
	Ours (this paper)	All Available ANTLR4 EBNF Grammars	Yes	EBNF /ATN	Yes (Pluggable)	Yes (Pluggable)	AI-ATN Graph	Network Attack Identification	Prototype	Prototype

using DML (Data Manipulation Language) and DDL (Data Definition Language) statements. In such a scenario, stripping out elements of the original EBNF grammar would be a pre-requisite to control the unintended inclusion of generated DQL and DML statements.

If the user has decided to create a bespoke grammar from the ground up, then it will discernibly have reduction designed into it, and thus this step of the methodology is considered optional. If the full scope of the grammar is required, then the subsequent steps of the methodology will aid in helping control the combinatory runaway that can sometimes occur. We believe that this type of reduction could be ignored once the medium used for decisions i.e., a form of AI is integrated. The reader may ask why, which is valid. Using randomized-based selection inherently voids any essence of uniformity and predictability to learn from. AI integration would infer unambiguous generations through learning the rewarding transitions to take.

B. REDUCE/REMOVE RECURSION

See *(step 2 in Fig 1)*. Certain types of recursions within EBNF grammars can possess a grammatically natural capability to aid with parsing but are heavily discouraged when using a top-down parser. Left recursion is the most problematic due to the concern associated with infinite loops, with right recursion being the most preferable option. *Table 3* shows the EBNF notation for transforming left recursive productions. This can be applied to any left recursive production rule.

However, in terms of generation, the associativity and structure of the intended output may influence the design and integration of the logic. We first have to understand how ANTLR4 deals with left recursion. ANTLR4 when presented with a production rule which uses left-recursion will recognise and alter the rule by transforming it.

TABLE 2. Prototypes expression generation outputs.

PROTOTYPE GENERATED EXPRESSIONS
(822 / (7 * (691) - ((3)))) - 741
(9)
((1 + 07)) / (((7 - 5 * ((706)) / (294) * (1))))
(6 - (71) - 4922)
28

1) DEALING WITH LEFT RECURSION

As we have discussed the concern surrounding left recursion, let us go further by referring to Table 4 which helps add more context. ANTLR4 when presented with a production rule that is left- recursive will mitigate the said rule through the aforementioned transformation process. This should highlight the importance both star and plus loops maintain during the generation process. However, it should be noted that more logic is required for recursion completion within the generator class which is discussed at a later stage in the paper. Table 3 provides an example of the generated output of the prototype based on the grammar seen in (Table 4, col 1). When generating a language we are not concerned about matching, but rather its control from a potential recursive explosion caused by recursive rules within the grammar. We can dictate

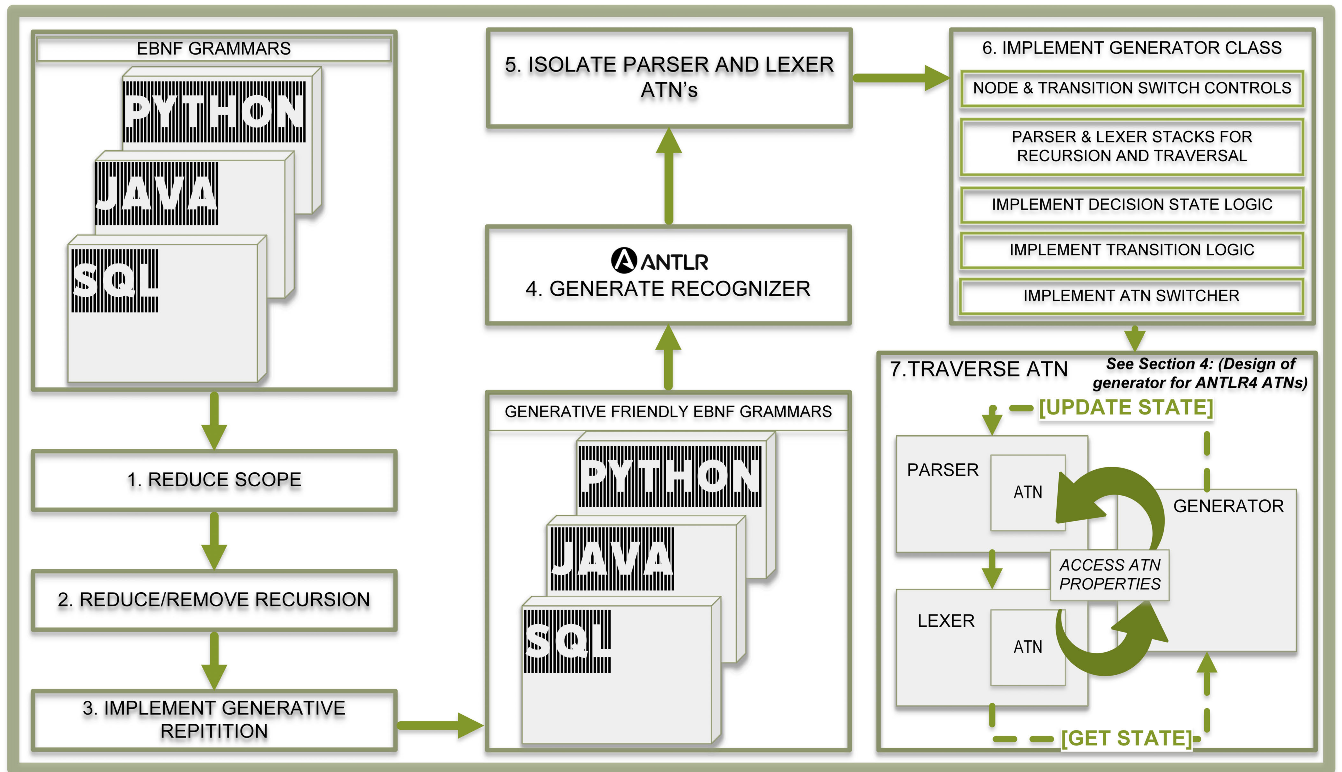


FIGURE 1. Framework for generative friendly grammar design and generation.

TABLE 3. Left recursion elimination.

Left Recursion Elimination
$A ::= Aa \mid B$
$A ::= BA'$
$A' ::= aA' \mid \varepsilon$

through logic within the generator when and how many times recursive rules are permitted. Recursive rules do present opportunities for generating more expressive languages, but carry a risk for memory exhaustion unless pragmatically mitigated through some form of monitoring control and rule transformation using left-recursion elimination. This risk is usually interrelated with the permutations of the rule. The ambiguity requires the implementation of a more comprehensive approach by extending the logic inside the generator. This is done to limit the repetition of the recursive rules.

It also requires logic for monitoring the recursive depth, with further logic required to correctly exit the rule to its intended follow-state, thus ensuring syntactic correctness. If the reader can refer to (Table 4, col 1, row 1), the example grammar highlights an assignment rule that can recognize an assignment in conformity with matching the input with the expected syntax. While generating an assignment from the rule, a syntactically similar output can be achieved with alternative results similar to that of a transitive-based algebraic expression, see (Table 4, col 3, row 1). The purpose of

TABLE 4. ANTLR4 left recursive rule implications.

Rule	Description	Generated Output
assignment: assignment EQ assignment | INT | ID	(Direct Left Recursion)	0 = c d = 8 h 2 = f = 3
assignment: ID EQ integer | ID EQ ID	(No Recursion)	ed = 74 d = 60 ha = ef cf = 9 ac = 1

this explanation is to highlight how certain production rules using recursion can go beyond the intended expectation of what the rule was designed to produce. This could be an undesirable result which helps emphasise the consideration needed when designing and or modifying rules when viewed from a generative perspective.

The removal of recursive rules is preferred, so long as it does not change the language it produces. If this is maintained, then recursive removal should be considered. The ambiguity from recursive rules on large grammars can be difficult to overcome and is not always generative friendly. Although recursion presents a problem, we do intend to show a pragmatic solution to ease some of the impact caused by uncontrolled recursion. We will discuss this at a later stage in the proposed methodology by way of stack implementations.

C. GENERATIVE REPETITION

See *(step 3 in Fig. 1)*. EBNF grammar enhances the ATN's capabilities through the definition of grammar rules when used alongside explicit quantifiers. There are three important EBNF quantifiers that enable **(GO -Generative Optionality)**, a term we created to better contextualize the disparity of the process from the understood paradigm of matching sequences of patterns. Please refer to Fig. 2 for a visual representation. The first two are the **Kleene plus '+'** and the **Kleene star '*'**. In basic terms, they are loops used for matching repetitive sequences. The third, being the **Question mark '?'** provides optionality where an absence of generation or by way of alternative choices is an acceptable outcome. Through the course of this paper, each of their observed generative advantages/disadvantages will become more apparent.

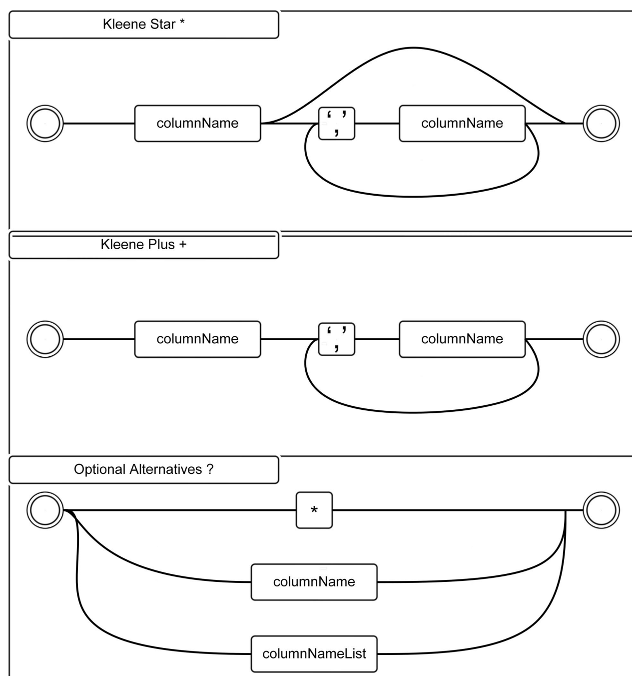


FIGURE 2. EBNF Quantifiers comparison.

We regularly see such notation in EBNF grammars and regular expressions which enable us to define a repetitive pattern pertaining to their greedy and non-greedy behavior.

```
1  STRING: '"' [a-Za-z]+ '"';
```

LISTING. 1. Greedy & Non-greedy implications.

Greedy and non-greedy quantifiers do not apply in the same way when they are being used for parsing as a goal as when they are used to generate sentences. See Listing. 1. From a parsing perspective, this lexer rule will match as many characters as possible of the alphabet up until the closing double quote. If we applied the non-greedy (lazy) notation "**+?**", we would match each character and proceed to look for the double quote for every matched character to hasten our exit. Furthermore, from a generative perspective when applying non-greedy notation it will have zero effect on the traversal shape of the graph when contrasted against its greedy notation. For generating language, not parsing, both have the same functionality. This is because the greedy and non-greedy quantifiers only apply when there is an input to match against. Notice there is no input to match in our approach.

