DOMAIN SPECIFIC LANGUAGE FOR DATA AND FORMULAS VISUALIZATION

Roman GUSEV*, Tudor POPOV, Andrei CERNÎȘOV, Dorin MALANCEA, Alexandr COVALENCO

Department of Software Engineering and Automatics, Group FAF-222, Faculty of Computers, Informatics, and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova

*Corresponding author: Roman Gusev, roman.gusev@isa.utm.md

Tutor/coordinator: **Irina COJUHARI**, conf. univ., dr., DISA

Abstract. *In a data-driven world, the visualization of complex datasets and mathematical formulas remains a critical challenge despite technological advancements. This article investigates these challenges and introduces a proposed solution in the form of a Domain-Specific Language, specifically for Data, such as Tabular Data from Excel Files or JSON Files, and Formulas, for example Mathematical Expressions, visualization. This language aims to address data and formula visualization in the form of different Graphical Representations, such as Bars or Graphs. Some of this paper's objectives are to describe the steps of the development of a Domain-Specific Language with the mentioned functionalities and such input mechanics, that everyone will easily understand how to work with it. Also, this paper aims to provide, through the proposed language, a better and easier way to visualize information and get an extensive graph representation, along with tabular representation, of the Formulas and Data that are provided by the user.*

Keywords: *ANTLR, Grammar, Graph, Lexer, Mathematical Expressions, Parser.*

Introduction

In an increasingly data-driven world, the ability to represent information graphically holds significant importance across various sectors including finance, marketing, healthcare, and data science. Motivated by the recognition of persistent difficulties in creating concise and expressive visualizations despite technological advancements, the development of this DSL aims to bridge the gap between complex data and accessible visualization, thereby enhancing the efficiency and effectiveness of data-driven decision-making processes [1].

Also, this paper aims to deliver this DSL's potential users a very user friendly and easy to understand Language, that will be helpful in their daily working routine or in their research projects. The relevance of this topic lies in its potential to simplify the process how data and formulas are visualized and understood, offering a higher level of abstraction and simplicity. The primary objective of this paper is to empower users across various domains to create visually compelling representations of their data and formulas through a proposed DSL.

Problem Description

In the field of data visualization, there are big hurdles that make it tough to show and analyze complex datasets and mathematical formulas properly. Even though technology keeps getting better, the tools we have now often can't meet the varied needs of users in different areas. Some of the main issues this DSL was designed to address are described further.

Managing large amounts of data can be tricky. It is hard to mix data from different places and make sure everything works together well. This is especially tough in projects that cover many subjects, where it's really important that data from different sources can work together smoothly for the analysis and visualization to be successful. As the scale of data grows larger, visualization tools often start to struggle, which makes people worry about whether these tools can handle really big datasets, especially when one needs to process and show data in real time. Plus, a lot of these

tools need a lot of computing power to show complex data, which means they're not very practical in places where a limited amount of computing resources is available.

Moreover, showing complex formulas, especially those with lots of variables or that change over time, is something many current visualization tools can't do well. This makes it hard to show real-life situations accurately. Many areas need specific ways to show formulas that standard tools just cannot offer. Without the ability to make these customizations, people often have to find complicated workarounds or simplify their models too much, which means the visualizations are not as accurate or helpful as they could be [2].

Another major problem is that many data visualization tools are hard to learn and use, which keeps a lot of people, especially those who are not experts, from using them. Lowering the entry barrier and making these tools easier to use for more people is crucial. In addition to that, a significant part of visualization tools does not do a good job of letting users interact with the data, which is a problem, because being able to dive into the data and play around with it leads to better insights and understanding.

This DSL is designed to overcome these challenges by creating a solution that fits the needs of data analysts, scientists, developers, students, business analysts and professionals in various fields. By making it easier to present data, offering better ways to combine different data sources, providing options to customize formula visualizations, making it more interactive and engaging for users, and improving how well it can handle large data sets and perform well, this DSL aims to help users get valuable insights from complex data and formulas in a more efficient and effective way.

Stakeholders

This Domain-Specific Language can cater to a broad spectrum of stakeholders and potential users, including data analysts, scientists, developers, business analysts, data engineers, product managers, educators, researchers, business executives, freelancers, consultants, data journalists, and UX/UI designers.

Data analysts and scientists form the backbone of data exploration and interpretation. This language can cater to their needs by providing an intuitive and efficient platform for manipulating and visualizing data.

Countability workers rely heavily on data visualization tools to analyze and interpret large volumes of numerical data efficiently and accurately. Whether they are working in finance, accounting, or any other field that requires meticulous data analysis.

