Final Project:
Special-purpose programming language (DSL)

Salazar Andrés - 20202020043
Panqueva Miguel - 20201020174

CONTENTS

LIST OF FIGURES

Final Project: Special-purpose programming language (DSL)

Abstract—**This paper presents the design and implementation of a special-purpose programming language (DSL) for a domain-specific compiler. The compiler processes SQL scripts, translating them into an intermediate representation that facilitates further analysis and transformation. The primary goal was to create a tailored DSL that optimizes parsing, improves error handling, and enhances flexibility compared to general-purpose alternatives. Several approaches were evaluated to refine syntax, semantics, and execution efficiency. The final implementation successfully compiles SQL code while ensuring correctness and adaptability. Future improvements may focus on extending the DSL's capabilities, optimizing performance, and broadening its compatibility with different SQL dialects.**

I. INTRODUCTION

Relational databases are widely used to store and manage structured data in various applications. However, interacting with these databases often requires proficiency in SQL, which can be a barrier for users without technical expertise. To address this challenge, we developed a special-purpose programming language (DSL) that facilitates database design by transforming natural language instructions into SQL scripts.

This approach simplifies the creation and modification of relational schemas, making database management more intuitive. The generated SQL scripts can also be used to produce relational diagrams, offering a visual representation of the database structure. By automating this process, our DSL enhances accessibility and efficiency in database modeling.

II. REQUIREMENTS ANALYSIS

Managing relational databases typically involves writing complex SQL queries and scripts to define tables, establish relationships, and manipulate data. For users unfamiliar with SQL syntax, this can be a cumbersome task. The primary goal of our DSL is to bridge this gap by allowing users to describe database structures using natural language statements, which are then converted into SQL scripts.

Our DSL provides an abstraction layer over SQL, enabling users to define entities, attributes, and relationships without directly interacting with SQL code. Once translated, the resulting scripts can be executed to create and manage relational schemas. Additionally, these scripts serve as input for generating relational diagrams, visually representing the database structure for better comprehension and documentation.

By reducing the complexity of manual SQL writing and automating diagram generation, our DSL offers a more accessible and streamlined approach to relational database management, benefiting both developers and non-technical users.

III. LIMITATIONS AND CONSIDERATIONS

While our DSL successfully defines entities and attributes with properties such as primary keys, NULL, and NOT NULL constraints, it lacks full SQL execution capabilities. Specifically, the system does not support running queries or handling JOIN operations, limiting its ability to validate relationships beyond structural definitions.

Additionally, although foreign keys can be identified as attributes, they are not enforced as actual constraints within the schema. This means that while relationships between entities are represented, referential integrity is not validated at the database level.

These limitations affect the practical application of the generated SQL scripts, as manual adjustments would be required for full database enforcement. Future iterations could integrate query execution and constraint validation to enhance accuracy and usability.

IV. DESIGN OF SPECIAL-PURPOSE PROGRAMMING LANGUAGE(DSL)

The design of our DSL will begin with the identification of the lexicon to be used, classifying tokens and their corresponding lexemes. This step will ensure a clear structure for interpreting natural language instructions and converting them into SQL scripts.

To achieve this, we will define token types using regular expressions, specifying the valid formats and possible variations for each element. This will allow for accurate token recognition while maintaining flexibility in how users write instructions. Additionally, we will establish rules for validating tokens entered by the user, ensuring that only syntactically correct terms are accepted.

As part of this process, we will determine which elements should be considered essential and which can be omitted for convenience. For example, whitespace, line breaks, and comments will be ignored during parsing to streamline processing without affecting meaning. Once validated, the recognized tokens will be stored for further processing, forming the basis for generating SQL scripts and, ultimately, relational diagrams.

After lexical analysis, we will implement syntactic analysis to ensure that user-defined instructions follow a structured format. This step will involve parsing the extracted tokens and verifying that they conform to the grammar rules of our DSL.

The semantic analyzer will ensure the correctness of entity and relationship definitions by verifying their logical structure and consistency. It will validate entities by detecting duplicate names, ensuring that attributes within an entity are unique, and checking for conflicts in attribute properties, such as the coexistence of primary and non-primary keys. Additionally, it will evaluate relationships by verifying that the involved entities exist and that the defined cardinality is valid. Any detected errors will be reported with contextual information, allowing inconsistencies to be corrected before generating SQL scripts.

The syntactic analyzer will process the tokenized input by identifying whether the user is defining an entity or a relationship. It will then validate the structure of attributes, properties, and relationships according to predefined rules. The parser will include functions to match expected tokens, validate attribute lists, and enforce correct relationship definitions. Additionally, error handling mechanisms will provide clear feedback when an invalid structure is detected.

By ensuring that inputs adhere to a consistent syntax before generating SQL scripts, this step will enhance the accuracy and reliability of database design, reducing the risk of structural errors.