When using *GO*, we must flip this concept and interpret it as permitting a potentiality for skipping or generating a sequence of tokens, irrespective of the medium used to make that decision, be it random, AI, etc. The overall recommendation for see *(step 3 in Fig. 1)*, is dependent on the intended language to be generated. Both parser and lexer generative potential for generating pattern repetitions should be carefully considered when applying such EBNF quantifiers to bespoke grammars. This consideration becomes more imperative where recursive rules are present and operating within the influence of a quantifiers loop due to how ANTLR4 applies rule transformations on direct left recursive production rules. This design choice can become more problematic for randomised-based transition selection where scaled combinatory decisions can quickly result in runaway generations unable to exit in an acceptable time-frame as previously discussed. This is why our proposition for integrating an AI model into the proposed generator is an approach that makes sense. An AI model would provide the functionality for dealing with such complexity and unknowns.

Through bespoke grammars, if a repetitive sequence is desired for a particular rule, the application of the Kleene plus quantifier would enable a minimum of one pass before hitting a decision state where a decision can be made to continue generating the sequence or exit the rule. The search space from the usage of these quantifiers will increase, be it through randomized or intelligent decision optimization. Overall, the generated language will have more variance in its output when using them. Original grammars may need to be limited due to their repetitive potential resulting from the combinatory explosion which can occur if random selection is the medium used for path traversal. Larger grammars can become cumbersome to alter. However, identifying where

repetition occurs and deciding if it is needed should be a decision that is made based on use case requirements

D. GENERATE RECOGNISER

See *(step 4 in Fig. 1)*. The recognizer is the combination of both the lexer and parser. Both components contain a list of ATNs. These ATNs are used alongside a deterministic finite automaton (DFA) to help predict the correct path to take during parsing. It allows us to construct syntactically correct languages without a need to parse any input. The ATN is a symbolic formalism of the input EBNF grammar. The ATN of both lexer and parser store all states and edges. This is the first step that is tailored to a concrete ATN generation prototype. In our case, we are using ANTLR4's recognizer for this purpose and it will be used in the rest of the steps.

E. ISOLATE PARSER AND LEXER ATNS

See *(Step 5 in Fig. 1)*. Generating a language from a grammar requires knowledge for accessing the ATNs. These provide us with a logical delineation of our grammar in a graph state form. The ATNs are the building blocks of the grammar attributable to their production rules. The ATN accommodates a list of states and transitions. Conceptually, no graph state structure exists as a whole. The parser and lexer ATNs store an ordered list of all states associated with a rules ATN. It is the data within them that specifies the relationships, thus it is not a concern by means of obtaining the first state which represents the root rule of the grammar and the first index position within the ATN state list.

F. IMPLEMENT GENERATOR CLASS

Fig. 3 shows the reader a complete overview of the general flow of the logic proposed to achieve the generation of the sentences. The different elements of the figure are explained in the full section.

See *(step 6 in Fig. 1)*. This will be the most essential component for generating a language. It consists of 5 crucial design principles used for prototyping, each of which helps traverse the ATN and manage incomplete recursive rule visitations in the ATN graph state machine. Although recursion is not preferable, the pragmatic solution we propose helps within reason to avoid infinite recursion. These 5 design principles will now be discussed.

1) NODE AND TRANSITION SWITCH CONTROLS

It is imperative that whilst traversing the ATN, a means for recognizing what node state and what transition state you are on is applied. The current ATN state should always be made local to both switches inside the while loop, see Fig. 4. Using this two-switch approach helps remove the duplicity of additional switches nested in each node type case block.

2) PARSER AND LEXER STACK FOR RECURSION AND TRAVERSAL

During the early stages of development for our proposed generator, a problem was identified with the use of random path traversal selection in exit states. Exit states can have many options to choose from, such as the rules they were invoked from. To mitigate this problem, our solution implemented two stacks, one for rule correction and one for recursion which push the *'follow-state'* numbers during rule transitions onto the stack.

This is the state number that appears after the rule before transitioning into it. Thus, once the stop state of the rule is reached, the parser state is set to the popped follow state value ensuring random selection has no effect when exiting out of a rule.

The question may arise as to the justification for implementing two stacks, however, this will be explained at a later stage in the paper. When dealing with recursive rules, the goal is to imitate recursion with the addition of using the stack size to keep track of depth. To visualize and contextualize this concern, see Fig. 6.

The reader may notice when a rule is being called, the follow-state number is stacked, i.e. *3 recursive rule entries*. When a decision is randomly made to designate a path that deviates from the recursive pattern, it results in transitioning to a stop state. This initiates the process of rule completion by popping the follow-state from the stack allowing the previous recursion to continue on the proper ATN traversal path. Please refer to Table. 6 which shows the proposed prototype generating from grammar as shown in Fig. 6.

TABLE 5. Recursive completion.

PROTOTYPE FUNCTION CALL GENERATIONS
COUNT( )
COUNT(AVG(COUNT(COUNT(MAX(AVG( ))))))
COUNT(MAX(MAX(AVG(AVG(MAX( ))))))
COUNT(AVG(MAX(MAX(MAX(AVG( ))))))
MAX(MAX(COUNT(COUNT(MAX(COUNT( ))))))

3) IMPLEMENTING DECISION LOGIC

This logic should be executed when traversing a node dependent on its specific type. Each ATN node has a unique type, and some of them are key for traversal direction such as the Decision state types. These decision states are fundamental to performing the selection of one of the multiple transitions that can be selected from such a state.

This selection is which aids in generative variability in the search space. Any AI component would interface with these specific state types. The goal would be to allow the AI to deduce the correct transition to take based on its previous visitations and paths taken.

4) IMPLEMENT TRANSITION LOGIC

It is equally important to know which transition the ATN is on. This logic requires execution when traversing a transition in the ATN depending on its type. The two most important transition types for token acquisition are **ATOM** transition and **SET** transition. Other transitions such as the rule-transition

FIGURE 3. System Flow for the Language Generation Tool.

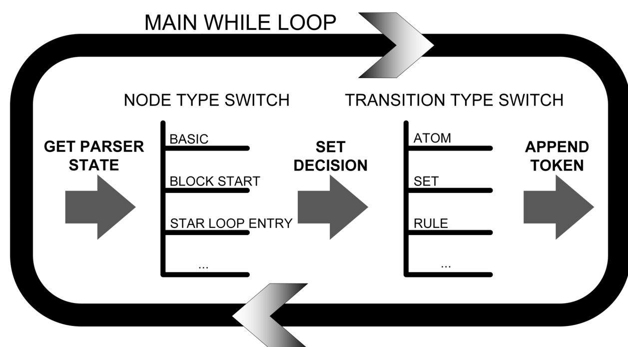


FIGURE 4. Switch Control Diagram.

type aid as alerts that help other parts of the generator object, such as the discussed stack implementations.

5) IMPLEMENT ATN SWITCHER

Both ATN belonging to the parser and lexer are used for generating language. The advantage of generating from lexer rules is in the EBNF shorthand notation which is only applicable to lexer rules.

The lexer generator is identical to the main generator class serving as a composite property, but instead traverses the lexer ATN and updates the lexer state during its traversal. If the benefits of utilizing the lexer rules do not outweigh its implementation, then transferring the definitions within them to a new parser rule can equally be done.

Please refer to Fig. 5 which illustrates the process of identifying when to switch over.

We feel that having lexer generation capabilities provides an advantage in terms of usability for the proposed prototype. This ensures compatibility for generating languages when designing or using already available grammar. The example grammar shown here Listing. 2 highlights a concern from the perspective of the parser and ATN.

If an attempt is made to append the result of the lexer "INT" production rule, it will return a null value. It is this null value

```
parse: 'my' 'age' 'is' INT;
INT: [0-9]+;
```

LISTING. 2. ATNSwitcher Example EBNF Grammar.

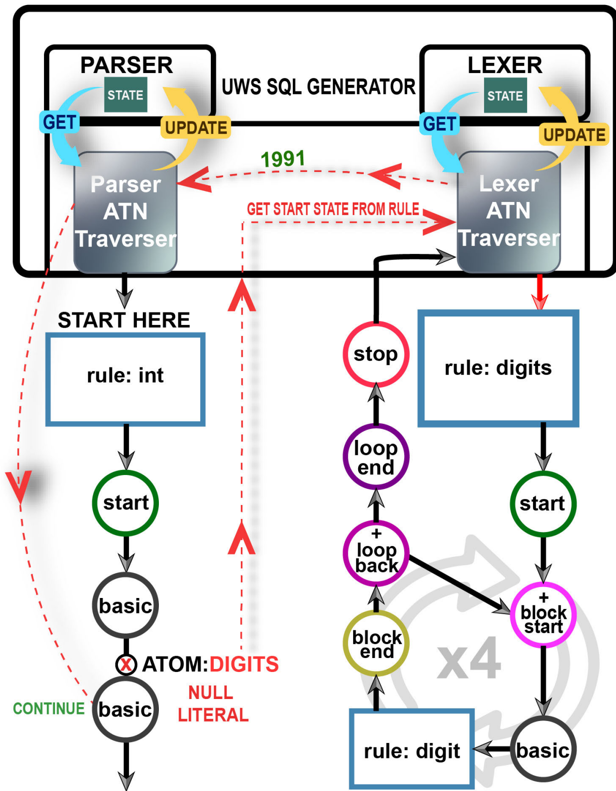


FIGURE 5. ATN Switcher Process Diagram.

that acts as the indicator that the generator has reached a non-string literal lexer rule.

This is the key to performing the switch during the traversal across ATNs so that the generation continues with the lexer ATN. Our approach assumes the lexer and its ATN component are void of recursive rules. We can always ensure that when the lexer ATN finishes generating tokens, there will always be a return to the parser ATN.

IV. DESIGN OF THE GENERATOR USING ANTLR4 AUGMENTED TRANSITIONAL NETWORKS

This section will narrow down the scope of the methodology for ANTLR4 and its ATN components by explaining in detail the design approaches of each of the different steps associated with the grammar generator, as they are introduced in *(Fig. 1, step 6)*.