Developers and programmers seek tools that seamlessly integrate data visualization into their coding practices. The language that is described in this paper can provide a solution to this need by offering a Domain-Specific Language that harmonizes coding logic with data representation.

Teachers and students seek tools that will help them in visualization of complex formulas in mathematics and that can provide extensive features to operate with them. At the same time, they are interested in working with large datasets and visualizing them in a suitable form for manipulation, with further adjustments.

For stock traders, having access to clear and concise data visualization tools is paramount for making informed investment decisions in an ever-changing market landscape. Stock traders can seamlessly integrate complex financial data into their analytical workflow, allowing them to create dynamic visualizations that highlight key metrics such as price movements.

Language Overview

This DSL for visualizing data and formulas is built with Python. Python was picked for a few key reasons - easy to use and clear code structure, wide range of tools and extensive features for data handling.

The DSL uses a data-driven execution model, starting tasks when the data it needs is ready. This fits well with the data centric nature of visualization tasks, letting users see their data visualized as soon as they put it in.

To provide extensive functionality and enhance the user experience, this project integrates several libraries:

1. NumPy: a go-to for scientific computing in Python, NumPy lets you work efficiently with complex data sets.
2. Pandas: built on NumPy, Pandas provides high-performance, easy-to-use data structures and data analysis tools, enabling intuitive data manipulation and preparation for visualization.
3. Matplotlib: a popular choice for data visualization in Python, Matplotlib lets users create both static and interactive charts, graphs and other kinds of visualizations.

To support the diverse data formats encountered in real-world scenarios, a range of data structures are utilized, including NumPy arrays, Pandas DataFrames, and classical Python data structures like lists, dictionaries, and sets. The DSL can handle a wide range of input formats, such as CSV, Excel, JSON, and text files, ensuring flexibility and compatibility with existing data sources.

Building on top of Python and its rich set of tools allows this project to balance simplicity with powerful features, helping users effectively visualize and gain insights from their data and formulas, without requiring a high level of technical expertise from the target audience.

Grammar overview

Grammar for a programming language is a collection of rules that specify how statements should be written in that language [3].

In programming languages, adherence to specific rules is imperative for code to function correctly. These rules are encapsulated within a framework known as grammar, which defines the language's syntax and structure. Broadly speaking, grammar outlines the rules that govern how valid expressions, statements, variables, and keywords are constructed within the language.

In a grammar for some DSL, Start Term refers to the initial non-terminal symbol from which the parsing of a language begins. It represents the starting point of the language's syntax tree or derivation process.

In this DSL, Grammar starts with this term:

Start Term – S = { <Program> }.

Terminal symbols are those that can occur in the outputs of a formal grammar's production rules but that the grammar's rules are unable to modify. Recursively applying the rules to a source string of symbols will typically result in a final output string that is exclusively made up of terminal symbols.

For this language, Terminal Terms are the following:

Terminal Terms – Vt = { Data, Formula, dataset, name, if, else, range, while, ReadFrom, ExportToFile, ExportToImage, VisualFormula, VisualData, graph, bar, pie, hist, png, jpg, csv, txt, json, excel, console, '', #, ;, :, ,, (,), _, -, [,], *, ^, log, sqr, sqrt, fact, +, -, ., ==, >, <, !=, >=, <=, {, }, /*, */, /, //, [a-z], [A-Z], [0-9] }.

Symbols that are not terminal can be swapped out. Another name for them would be syntactic variables. Formal grammar has a start symbol, which is a named member of the set of non-terminals from which all the language's strings can be generated by applying the production rules one after the other.

The set of terminal strings from which such a language can be derived is precisely the language defined by grammar.

In this language, Non-Terminal Terms are the following terms:

Non-Terminal Terms – Vn: { <Program>, <CommandsList>, <Command>, <IfStatement>, <WhileStatement>, <Comment>, <ReadCommand>, <ExportCommand>, <VisualizeCommand>, <VariableName>, <Formula>, <ReadFrom>, <VisualizeFormula>, <VisualizeData>, <Condition>, <VariableName>, <Expression>, <ReadFromFile>, <FormulaContent>, <ExportToFile>, <ExportToImage>, <VisualizeData>, <VisualizeFormula>,

<PathTo>, <PathName>, <ImageType>, <PlotType>, <VisualizationType>, <FileType>, <Operators>, <Digit>, <Integer>, <Float> }.

Since this DSL has a complex topic, Grammar was made as easy as possible for understanding for all users of the DSL we develop. It has several conceptualized Rules that will help the Language to be Tokenized and Parsed correctly.