The SQL code generation process will ensure that entities and relationships are correctly translated into SQL-compliant structures. The CodeGenerator class will create CREATE TABLE statements for entities, considering attribute properties such as primary keys, NOT NULL constraints, and auto-incrementation. It will also handle foreign key constraints by identifying relationships and enforcing referential integrity through ALTER TABLE statements. This validation will ensure that SQL syntax and database constraints are correctly defined, reducing potential schema errors.

Once the SQL script is generated, the diagram generator will extract table definitions and foreign key relationships. It will parse CREATE TABLE and ALTER TABLE statements to construct a graph-based representation using NetworkX. Each entity will be displayed as a node containing attribute details, while foreign key relationships will be represented as directed edges. The diagram will help visualize the database structure, including dependencies between entities, improving clarity and validation of the schema before deployment.

V. RESULTS

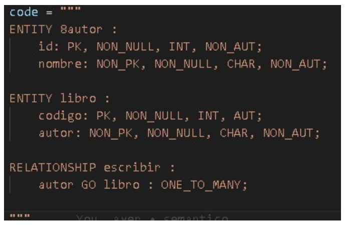
The implementation of our DSL followed a structured process, ensuring that natural language instructions were correctly interpreted and transformed into SQL scripts.

First, during lexical analysis, tokens were extracted from user input based on predefined regular expressions. This step allowed for the identification of keywords, attributes, and entity names while discarding unnecessary elements such as whitespace and comments. If an invalid token was encountered, such as an unrecognized keyword or an incorrectly formatted attribute, a lexical error was triggered, providing immediate feedback.

Next, syntactic analysis verified that the extracted tokens conformed to the expected structure. The parser categorized inputs as entity or relationship definitions, ensuring correct attribute formatting and enforcing predefined grammar rules. If an instruction deviated from the expected format—such as a missing colon (:) in an entity definition or an incorrect relationship declaration—a syntax error was raised, guiding users to correct their input.

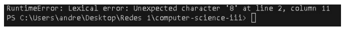
Following this, semantic analysis validated the logical consistency of the defined entities and relationships. It checked for duplicate entity names, unique attributes within entities, and valid cardinality definitions. If an entity referenced an undefined attribute or if an invalid relationship cardinality was specified, a semantic error was reported with detailed feedback, allowing users to resolve inconsistencies before proceeding.

To illustrate how errors are handled at different stages of the process, the following images showcase examples of lexical, syntactic, and semantic errors detected during input validation. Each error message provides specific feedback to help users correct mistakes and refine their database definitions.



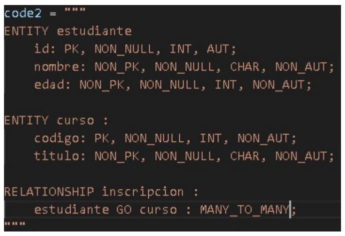
```
code = """
ENTITY 8autor :
    id: PK, NON_NULL, INT, NON_AUT;
    nombre: NON_PK, NON_NULL, CHAR, NON_AUT;

ENTITY libro :
    codigo: PK, NON_NULL, INT, AUT;
    autor: NON_PK, NON_NULL, CHAR, NON_AUT;

RELATIONSHIP escribir :
    autor GO libro : ONE_TO_MANY;

"""        You, ayer • semantico
```

Fig. 1. The lexical analyzer is expected to find an error since no identifier can start with a number.



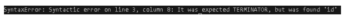
```
RuntimeError: Lexical error: Unexpected character '8' at line 2, column 11
PS C:\Users\andre\Desktop\Redes 1\computer-science-iii> 
```

Fig. 2. lexical analyzer result.



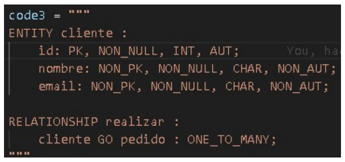
```
code2 = """
ENTITY estudiante
    id: PK, NON_NULL, INT, AUT;
    nombre: NON_PK, NON_NULL, CHAR, NON_AUT;
    edad: NON_PK, NON_NULL, INT, NON_AUT;

ENTITY curso :
    codigo: PK, NON_NULL, INT, NON_AUT;
    titulo: NON_PK, NON_NULL, CHAR, NON_AUT;

RELATIONSHIP inscripcion :
    estudiante GO curso : MANY_TO_MANY;
"""
```

Fig. 3. A syntax error is expected on line 3 due to the lack of the TERMINATOR ":".



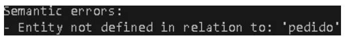
```
SyntaxError: Syntactic error on line 3, column 8: It was expected TERMINATOR, but was found 'id'
```

Fig. 4. Parser result.