A. NODE AND TRANSITION SWITCH CONTROLS

The following types of states represent all the possible conditions that can occur in the ATN generated from the EBNF grammar and represent different events that can be processed across the traversal of the network. Decision states imply nondeterminism, meaning that multiple paths can be chosen to achieve the same intended outcome of reaching the end. The medium used, i.e. through AI, would greatly influence the generated output based on those decisions being made. Some of these ATN states are:

- **BLOCK-START**
- **STAR-BLOCK-START**
- **STAR-LOOP-ENTRY**

These states represent the beginning of a decision process, usually represented by the pipe symbol '|' in EBNF for alternatives, or '?' for optional inclusion when used alongside sub-rules. These three node types can be associated with various quantifiers as previously discussed, such as the plus '+' symbol (one or more), or the star '*' (zero or more) symbol representing rules and or sub-rules repetitive capabilities. Or put simply, a loop within the ATN requiring explicit passes, or the option to skip it entirely. Analogously we do have the END counterparts:

- **BLOCK-END**
- **LOOP-END**

These states have deterministic transitions where no decision evaluation is taken as they only have one available transition to traverse unless purposely deviated from the grammar rule representation in the ATN for use case reasons. There is another type of node with an end counterpart that is non-deterministic, such as:

- **PLUS-LOOP-BACK**

Although both loop types use the loop-end node, plus-loop states also referred to as kleene plus, have their decision at the end of the looping process. A plus '+' loop signifies that at least one code block will be generated and thus, a decision must be taken at the end to decide on continuing or breaking the loop. There are two other ATN state types that represent the entry and exit point of the available rules in the grammar:

- **RULE-START**
- **RULE-STOP**

The start node is deterministic whereas the stop node is non-deterministic as a result of its multiple transitions into other rules it is referenced. This justifies the stack implementation we previously discussed in (*subsection III-F2*), which noted the concern of traversing into the incorrect rule when exiting. With respect to transitions, the following types of states represent the most common situations that can occur in the ATN generated from the ANTLR4 EBNF grammar:

- **ATOM** is an atomic transition where the expected token between the initial state and the next can be found. This expected token is acquired by-way of its literal name.
- **EPSILON** Transitions represent the vast majority of transitions. They represent pathways in which no symbol is required to dictate what state should be visited.
- **SET** is a transition type in which the ATN stores SetIntervals comprised of multiple expected tokens for that particular state. Unlike ATOM where only one can

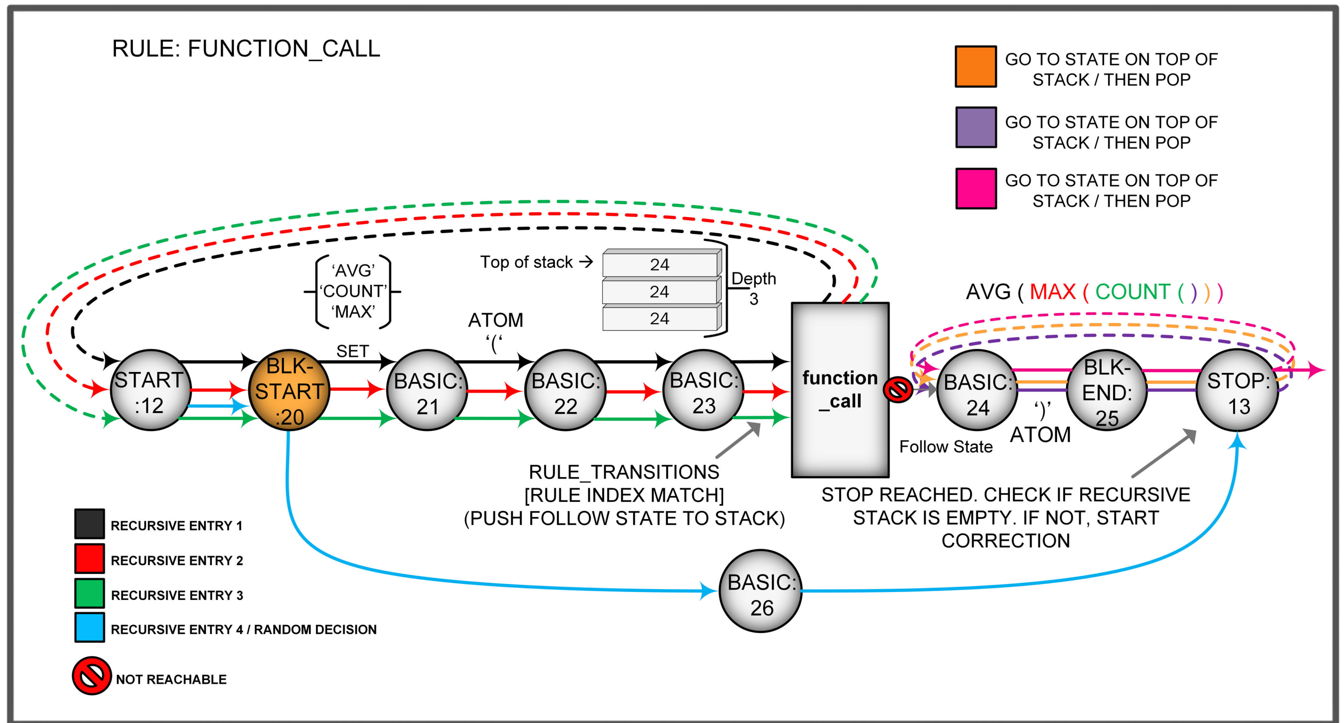


FIGURE 6. Recursive recovery and token acquisition of function call rule.

be chosen, SET Transitions provide many options while still remaining syntactically valid.

- **RULE-TRANSITION** is a transition type that signifies the entry and beginning of another rule in the ATN state machine.

It is important to remark that it is exactly in these non-deterministic states such as(*BLOCK- START*, *PLUS-BLOCK-START*, *STAR-LOOP-ENTRY* and *PLUS-LOOP-BACK*) where we leave space for the AI to help drive the decision of the non-deterministic choices. The approach here is to implement a weight- based system that enables AI to drive such choices whilst optimizing the output using a heuristic reward function. This is probably one of the most significant insights of the proposed methodology where we have clearly identified those states where the AI implementation must interface during the traversal of the ATN.

B. PARSER AND LEXER STACK FOR RECURSION AND TRAVERSAL

It has been decided to use two stacks for the parser and one for the lexer since lexer rules with recursion are generally avoided. The first stack, ruleCorrectorStack, serves the purpose of ensuring that the traversal of the ATN is syntactically correct so that when exiting a rule, there remains a continuation with the path of the previously invoked parent's rule state. The second stack, the recursive stack has a similar purpose but is used specifically for managing the recursive state visitations within the same rule. This allows accurate depth tracking based on its size, thus the need for two separate stacks.

There are key events in the traversal of the ATN that have been selected to manage the stacks used to control the evaluation of the rules. One of these events occurs during a transition to another rule while the ATN is traversing from state to state. (Algorithm. 1) outlines the pseudo-code

Algorithm 1 Implementing Stack Logic Stage 1

```
Data: decision, ruleTransition, currentState
Result: Update recursiveStack or ruleCorrectorStack
begin
    decision ← ruleTransition switch
     decision.transitionType do
        case Transition.RULE do
            if decision.target.ruleIndex =
             currentState.ruleIndex then
                recursiveStack ← stack() followState
                 ← decision.followState /* push onto
                 stack */
                 recursiveStack.push(followState)
            else
                ruleCorrectorStack ← stack()
                 followState ← decision.followState /*
                 push onto stack */
                 ruleCorrectorStack.push(followState)
```

used to handle such scenarios by pushing the invoked rules follow-state onto the respective stacks based on specific

conditions such as rule index match indicating recursion, and if not, rule correction during all rule transitions for correct rule exiting. Another key moment is during the traversal of a rule when exiting RULE-STOP. This signifies the moment for determining its continuation and correct path decision across the ATN states. This is where we perform the pop of the previously obtained follow-state before entering the rule. If the reader examines (Algorithm. 2), it will highlight the logic used when dealing with such functionality.

Algorithm 2 Implementing Stack Logic Stage 2

Data: ATNState, currentState, parser, ruleCorrectorStack, recursiveStack
Result: Parser state is set to the popped follow state conditionally
switch *currentState* ← *stateType()* **do**
  case *ATNState.RULE_STOP* **do**
    if ¬*ruleCorrectorStack* ← *isEmpty()* **then**
      if ¬*recursiveStack* ← *isEmpty()* **then**
        /* Set parser state to popped follow-state */ *parser* ← recursiveStack.pop()
      else
        /* Set parser state to popped follow-state */ *parser* ← ruleCorrectorStack.pop()
  otherwise do
    Continue

C. IMPLEMENTING DECISION LOGIC

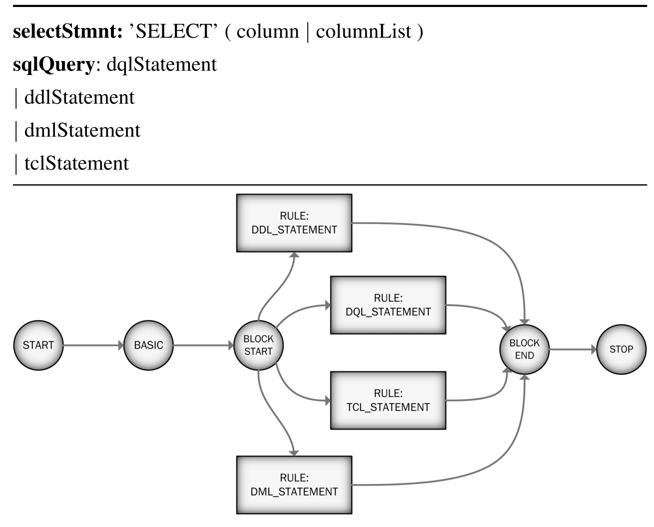
BLOCK-START: Let us use this simple parser rule seen at (Table 6, row 1) and the ATN that it generates seen in the second row below. Notice how the traversal of the 'sqlQuery' rule presents 4 alternatives based upon the grammar production rule. Any AI integration would almost certainly use such ATN state types to help drive the decision process. Block-Starts can encapsulate large areas to traverse making them excellent for offering search space variation.

This BLOCK-START state type is significant as it can contain alternative transitions associated with its rule or sub-rules. This state can conceptually be considered a high-priority decision state. When reached on the ATN and accessed through the generator ATN state type switch, the transitions should be retrieved, and a decision should be made on which one to traverse. (Algorithm. 3) shows the pseudo code associated with that decision where the current state transition selection is accessed. The decision must be within the index boundary which can scale depending on the number of available transitions.

STAR-LOOP-ENTRY: The Star-Loop-Entry state type is the entry point for a block that can be traversed zero or more times. We will use the example grammar shown in Table 7 to illustrate its *GO* including the output from the generator class, see *(Table 8)*.