Rules for the Grammar that will be presented below, follow classical Meta Notation for Grammar Description (See Table 1):

Table 1

Meta Notation for Grammar Description

Symbol	Meaning
<foo>	means foo is a nonterminal.
foo	(in bold font) means that foo is a terminal i.e., a token or a part of a token.
[x]	means zero or one occurrence of x.
x^*	means zero or more occurrences of x.
x^+	means one or more occurrences of x.
{ }	large braces are used for grouping.
|	separates alternatives.

The following description is the set of Instruction/Rules that this Domain Specific Language is based on:

Production Set – P = {

<Program> ::= <CommandsList>
<CommandsList> ::= { <Command>
| <IfStatement>
| <WhileStatement>
| <Comment> }$^+$
<IfStatement> ::= **if** (<Condition>) {<CommandsList>}
[**else** {<CommandsList>}];
<WhileStatement> ::= while (<Condition>) {<CommandsList>};
<Command> ::= <ReadCommand>;
| <ExportCommand>;
| <VisualizeCommand>;
<Comment> ::= /*{**a-zA-Z0-9** /.}$^+$ */
| #{**a-zA-Z0-9** /.}$^+$
<ReadCommand> ::= **Data** <VariableName> = <ReadFromFile>
| **Formula** <VariableName> = <FormulaContent>
<Condition> ::= <VariableName> <Expression> { <VariableName>
| <Digit> | <Integer> | <Float> }
<Expression> ::= == | != | > | < | >= | <=
<ReadFromFile> ::= **ReadFrom** (<PathTo>)
<ExportCommand> ::= **ExportToFile** (<PathTo>) <ExportToFile>
| **ExportToImage** (<PathTo>) <ExportToImage>
<ExportToFile> ::= **dataset** = (<VariableName>) **name** =
(<VariableName>.<FileType>)
<ExportToImage> ::= <PlotType>(<VariableName>) **name** =
(<VariableName>. <ImageType>)
<VisualizeCommand> ::= <VisualizeData> | <VisualizeFormula>
<VisualizeData> ::= VisualData (<VisualizationType>) dataset =
(<VariableName>)
<VisualizeFormula> ::= **VisualFormula** (<FormulaContent>) range =

({<Digit>| <Integer> | <Float> },{ <Digit> | <Integer> | <Float> })
| **VisualFormula** (<VariableName>) **range** = ({<Digit> | <Integer> | <Float> },{<Digit> | <Integer> | <Float> })
<VariableName> ::= {**a-zA-Z_/**}$^+${**a-zA-Z0-9_/**}*
<PathTo> ::= "<PathName>"
<PathName> ::= {**a-zA-Z0-9_/**}$^+$
<ImageType> ::= **png** | **jpg**
<FormulaContent> ::= {<VariableName> | <Operators> | (|)
| <Digit> | <Integer> | <Float>}$^+$
<VisualizationType> ::= **console** | <PlotType>
<PlotType> ::= **graph** | **bar** | **pie** | **hist**
<FileType> ::= **csv** | **text** | **json** | **excel**
<Operators> ::= * | ^ | **log** | **sqr** | **sqrt** | **fact** | - | +
<Digit> ::= {**0-9**}
<Integer> ::= [-]<Digit>$^+$
<Float> ::= <Integer>.<Digit>$^+$
}

Example Valid Code

In order to describe in a much better way how the syntax for this DSL is looking like, here is provided an example of a "program" (See Figure 1), that was parsed by a parser implemented using ANTLR.

```
Data tableData = ReadFrom("/path1/folder1/file.txt");
Formula formula = x^2.2 + sqrt(x);
```

Figure 1. Code Example - Data and Formulas Initialization

In this example is presented variable declaration and initialization, specifically - Data variable that is called "tableData" and is read from a ".txt" file, alongside with a Formula variable, called "formula", that stores the mathematical expression (See Eq. (1)):

$$formula = x^{2.2} + \sqrt{x} \quad (1)$$

Previously described code is a valid and correct program code that is fully parsed without encountering any errors, which is described in the next section.

Tokens

First step in every DSL is the tokenization, which is done by Lexer, also called Lexical Analyzer. The lexical analyzer defines how the contents of a file are broken into tokens, which is the basis for supporting custom language features [4]. In the above example are several tokens, such as: ID, ASSIGN or OPERATORS tokens, that are a set of Lexemes with an assigned meaning to them. Lexemes, on the other hand, are only some strings of characters known to be of a certain kind. In the following image (See figure 2) are the tokens that were extracted from the input code

```
Tokens:
[<DATA> ='Data']
[<ID> ='tableData']
[<ASSIGN> ='=']
[<READ_FROM> ='ReadFrom']
[<LPAREN> ='(']
[<PATH> ='"/path1/folder1/file.txt"']
[<RPAREN> =')']
[<SEMICOLON> =';']
[<FORMULA_T> ='Formula']
[<ID> ='formula']
[<ASSIGN> ='=']
[<ID> ='x']
[<OPERATORS> ='^']
[<FLOAT> ='2.2']
[<OPERATORS> ='+']
[<OPERATORS> ='sqrt']
[<LPAREN> ='(']
[<ID> ='x']
[<RPAREN> =')']
[<SEMICOLON> =';']
[<EOF_TOKEN> ='<EOF>']
```