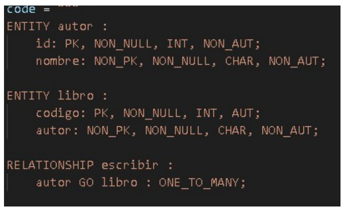
```
code3 = """
ENTITY cliente :
    id: PK, NON_NULL, INT, AUT;        You, ha
    nombre: NON_PK, NON_NULL, CHAR, NON_AUT;
    email: NON_PK, NON_NULL, CHAR, NON_AUT;

RELATIONSHIP realizar :
    cliente GO pedido : ONE_TO_MANY;
"""
```

Fig. 5. A semantic error is expected due to the non-existence of the "pedido" entity.



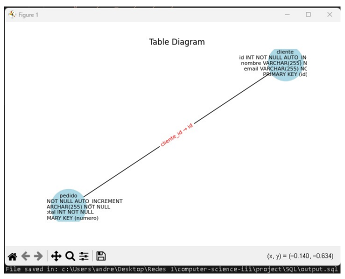
```
Semantic errors:
- Entity not defined in relation to: 'pedido'
```

Fig. 6. Semantic analyzer result.



```
code =
ENTITY autor :
    id: PK, NON_NULL, INT, NON_AUT;
    nombre: NON_PK, NON_NULL, CHAR, NON_AUT;

ENTITY libro :
    codigo: PK, NON_NULL, INT, AUT;
    autor: NON_PK, NON_NULL, CHAR, NON_AUT;

RELATIONSHIP escribir :
    autor GO libro : ONE_TO_MANY;
```

Fig. 7. Correct use of the rules final result if the proposed code is correct.



```
File saved in: c:\Users\andre\Desktop\Redes 1\computer-science-iii\project\SQL\output.sql
```

Fig. 8. Result generated by a correct text.

Once validated, the SQL code generation phase transformed the structured definitions into SQL statements. The system generated CREATE TABLE statements, incorporating primary key assignments and attribute constraints such as NULL and NOT NULL. The resulting SQL script was then saved as a .sql file, allowing further use in database environments.

Finally, the diagram generation phase used the SQL script to visualize the database structure. The extracted table definitions and relationships were processed to generate a relational diagram that displayed entities, attributes, relationships, and cardinalities. This graphical representation facilitated a clear understanding of the schema, ensuring that database relationships were accurately captured before deployment.

VI. Conclusions

This project successfully demonstrated the feasibility of transforming natural language input into structured SQL code. Through lexical, syntactic, and semantic validation, we ensured that user-defined entities, attributes, and relationships were correctly structured before generating the corresponding database schema. The system effectively detected and classified errors at each stage, providing meaningful feedback to guide users in refining their input.

The structured approach facilitated a smooth transition from high-level conceptual design to SQL implementation, reducing the manual effort needed to define relational databases. However, the lack of advanced SQL constraints, such as foreign key validation and JOIN operations, restricted the completeness of the generated schema, necessitating further refinement for real-world deployment.

A comparative assessment of different error-handling techniques showed that syntactic validation was the most critical phase, as it prevented invalid structures from propagating into later stages. Clearly defining the start and end of the input text played a crucial role in this phase, ensuring that the system correctly interpreted the intended structure and avoided parsing ambiguities. Meanwhile, semantic checks ensured logical coherence but required predefined rules, which could be further expanded for improved adaptability.

To summarize, this approach proved to be a viable and efficient method for automating database schema generation while maintaining error control. Future improvements could enhance SQL capabilities and extend constraint validation to further refine the generated output.

To enhance the system's effectiveness, future improvements should focus on incorporating advanced SQL constraints, such as foreign key validation and JOIN operations, to generate a more complete and functional database schema. Additionally, refining error-handling mechanisms and expanding semantic validation rules would improve adaptability and accuracy. Further research could explore integrating natural language processing techniques to enhance user input interpretation, making the DSL more intuitive and robust for database design.

References

[1] D. Castelvecchi, "DESARROLLO DE UN LENGUAJE DE DOMINIO ESPECIFICO (DSL) GRAFICO PARA EL MODELAMIENTO DE BASES DE DATOS ESPACIALES" 2019. [Full Text]. Available: Universidad Distrital Francisco José De Caldas, https://repository.udistrital.edu.co/items/ad5573f4-db70-4134-bb4d-df6ffbcb1f52. Accessed on: Feb. 14, 2025.

[2] "Building Your Own Custom DSL: A Comprehensive Guide," Oct. 20 2024. [Online]. Available: https://medium.com/@robertdennyson/building-your-own-custom-dsl-a-comprehensive-guide-9be7bb70524d. [Accessed: Feb. 14, 2025].

[3] "DSL for the uninitiated," Jul. 2011. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1965724.1965740. [Accessed: Feb. 14, 2025].