TABLE 6. Block start decision ATN Diagram.

selectStmnt: 'SELECT' (column | columnList)
sqlQuery: dqlStatement
| ddlStatement
| dmlStatement
| tclStatement

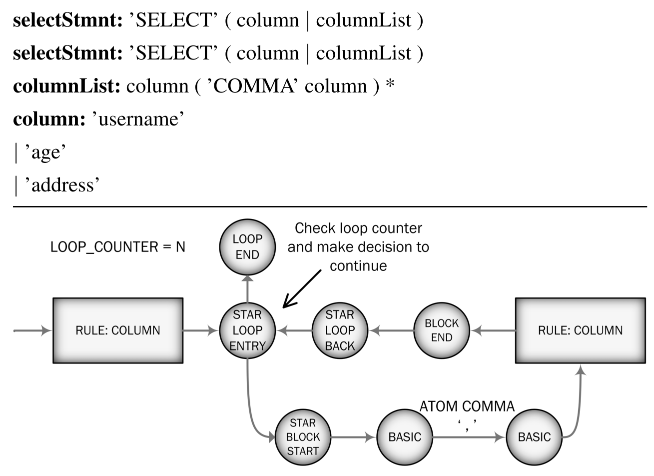


Algorithm 3 Block Start Logic

Data: BlockStartState, currentState, parser, blockStart
Result: Implementation for transition selection
begin
  blockStart ← (BlockStartState) currentState;
  parser.setState(blockStart.transition(decision) .target)

TABLE 7. Star-loop-entry decision ATN Diagram.

selectStmnt: 'SELECT' (column | columnList)
selectStmnt: 'SELECT' (column | columnList)
columnList: column ('COMMA' column) *
column: 'username'
| 'age'
| 'address'

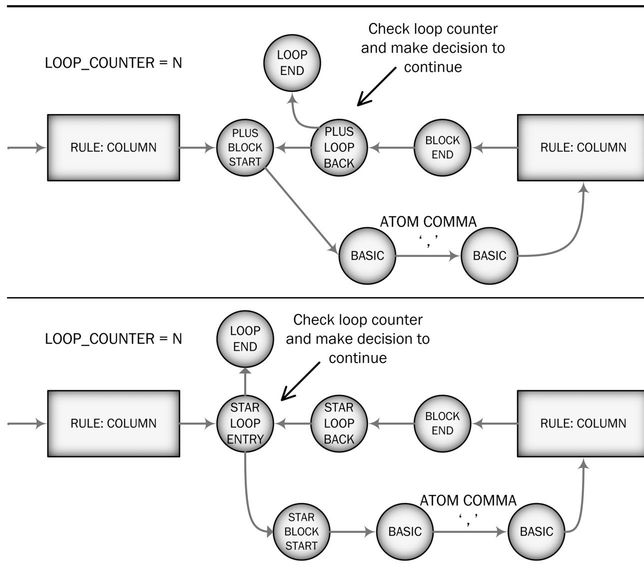


(Table 7, row 2) represents a typical ATN structure for STAR-LOOP-ENTRY states which is based upon the grammar sample in *(Table 7, row 1)*. As star loops provide an ability to loop zero or more times, implementing a variable to act as a counter is an essential requirement for controlling the loop.

TABLE 8. Star loop entry generations.

GENERATED
SELECT age , username , age , username , age , address , username
SELECT address
SELECT address , username , address , username , username
SELECT username , username , address , age , address , age , address
SELECT address , age , address , username , address
SELECT age , username , username , username , age , age
SELECT age
SELECT address , username , address , username , username

TABLE 9. Decision loop comparisons.



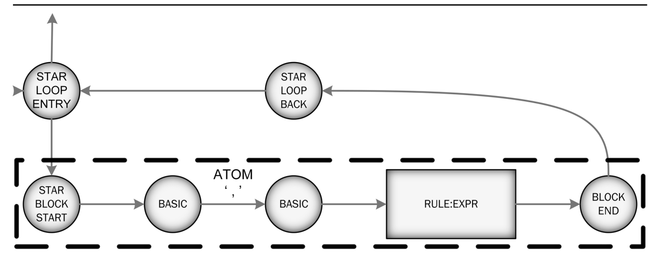
This is especially important when some rules can contain nested loops at a deeper level of the invocation stack which highlights a concern such as the risk for combinatory explosions. Recursive rules can also be looped within some grammars which emphasizes the importance of controlling the loop count, see *(Algorithm. 4)*. How the loop count varies at run-time is a use-case matter. It is also important to note, that once the loop end state is reached the counter should be zeroed. One may consider the conflicts of zeroing the counter with both loop types. A pragmatic solution is to introduce two loop counters and Boolean conditions to see which loop is still active. If the loop is not active once a loop end state is reached, it can be zeroed helping avoid any crossover interference. Another important factor that must be addressed is the recursive shut-off point. As loops play an important role within ANTLR4 during left-recursive rule transformation, the shut- off should happen inside the decision states of the two different loop types. This ensures a force to rule stop will not miss the traversal of certain transitions leading to incomplete generations. Once a condition is met, the stop state of the current rule should be set within the parser.

Algorithm 4 Star Loop Entry Logic

Data: starLoopEntry, loopCounter, maxLoops, ATNState, parser
Result: Loop control with user set boundary
maxLoop ← 5 **begin**
  if *loopCounter* ≥ *maxLoop* **then**
    for *each transition in starLoopEntry.getTransitions()* **do**
      if *transition.getStateType()* = *LOOP_END* **then**
        /* set parser state to target state number */
      else
        Set parser state using either randomisation or alternative method for selection

TABLE 10. Star block start decision function call ATN Diagram.

parse: functionCall
functionCall: ID '(' argumentList? ')'
argumentList: expr (',' expr)*
expr: INT
ID: $[a - zA - Z_][a - zA - Z_0 - 9]+$
INT: $[0 - 9]+$



PLUS-BLOCK states act **almost identically** to STAR-LOOP-ENTRY states in regards to its functional capabilities with the only real difference being the placement of the decision process which provides an edge for exiting the loop. This means that during the generation process, we are guaranteed to generate at least one column name at the very minimum. A comparison is provided in Table 9.

PLUS-LOOP-BACK: The plus-loop-back state is a decision state for a block that must be traversed one or more times. Using a basic grammar see (Table 10, row 1), we will highlight how such a constraint is applied. Please refer to *(Table 11, row 2)*. This provides insights into using plus loops, which reveals when used on a production rule or sub-rule, the generator will have no option but to loop at least once when first encountering the loop. The PLUS-LOOP-BACK state type is the only node with a capability for exiting this loop type. This signifies that the logic in the generator class must

TABLE 11. Plus-loop-entry decision ATN Diagram.

selectStmnt: 'SELECT' (column | columnList)
columnList: column ('COMMA' column) +
column: 'username'
| 'age'
| 'address'

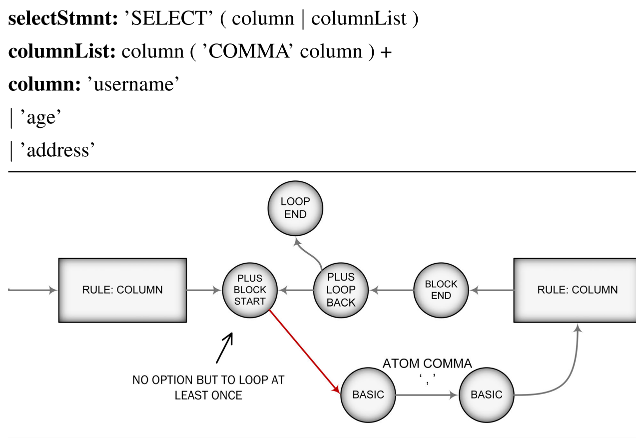


TABLE 12. Star block start function call generations.

Function Call
C2MyH()
jpA()
Ro6(62, 9, 5539)
FRcg(2869,4,26,5)
PaKo(97,59)

be bound to this node type which ensures a decision will be made to continue or break.

type represents the start of a code block in the grammar which must be traversed zero or more times inside of a loop. It is similar in nature to the block-start but offers alternatives during the looping process. If we use the rule provided in *(Table 10, Row 1)*, it illustrates a grammar that defines a function call. The row below provides the grammar as it would be represented in the parser ATN. The dashed box encapsulates the 'block' which the loop repeats N times. During the looping process, we can make decisions within the loop to alter the generated output. This example would generate a function name and call, including or excluding arguments. You can see from Table 12 the generated function calls from the proposed prototype. Randomized decisions dictated the arguments passed into the generated function and its name. We believe that with what has currently been discussed regarding decision logic, weight implementation would integrate nicely into such a system. These are just states connected by transitions, and as such, they would allow us to interface an AI model to intelligently control such decision-making processes.

D. IMPLEMENT TRANSITION LOGIC

It is equally as important to know which transition the ATN is on. This logic requires execution when traversing a transition in the ATN depending on its type. The two most important transition types for token acquisition are ATOM transition and

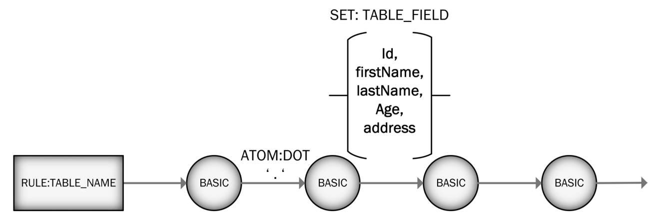


FIGURE 7. Set transition ATN diagram.

SET transition. Other transitions such as the rule-transition type aid as alerts that help other parts of the generator object, such as the discussed stack implementations.

ATOM: The main logic associated with Atom transitions is for token acquisition, Lexer rule detection, and ATN traversal breaking. The break is associated with the root rule which contains the special EOF (End of File) token at the right edge of the production rule.

Algorithm 5 Implementing Token Acquisition and Break

```
Data: decision, atomTransition, currentState,
      atomToken
Result: Token aquisition from either parser or lexer,
        and main break
decision ← atomTransition begin
  switch decision ← transitionType do
    case Transition.ATOM do
      if atomTransition.label() == EOF then
        break main while loop
      else
        atomToken ←
          parser.getVocabulary().getLiteralName
          (atomTransition.label().getMinElement());

        if token == null then
          Lexer Rule detected Jump to Lexer
            ATN Append lexer token to string
        else
          Append parser token to string
    otherwise do
      Continue
```

Please refer to (*Algorithm. 5*), which provides the pseudo code for obtaining an atom token during the traversal of the ATN. it also shows how the generator knows when to stop the generating process using the aforementioned EOF token to break the main loop.

SET: Set transitions provide great variations and syntactically correct language alternatives. Set transitions provide alternative tokens which are expected at the transition the ATN is currently on. Only one set Interval alternative is selected from all the possible choices.