Figure 2. Tokens - Valid Program Code Example

These Tokens give the language the possibility to give a meaning to the lexemes that it encounters in the process of tokenization of an input program, but in order to maintain a correct syntax structure of the code, it also requires another process to pass - Parsing.

Parse Tree

In order to parse the above program code (See Figure 1) and display it as a Tree, ANTLR, which is responsible for Parser and Lexer generation, can provide a better view of the so-called Parse Tree, which is a representation of the structure of a sentence or a string [5], based on the Tokens and the above rules (See section “Grammar overview”).

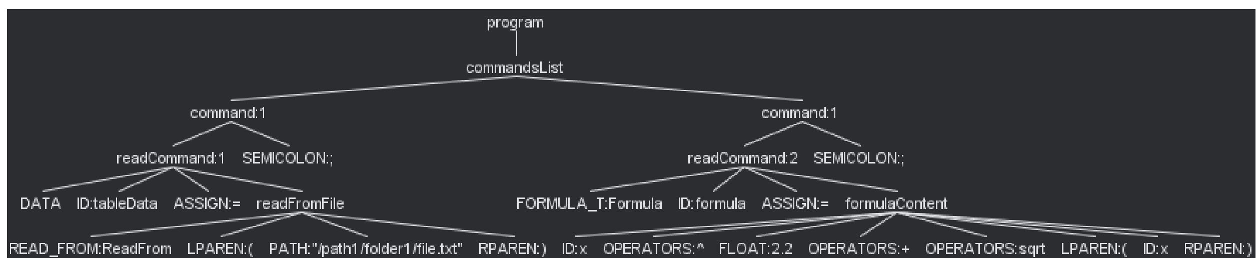

Figure 3. Parse Tree - Valid Program Code Example

As it may be seen, there are clearly two separate read commands - for Data and Formula declaration and initialization, and they both are parsed separately, based on their own rule, into “smaller” Tokens, until it reaches a Terminal symbol that cannot be derived further, indicating that there are no more Lexemes to parse and that there are no encountered syntactic errors in the Input Program Code example.

Conclusions

All things considered, a domain specific language for data and formulas visualization can simplify the process of investigation and analysis of different datasets and, at the same time, mathematical expressions. This specific feature can be used to simplify the work of different Data Analysts, financial workers, students and professors, that is based on the visual analysis of different data sources.

This DSL provides extensive possible utilities that may be used to transfer different datasets from one format to another, visualize it in a better and more comprehensive way, so that every user can easily understand what a piece of data is about.

At the same time, mathematicians and other scientific workers and researchers may be interested in the visualization of different mathematical formulas, that is a commonly-used way of proving different theories that include formulas.

Overall, this DSL's possible features may greatly improve the consistency, quality, and automation of data visualization procedures, making it an invaluable resource for all workers that are related with data analysis and mathematics.

References

[1] K. Smeltzer and M. Erwig, "A domain-specific language for exploratory data visualization," in *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, Boston MA USA: ACM, Nov. 2018, pp. 1–13. doi: 10.1145/3278122.3278138. Available: https://dl.acm.org/doi/10.1145/3278122.3278138. [Accessed: Apr. 01, 2024].

[2] "DSL for Business Intelligence Visualization," in Proceedings of 2016 the 6th International Workshop on Computer Science and Engineering, WCSE, 2016. doi: 10.18178/wcse.2016.06.066. Available: http://www.wcse.org/WCSE_2016/066.pdf. [Accessed: Apr. 03, 2024]

[3] Pietro, "What Is A Programming Language Grammar?" Compilers. Available: https://pgrandinetti.github.io/compilers/page/what-is-a-programming-language-grammar/. [Accessed: Apr. 12, 2024]

[4] "Lexical analysis," Wikipedia. Mar. 16, 2024. Available: https://en.wikipedia.org/w/index.php?title=Lexical_analysis&oldid=1214000313. [Accessed: Apr. 13, 2024]

[5] "Parse Tree in Compiler Design," GeeksforGeeks, Sep. 16, 2020. Available: https://www.geeksforgeeks.org/parse-tree-in-compiler-design/. [Accessed: Apr. 13, 2024]