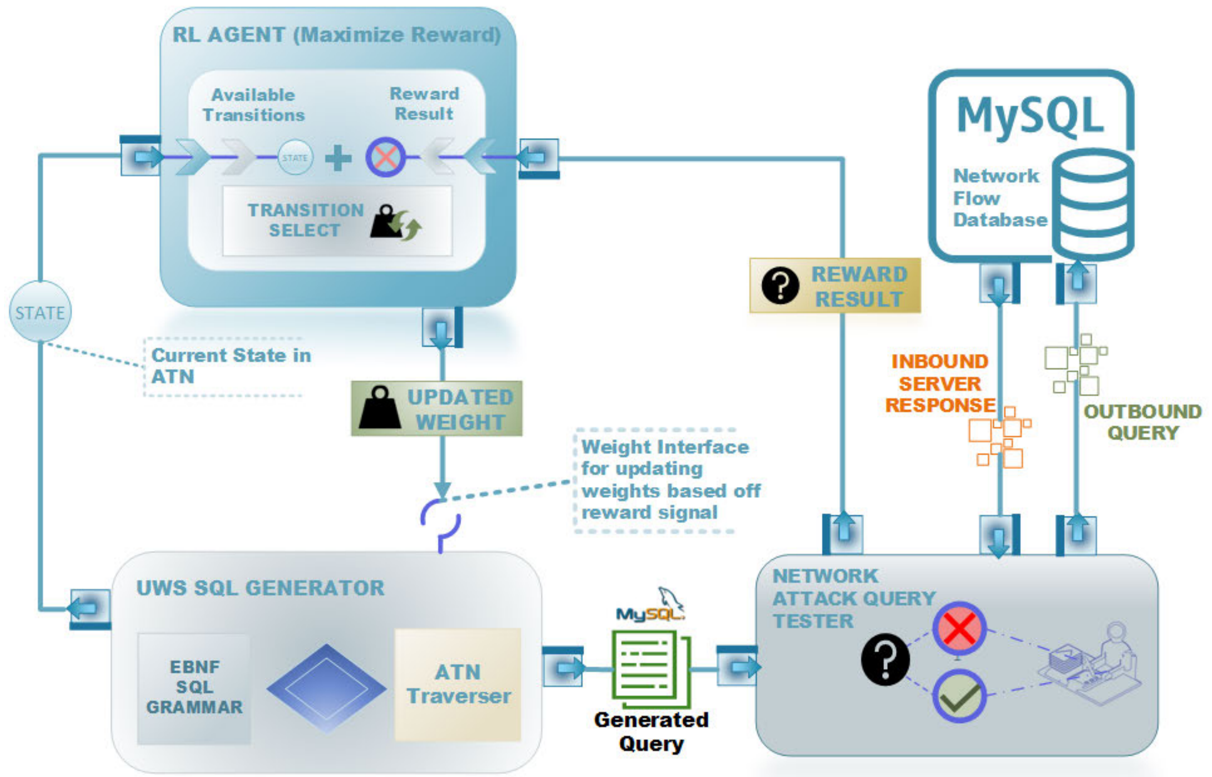


FIGURE 8. Reinforcement Learning interfaced with generator.

Let us take the example of a grammar representing SQL. One would expect a table entity to have multiple fields. These fields could be registered as set intervals as shown in Fig. 7. In simplistic terms when encountering a set transition, a list of potential tokens is made available to choose from. The generator's job is choosing one and appending it to the query string. Any AI integration could utilize sets by weighting each of the tokens in the set-list. The selection process could be controlled by the AI, allowing it to infer the correct semantics through training.

E. TRAVERSAL LOGIC DESIGN

This logic performs the continuous generation of the grammar as it contains a main while loop that keeps pushing us forward the transverse of the ATN.

The inclusion of the aforementioned switch referred to here in Fig. 5 helps by updating the parser state. Please refer to (*Algorithm. 6*) to see the implementation of this logic. The main loop will run until the EOF token is reached, as previously discussed. If a state's transitions amount to 1, then no decisions will take place, hence why the decision is set to 0. The bulk of transitions from states are going to be epsilon, which means there is only one transition to take.

Algorithm 6 Traversal Logic Design

```
Function GenerateFromGrammar(parser, lexer):
  begin
    parser.setState(0)
    continuation = true
    while continuation do
      parser ←
        currentState.transition(0).target
      currentIndex ← currentState.ruleIndex
        currentState = getATN().states ←
        parser.getState()
      switch currentState ← stateType()
       do
          make decision and set parser to target
           state *
      switch decision ← serializationType() do
          check for recursion * push to stacks
           follow state * obtains atom and set
           interval tokens * append to string
           builder object *
  return The generated language
```

F. ENVISIONED AI-DRIVEN GENERATION ARCHITECTURE

Fig. 8 illustrates our research use-case for the next iteration of our proposed SQL generator prototype. It highlights how an AI model could in theory be interfaced with the different components and flow direction of the generator. The main goal

of the SQL Generator can be analogized as a car providing the driver with the required interfaces needed for traversing forward from one point to the other. The driver is allocated a map that binds the driver within a topological boundary and defines the accepted rules of the road. The driver must make many decisions when arriving at intersections to choose the best route and is rewarded when told the decision had a positive outcome.

However, unbeknownst to the driver, these rewards are only given based upon the consequences of the driver's decisions which are being provided by another influence. The driver now has an incentive to train and to ensure that mistakes are minimal for the maximum attainment of rewards. While this process ensues, the history of the driver's chosen route is being logged. These are the names of the road signs passed by whilst on a particular road. These names of the different road signs when read in the order of traversal, constitute only a part of the route taken and direction gone. The driver only knows when the end is reached when the road sign encountered is named (End Of Road).

Now if we replace driver with AI, map with grammar, road with transition, road sign with token, and intersection for decision state, it starts to become apparent that this analogy at its core is analogous with RL (Reinforcement Learning). Based on our preliminary research thus far, this is the selected training method we consider an optimal first choice for AI generator integration. From a design perspective, it offers a pragmatic solution for interface compatibility due to the basic concept of the goal, which is maximizing the reward. This training method is more modular, which allows us to interface the AI around the SQL Generator more easily.

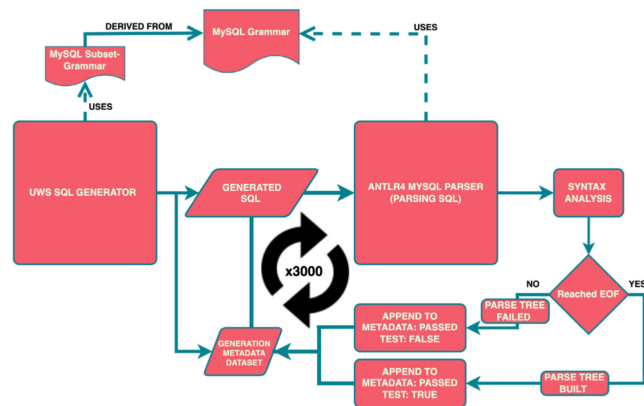


FIGURE 9. General Testing Process.

V. EXPERIMENTAL RESULTS

A. EXPERIMENT SUMMARY

Please refer to Fig. 9 which illustrates the testing, data collection, and validation process. The target language used was Java. An ANTLR4 MySQL grammar was used for validation, while a subset of the same grammar was used for generation. Various metrics were acquired through a process of logging the prototypes' randomized behavior, including the parameters set before execution to produce the generated output. The metrics collected were:

- The maximum depth reached.
- The number of total states visited.
- Generations attributed to the parser.
- Generations attributed to the lexer
- The maximum allowed rule visitations (set before execution)
- The maximum allowed recursion (set before execution)
- The maximum allowed loops (set before execution)
- The time taken to produce the generated query.
- The generated query string.

A total of 3000 samples were collected, of which 1750 met the conditions to satisfy the validity of the syntax. Higher depth resulted in greater ambiguity and failure. Additionally, generator capabilities were tested against grammar complexity to ascertain the impact on time. Findings showed that every 250 non-terminals incur a time penalty of **0.0027546ms**. Furthermore, testing was conducted to validate our solution proposed for recursive mitigation that showed positive results aligning with the aforementioned logic already discussed regarding the design and code to achieve this. Metadata from each generation was used to confirm the depth reached and corrections made to resolve it, with both showing equal values.

B. EXPERIMENTAL VALIDATION

1) SQL GENERATION

The approach taken for the validation has been to use a publicly available real-world ANTLR4 MySQL EBNF grammar for validation. To be concrete, the complete MySQL EBNF grammar was utilized for generation and validation. Later on, such grammar was reduced to a subset that only contains the Data Query Language (DQL) part of the SQL language leaving outside both Data Manipulation Language (DML) and Data Definition Language (DDL). The grammar contains 222 rules totaling a number of 841 lexer tokens, with a further 55 lexer rules which can be expanded into subsets of string literals through shorthand notation. Of the 222 rules, 34 of them have recursive properties. 3000 randomized generations of MySQL SELECT queries were produced by the proposed prototype. During every generation of the ATN, implemented in ANTLR4, both lexer and parser were traversed using the proposed approach explained previously in past sections of this paper.

It is worth indicating that the only comparable results available in the state of the art against our solution [14] do not have any available open source to be used for comparison. It has not allowed us to establish a baseline metric for evaluation between tools. The results are shown in Fig. 10). The figure represents the frequency on the Y axis related to the number of executions. The X-axis represents the maximum depth reached in the recursion stack during the generation, e.g. 6 represents the maximum depth in the recursion each particular generation reached. This has been achieved by implementing a metric observer design pattern which would

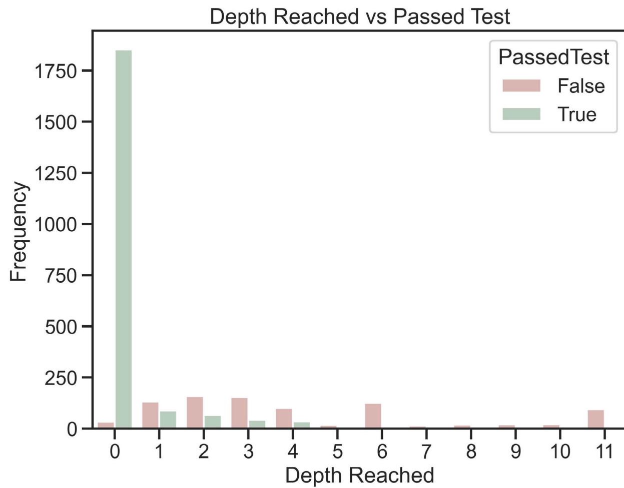


FIGURE 10. Depth Reached Vs passed Testing.

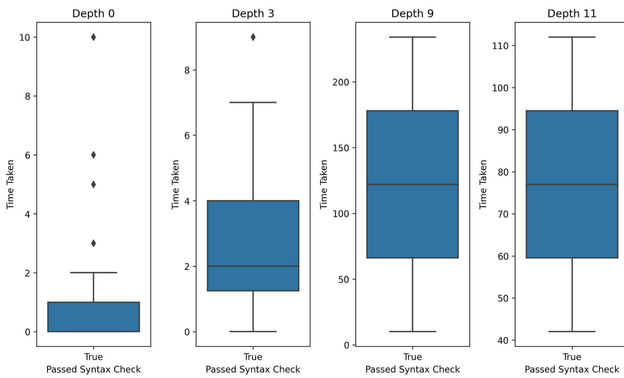


FIGURE 11. Depths Reached Distributions.

log all information pertaining to the generated query. It is worth mentioning that there are two different series available in the graph. One is related to generated queries that have passed syntax validation, whereas the other refers to those that have failed.

The test used to determine if a query is validated or not is defined as follows. After the query has been generated, the said query is fed to ANTLR4's input where syntax analysis can begin. The syntax analysis is validated on the basis of a successful parse tree being created. This means that syntactically, the proposed prototype has been successful.

The reader can see how 1750 passed the test when analyzed against the depth reached during generation. If the reader looks at Fig. 10 it will convey that most successful generations are heavily favored towards a depth of 0, with some success from 1 to 4. This may mean more parameters need to be tweaked inside the generator to ascertain the problems of the query losing coherence when depth increases. However, another factor may reside within lexer generations where the random nature of the generated tokens makes the syntactic validity of the parser generations more noisy and more prone to be misaligned with the grammar rules.

Fig. 11 highlights the distribution of the syntactically valid generations. If we take an example recursion depth, i.e. 3:

$$\bar{t} = \frac{1}{N}\sum_{i=1}^{N} t_i \pm s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(t_i - \bar{t})^2} \quad (1)$$

Please refer to (1) where **t** represents the average time of query generation at a depth of 3 where **n** represents the total number of queries that reached a depth of 3 and **s** represents the standard deviation in time taken to generate the query.

The average execution time is **2.857ms** with a standard deviation of **2.258ms**. When there is no recursion the analyzed data shows an average execution time of **0.307ms** with a standard deviation of **0.554ms**. Due to the greater amount of zero depth generations, a pronounced positive skew can be seen due to the occurrence of a greater amount of outliers.

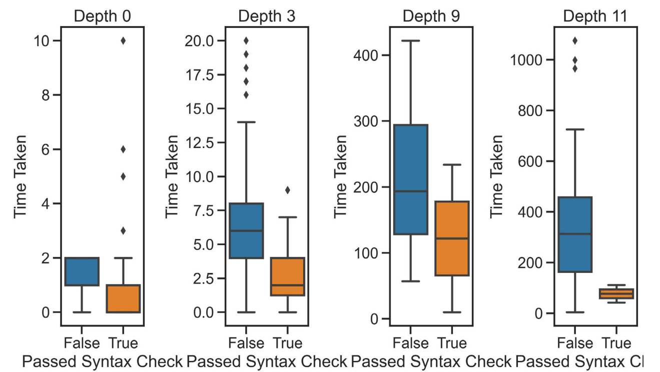


FIGURE 12. Depths Reached Distributions of passed and failed syntax check.

Fig. 12 shows to the reader the execution time associated with each of the 3000 different executed queries. The reader can see the relationship between execution time and the recursion depth reached and also the relationship between execution times and syntax validity, shown as passed or failed. It is worth mentioning that there is exponential growth over time as the depth is increased.

2) GRAMMAR COMPLEXITY EXPERIMENTATION

We have chosen to explore the capability of the proposed prototype and the consequential effect that grammar complexity incurs on generation time through additional non-terminals. This is important for two reasons. 1. It can reveal insights into grammar performance for token generation, and 2. It allows us to understand the proportional impact on generation time whilst generating the same number of tokens using additional non-terminals.

Although EBNF grammars can contain extended complexity such as recursive production rules, quantifiers, and alternatives, it is not our intent to include them as of yet. We instead want to focus on the core syntax such as production rules, terminals, and non-terminals.

The methodology for this experiment can be seen in Fig. 13 which illustrates the process of generating all

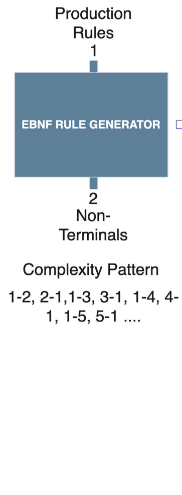


FIGURE 13. Complex Grammar Experimentation Method.

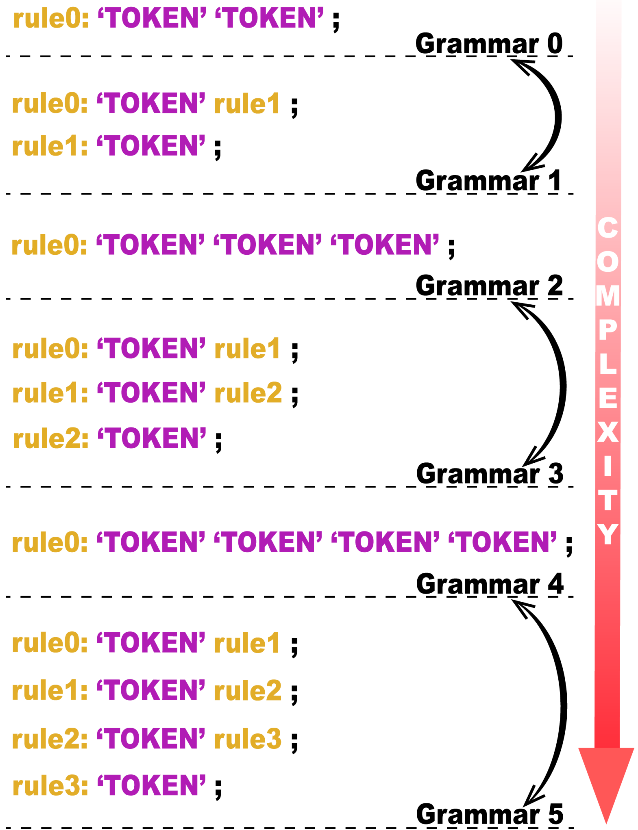


FIGURE 14. Grammar Complexity Pattern.

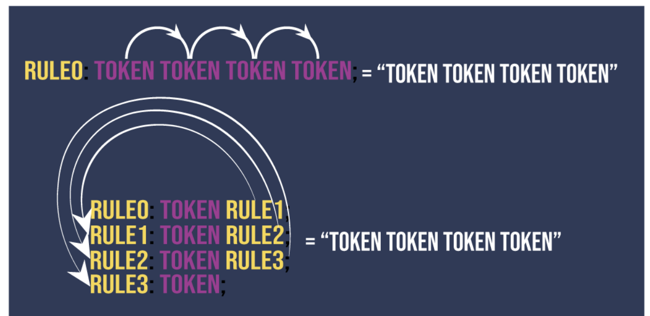


FIGURE 15. Grammar Complexity Association.

required grammars for the experiment using our EBNF rule generator. We generated over 6000 ANTLR4 EBNF grammars which were retrieved in a linear order and used to generate the recognizer which stores the ATN representations needed for traversal. Both parser and lexer are compiled and then dynamically loaded using Java's reflection features at run-time, thus allowing us to re-initialize and execute the generator prototype consecutively until all logical grammar representations are traversed using their ATNs. Furthermore, the metadata collected during the ATN traversal process is used to aggregate and assess the generator capabilities alongside grammar complexity when dealing with deep nested rule traversal and token generation.

Grammars are generated in order of complexity. This means that the first grammar generated will contain only two tokens with the second grammar generated maintaining a direct association with the previous through the total number of equal tokens.

Fig. 14 highlights the logic of the pattern utilized when implementing increasing complexity into each grammar. You can see as the terminal count increases in the baseline grammar, the following grammar must distribute the same amount of terminals over an equal number of rules.

If you direct your attention to Fig. 15 it reveals that although these grammars differ, the intended output remains the same. The only difference is in the way the tokens are acquired through the traversal process. This maintains an association between both grammars allowing us to statistically investigate the impact of additional non-terminals with equal token generation.

The results of the grammar complexity experiment can be visualized in Fig. 16. This experiment allowed us to quantify the impact on generation time through the inclusion of additional non-terminals. With our metadata data-set, we utilized a scatter plot with a Y-axis representing the time taken to generate the tokens, and an X-axis representing the total amount of tokens generated. We used a third color dimension to represent the number of rules traversed to highlight greater complexity. The red dash lines use a step count of 250 and indicate the association between the incremented grammars containing additional rules and the non-complex baselines below it. It is here from which our delta values and proportional impacts are extracted.

Table 13 provides 3 columns, the first being the deltas of time (2) relating to the differences observed from the final and initial steps. This is done in increments of 250 which we consider to be a balanced approach while maintaining an adequate number of results for analysis. Column two provides the deltas of rules traversed (3) and highlights the step count

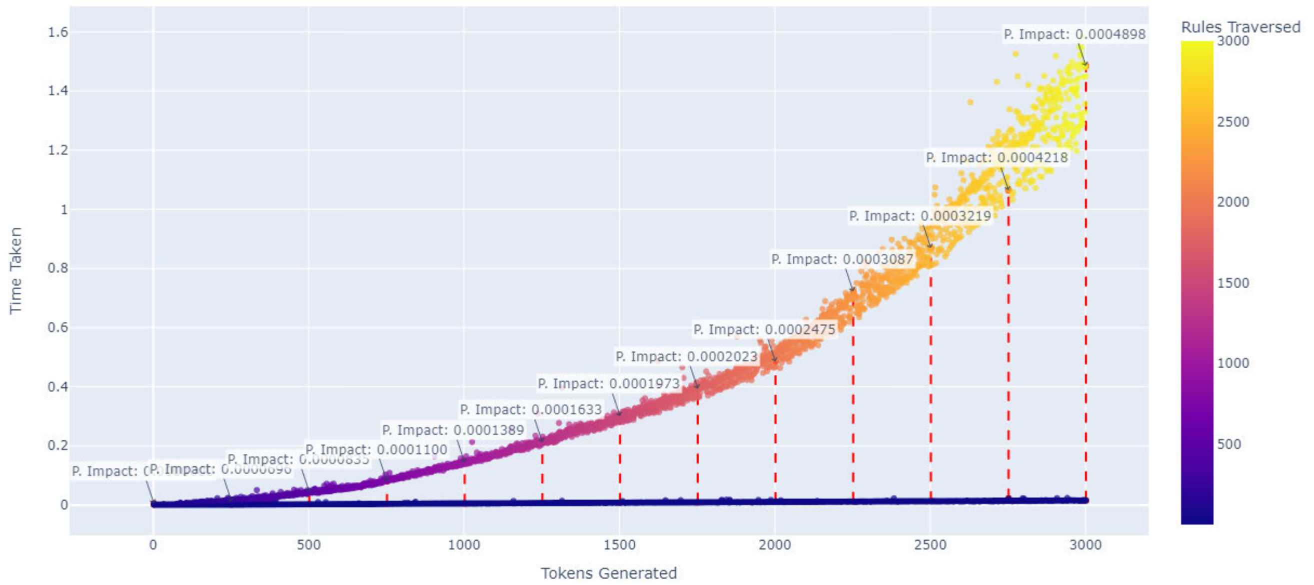


FIGURE 16. Impact on time through increasing non-terminals within prod rules.

TABLE 13. Deltas of Time taken and Rules Traversed with impact.

ΔTimeTaken	ΔRules Traversed	Proportional Impact (Milliseconds)
0.017401	250.0	0.0000696
0.041758	500.0	0.0000835
0.082515	750.0	0.0001100
0.138912	1000.0	0.0001389
0.204135	1250.0	0.0001633
0.295879	1500.0	0.0001973
0.354096	1750.0	0.0002023
0.494922	2000.0	0.0002475
0.694566	2250.0	0.0003087
0.804652	2500.0	0.0003219
1.160078	2750.0	0.0004218
1.469374	3000.0	0.0004898

used. Column three measures the proportional impact (4) incurred on the time taken to generate the tokens with the additional non-terminals.

$$\Delta \text{TimeTaken} = \text{TimeTaken}_{\text{final}} - \text{TimeTaken}_{\text{Initial}} \quad (2)$$

$$\Delta \text{RulesTraversed} = \text{RulesTraversed}_{\text{final}} - \text{RulesTraversed}_{\text{Initial}} \quad (3)$$

$$\text{impact} = \frac{\Delta \text{TimeTaken}}{(\Delta \text{Rules Traversed})} \quad (4)$$

The results show a linear relationship between time taken and rules traversed. As more non-terminals are added a rise in time is observed. The proportional impact shown in Table 13 does not scale by a constant factor; rather, the values show slight deviations in the amount of increase per 250 rules. This may indicate the computational load on the system as more rules require slightly more time for generation than the previous rule. However, the discrepancies are insignificant to warrant a deeper analysis. From these results, we can discern the average impact of grammar complexity concerning generation time for every 250 non-terminals in Milliseconds.

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{0.0027546}{12} = 0.00022955 \quad (5)$$

(5) shows us that the average impact on time per every 250 non-terminals equates to **0.0027546ms**, of which when converted to seconds is **0.0000027546** of a second. From the results, we can conclude that there is indeed an impact penalty when contrasted against the increase in non-terminals and tokens.

3) VALIDATING RECURSIVE MITIGATION

We have previously discussed in our paper the issues surrounding recursive rules contained within grammars and propositioned our solution for mitigating the exponential growth. Our solution makes use of a stack for maintaining a record of follow states with additional logic for conditional execution resulting in the correct recursive closure while maintaining syntactic validity.

We can demonstrate the effectiveness of our recursive mitigation proposal using a test grammar, see here (listing 3), and here Fig 17 which highlights how our constraint of permitted recursions can directly influence the ascend and descend behavior observed whilst traversing recursive rules in the ATN. The left plot provides a 2-dimensional view of the data where X represents the state visited and Y represents the time taken to get to that state from the previous. The color acts as a third dimension for the 2D scatter plot of the depth reached. The right scatter plot uses the same X and Y variables but with an additional third dimension Z for depth. This provides an approach that we feel offers a more practical way of visualizing the observable ascend and descend behavior. Both plots are populated with the same data.

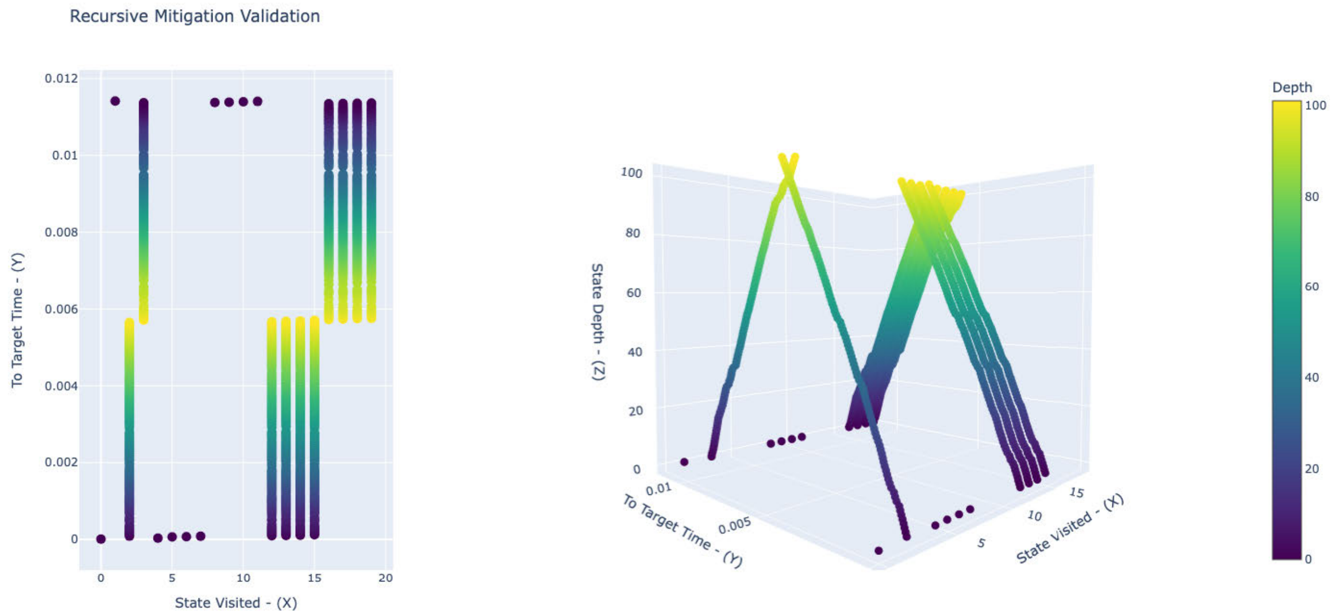


FIGURE 17. Recursive Validation in 3 Dimensions.

```
1 rule1: 'ALPHA' 'BRAVO' 'CHARLIE' rule2
2 'CHARLIE' 'KILO' 'FOXTROT' 'EOF';
3 rule2: 'INDIA' 'LIMA' 'NOVEMBER' 'MIKE'
4 'GOLF' '(' rule2 ')' 'OSCAR' 'CHARLIE';
```

LISTING. 3. Recursive test grammar.

What can be seen from the data in the 3D scatter plot is the rise as depth increases and we begin to ascend. This continues until our constraint is met, at which point it begins to fall. The peak reached before starting the descent is the moment in which the mitigation control begins working, which is 100. This was the constraint value chosen for this experiment. If no constraint was used there would be exponential growth as a result of the infinite recursion. This constraint signifies that regardless of the decision process used when faced with multiple path options if the constraint is met, it will forcefully complete the recursion using the stack to pop the follow state and exit the recursion normally until we feel ready to release that control. For the sake of brevity, only 5 examples have been chosen to highlight their proper closures, see Table 14. Two properties were used from the generation metadata showing our recursive corrections match the depth reached.

VI. LIMITATIONS AND FUTURE WORK

The proposed framework is able to successfully perform its intended task. However, it is recognized that there are some limitations and room for improvement. They can be summarized as follows.

A. SEMANTICS

Our proposal generates syntactically valid SQL queries. This does not always equate to a coherent query for the intended purpose. To achieve semantics validity, a required knowledge of the concrete use case being addressed by the SQL query generated. A way to address this is by implementing use-case-specific semantic predicate actions on the grammar, coupled with the insertion of custom action logic for use-case-specific

TABLE 14. Recursive Generations validated against generation metadata.

Generated String	Depth Constraint	Recursive Stack Corrections
ALPHA BRAVO CHARLIE GOLF HOTEL ( ) INDIA JULIETT DELTA ECHO FOXTROT	1	1
ALPHA BRAVO CHARLIE GOLF HOTEL (GOLF HOTEL ( ) INDIA JULIETT) INDIA JULIETT DELTA ECHO FOXTROT	2	2
ALPHA BRAVO CHARLIE GOLF HOTEL (GOLF HOTEL (GOLF HOTEL ( ) INDIA JULIETT) INDIA JULIETT) INDIA JULIETT DELTA ECHO FOXTROT	3	3
ALPHA BRAVO CHARLIE GOLF HOTEL (GOLF HOTEL (GOLF HOTEL (GOLF HOTEL ( ) INDIA JULIETT) INDIA JULIETT) INDIA JULIETT) INDIA JULIETT DELTA ECHO FOXTROT	4	4
ALPHA BRAVO CHARLIE GOLF HOTEL (GOLF HOTEL (GOLF HOTEL (GOLF HOTEL (GOLF HOTEL ( ) INDIA JULIETT) INDIA JULIETT) INDIA JULIETT) INDIA JULIETT) INDIA JULIETT DELTA ECHO FOXTROT	5	5

decisions. Another approach is to make use of AI to drive the generation of sentences to infer the semantics. These are clearly areas yet to be explored.

B. RECURSION

Recursion can be extremely beneficial for complex SQL generations, for example, for generating nested sub-queries. However, unless mitigated, it can lead to scenarios where recursion nested within loops cascades beyond a recovery point. In our approach, when repetitive rules are used using the Kleene plus or Kleene Star quantifiers, counters are deployed to monitor such repetitions. Finding the sweet spot for the optimal boundaries to cut the recursion depth is difficult. Currently, we are providing such cutting threshold as a fixed constraint which can be seen as a limitation. We believe that a more robust heuristic-based implementation could be further explored to monitor how depth is influencing the transitioning between states.

C. RANDOMISED TRANSITION SELECTION

As previously discussed, our current implementation when faced with multiple paths during ATN traversal is through a random choice. However, using such a system for transitions can result in inconsistencies and added ambiguity in some generations. This is where we consider that a clear next step is the creation of a weight-based solution to help drive such selections permitting an AI solution to help search the optimal search space for the use case being addressed.

VII. CONCLUSION

In In this paper we proposed a novel framework to aid in generating syntactically valid languages from EBNF grammars. We highlighted steps for creating generative-friendly grammar which helped reduce the risk of recursive and combinatory runaways through the modification and/or removal of said grammar rules. We concurrently developed and implemented a generation prototype using ANTLR4 and its ATN components of both lexer and parser, highlighting possible areas of interest for AI integration including the capability to utilize lexer rules for generation by means of switching ATNs. We were also able to provide pragmatic solutions for controlling randomized generations through the implementation of stacks. This would alleviate the concerns of recursive rule traversal, including correcting the exit transition to ensure syntactic validity. Using the proposed prototype we were able to generate 3000 SQL statements, with an average execution time of **0.307ms**.

We analyzed the syntactic validity of the generated SQL statements and found some problems resulting from lexer generations and certain recursive rules.

Furthermore, extended generator experiments were conducted to ascertain the effect grammar complexity has on the time taken to generate. The results highlighted for every 250 non-terminals traversed, a time penalty of **0.0027546ms** was incurred. We also validated our recursive mitigation solution pertaining to recursive production rules and provided visual validation of the ascend and descend behavior reacting to our pre-determined constraint.

We understand our proposal is not a one-shot solution regarding language generation and recursion mitigation. Instead, it is a step in the right direction by helping other researchers to see the areas of interest for exploration regarding generating language. We know that the complexity of certain grammars can make it time -consuming to modify them, hence the justification for leaving recursive rules in when generating SQL. This helped align it with real-world expectations. From a goal-oriented perspective, the tool performed really well and was extremely quick when outputting generated SQL statements. We had zero recursive and combinatory runaways due to the implementation of the proposed control logic. Our future work aims to fix any persistent problems and improve upon the framework. Moreover, inspired by the potentially wide applicability of the proposed framework, we aim to focus on a cybersecurity use case for detecting different kinds of cyber- attacks through the creation of novel metrics using SQL and AI.

REFERENCES

[1] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley, "Evolution through large models," 2022, *arXiv:2206.08896*.

[2] *Gpt-4 Technical Report*, OpenAI, San Francisco, CA, USA, 2023.

[3] R. Thoppilan et al., "LaMDA: Language models for dialog applications," 2022, *arXiv:2201.08239*.

[4] (Jan. 5, 2021). *Dall·e: Creating Images From Text*. [Online]. Available: https://openai.com/research/dall-e

[5] P. von Platen, S. Patil, A. Lozhkov, P. Cuenca, N. Lambert, K. Rasul, M. Davaadorj, and T. Wolf. (2022). *Diffusers: State-of-the-art Diffusion Models*. [Online]. Available: https://github.com/huggingface/diffusers

[6] A. Meier and M. Kaufmann, *SQL and NoSQL Databases* (Database Languages). Cham, Switzerland: Springer, 2019, pp. 85–121.

[7] PlantUML. *PlantUML: Ebnf*. Accessed: Apr. 10, 2023. [Online]. Available: https://plantuml.com/ebnf

[8] S. de Agostlno and R. Greenlaw, "Automata theory," in *Encyclopedia of Information Systems*, H. Bidgoli, Ed. New York, NY, USA: Elsevier, 2003, pp. 47–63. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B0122272404000046

[9] W. A. Woods, "Augmented transition networks for natural language analysis," Comput. Lab., Harvard Univ., Cambridge, MA, USA, Tech. Rep., CS-1, 1969.

[10] T. Parr, S. Harwell, and K. Fisher, "Adaptive LL(*) parsing: The power of dynamic analysis," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.* New York, NY, USA: Association for Computing Machinery, Oct. 2014, pp. 579–598, doi: 10.1145/2660193.2660202.

[11] C. Sugandhika and S. Ahangama, "Heuristics-based SQL query generation engine," in *Proc. 6th Int. Conf. Inf. Technol. Res. (ICITR)*, Dec. 2021, pp. 1–7.

[12] R. J. L. John, D. Bacon, J. Chen, U. Ramesh, J. Li, D. Das, R. Claus, A. Kendall, and J. M. Patel, "DataChat: An intuitive and collaborative data analytics platform," in *Proc. Companion Int. Conf. Manag. Data*, 2023, pp. 203–215. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85162911612&doi=10.1145%2f3555041.3589678&partnerID=40&md5=a38430a47554d94b2968e6a952e3bf0c

[13] A. Anisyah, T. E. Widagdo, and F. N. Azizah, "Natural language interface to database (NLIDB) for decision support queries," in *Proc. Int. Conf. Data Softw. Eng. (ICoDSE)*, Nov. 2019, pp. 1–6.

[14] S. Sargsyan, J. Hakobyan, M. Mehrabyan, R. Mkoyan, V. Sahakyan, V. Melkonyan, M. Arutunian, A. Fahradyan, and A. Avetisyan, "Advanced grammar-based fuzzing," in *Proc. Ivannikov Memorial Workshop (IVMEM)*, Sep. 2022, pp. 61–64.

[15] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, and S. Asryan, "Grammar-based fuzzing," in *Proc. Ivannikov Memorial Workshop (IVMEM)*, May 2018, pp. 32–35.

[16] F. Palmas, J. Raith, and G. Klinker, "A novel approach to interactive dialogue generation based on natural language creation with context-free grammars and sentiment analysis," in *Proc. IEEE 20th Int. Conf. Adv. Learn. Technol. (ICALT)*, Jul. 2020, pp. 79–83.
[17] J. O. Ryan. (2017). *Expressionist*. [Online]. Available: https://github.com/james-owen-ryan/expressionist
[18] J. P. Pires and F. B. E. Abreu, "Knowledge discovery metamodel-based unit test cases generation," in *Proc. IEEE 11th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2018, pp. 432–433.
[19] (Dec. 2016). *About the Knowledge Discovery Metamodel Specification Version 1.4*. [Online]. Available: https://www.omg.org/spec/KDM/1.4/About-KDM
[20] (2022). *About Xunit. Net*. [Online]. Available: https://xunit.net/
[21] J. Wang, P. Zhang, L. Zhang, H. Zhu, and X. Ye, "A model-based fuzzing approach for DBMS," in *Proc. 8th Int. Conf. Commun. Netw. China (CHINACOM)*, Aug. 2013, pp. 426–431.
[22] P. Parikh, O. Chatterjee, M. Jain, A. Harsh, G. Shahani, R. Biswas, and K. Arya, "Auto-query—A simple natural language to SQL query generator for an e-learning platform," in *Proc. IEEE Global Eng. Educ. Conf. (EDUCON)*, Mar. 2022, pp. 936–940.
[23] J. G. Meyer, R. J. Urbanowicz, P. C. N. Martin, K. O'Connor, R. Li, P.-C. Peng, T. J. Bright, N. Tatonetti, K. J. Won, G. Gonzalez-Hernandez, and J. H. Moore, "ChatGPT and large language models in academia: Opportunities and challenges," *BioData Mining*, vol. 16, no. 1, p. 20, Jul. 2023, doi: 10.1186/s13040-023-00339-9.
[24] J. Cámara, J. Troya, L. Burgueño, and A. Vallecillo, "On the assessment of generative AI in modeling tasks: An experience report with ChatGPT and UML," *Software and Systems Modeling*, vol. 22, no. 3, pp. 781–793, 2023. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85160072219&doi=10.1007%2fs10270-023-01105-5&partnerID=40&md5=d8583d705f4bf2c7e97fe11bfb1b00c8
[25] B. D. Goel, A. Gupta, S. C.Batra, and Hunar, "MUCE: A multilingual use case model extractor using GPT-3," *Int. J. Inf. Technol.*, vol. 14, no. 3, pp. 1543–1554, 2022, doi: 10.1007/s41870-022-00884-2.
[26] X. Tang, H. Gao, and J. Gao, "Knowledge-based questions generation with seq2seq learning," in *Proc. IEEE Int. Conf. Prog. Informat. Comput. (PIC)*, 2018, pp. 180–184.
[27] *Overview—Seq2Seq*. Accessed: Jun. 18, 2023. [Online]. Available: https://google.github.io/seq2seq/
[28] Z. Gao, W. Dong, R. Chang, and C. Ai, "The stacked Seq2seq-attention model for protocol fuzzing," in *Proc. IEEE 7th Int. Conf. Comput. Sci. Netw. Technol. (ICCSNT)*, Oct. 2019, pp. 126–130.
[29] D. Lu, J. Fei, L. Liu, and Z. Li, "A GAN-based method for generating SQL injection attack samples," in *Proc. IEEE 10th Joint Int. Inf. Technol. Artif. Intell. Conf. (ITAIC)*, vol. 10, Jul. 2022, pp. 1827–1833.
[30] DukeNLIDB. (2017). *Natural Language Interface to Databases (NLIDB)*. Accessed: Apr. 8, 2023. [Online]. Available: https://github.com/DukeNLIDB/NLIDB

CHRISTOPHER TROY received the Bachelor of Science (B.Sc.) degree (Hons.) in web and mobile development, in 2019, and the Master of Science (M.Sc.) degree (Hons.) in cyber security, in 2021. He is currently pursuing the Ph.D. degree with the University of the West of Scotland, U.K. His research interests include software development, advancements in AI, network security, and natural language processing. He has been awarded the court medal twice from the University of the West of Scotland for showing the highest academic standing in both the B.Sc. and M.Sc. courses. He is also undertaking a scholarship from the Carnegie Trust for the Universities in Scotland.

SEAN STURLEY (Member, IEEE) received the M.Sc. and M.Eng. degrees in cyber security. He is currently a Senior Lecturer in cyber security with the School of Engineering and Computing, University of the West of Scotland, with a specialism in network forensics. After 20 years of industrial experience in banking, finance, and retail network administration and security, he has sought to apply this wealth of knowledge and experience in the higher education sector by the design and delivery of a practically focused the M.Sc. and M.Eng. degrees. He is also a network security consultant to several charitable bodies, a Panel Member of a major U.K. Cyber Security Accreditation, and an Editor of a major European Computer Science Conference.

JOSE M. ALCARAZ-CALERO (Senior Member, IEEE) received the B.Eng., M.Eng., and Ph.D. degrees. He is currently a Full Professor in networks and security with the School of Engineering and Computing, University of the West of Scotland, U.K. He is also a Representative Member of the EU 5G-PPP Technological Board, the NATO Working Group IST-118, and the Internet Technical Committee. He has been involved in international research projects totaling more 20m EUR. He is also a Co-Technical Coordinator of the H2020 5G PPP SELFNET Consortium (6.8m EUR). He shifted to the industrial side of research working four years for the Cloud and Security Laboratory, Hewlett-Packard Research Laboratories, U.K. He has published more than 100 articles in journals, magazines, books, and conferences in computer science and telecommunications, including more than 15 international patents and intellectual properties rights. He is also a FHEA. He is also an Associate Editor of a number of prestigious journals, such as *IEEE Communication Magazine* and the Chair in a number of prestigious conferences, such as IEEE TrustCom and IEEE ATC. He has received a number of international awards, such as the Best Ph.D. Award on Computer Science and Outstanding Chair Award.

QI WANG is currently a Full Professor with the School of Computing, Engineering and Physical Sciences, University of the West of Scotland (UWS), U.K. He has served as a Board Member of the Technology Board for EU Fifth-Generation Public Private Partnership (5G PPP), a member for several 5G PPP Working Groups (Architecture and Software Networking), EU 6G Industry Association (6G-IA), Scotland's Developing AI and AI Enabled Products and Services Working Group, and International Telecommunication Union (ITU) Focus Groups of ML5G and Autonomous Networks, and an Academic Member of ITU and European Telecommunications Standards Institute (ETSI). He was the Winner of U.K. Times Higher Education Awards 2020-Knowledge Exchange/Transfer Initiative of the Year Award and the Winner of 2020 Scotland Centre for Engineering Education and Development Industry Awards—Innovation Award.