DEVELOPING A DOMAIN-SPECIFIC LANGUAGE FOR GRAPHS

Andrei SAROV*, Danu MACRII, Victor BOSTAN, Vlad PRUTEANU

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: Andrei Sarov, andrei.sarov@isa.utm.md

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

Abstract. *This paper introduces the development of a domain-specific language tailored for graph illustration, focusing on its grammar, lexical parser, and functionality. The necessity for specialized tools in graph visualization, particularly in interpreting complex datasets, has led to the creation of this domain-specific language aimed at enhancing user interaction with graphical data. By detailing the implementation of the language's syntax and semantics, a comprehensive overview of how it simplifies the construction and manipulation of graph representations is provided. Furthermore, the practical applications of this domain-specific language in various fields are explored, demonstrating its potential to facilitate clearer data insights and improve decision-making processes. The paper culminates with a conclusion that reflects on the implications and future directions of this domain-specific language in graph illustration.*

Keywords: *Graph, Domain-specific language, Grammar, Syntax, Parser*

Introduction

Graphs serve as a foundational tool in mathematics and computer science, offering a robust framework for representing and analyzing complex relationships within various datasets. This paper focuses on the development of a domain-specific language (DSL) named GraphExpress, aimed at streamlining graph manipulation and analysis. GraphExpress leverages an intuitive syntax and a comprehensive suite of functionalities to make graph operations more accessible to a broader audience, enhancing user interaction with graphical data. The advent of GraphExpress represents a significant stride in graph visualization, particularly in interpreting complex datasets, by simplifying the construction and manipulation of graph representations and demonstrating its potential in facilitating clearer data insights and improving decision-making processes across various fields.

Domain Analysis

In increasingly interconnected world, the relevance of graph theory transcends a wide array of fields, offering profound insights into the structural and dynamic properties of complex systems. From social networks to biological systems, transportation networks, and beyond, graphs serve as a universal language for modeling relationships and analyzing interconnected data. The advent of the digital age, marked by the exponential growth of the internet and social media, has further amplified the applications of graph theory, making it an indispensable tool in navigating the complexity of modern life [1].

Graph theory, with its roots tracing back to Euler's bridges, has evolved from solving recreational mathematical puzzles to a significant area of mathematical research and application. This evolution reflects the growing complexity of the systems we seek to understand and optimize, from urban planning and logistics to the spread of information and diseases [2].

Graphs come in various forms, each suited to model different aspects of relationships within systems [3]:

- Simple Graphs: offer a foundational model for representing distinct entities and their interactions, widely applied in theoretical and practical domains.
- Multigraphs and Directed Acyclic Graphs (DAGs): cater to scenarios where relationships can vary in context or strength, or where processes with dependencies need to be modeled, such as in task scheduling and data processing pipelines.
- Trees and Forests: symbolize hierarchical structures, foundational in data structures, organizational charts, and decision-making processes.
- Bipartite and k-partite Graphs: useful in scenarios where entities are divided into distinct categories, such as matching jobs to applicants.
- Planar Graphs and Weighted Graphs: essential in geographic mapping and optimizing paths or flows in networks based on various criteria like cost or capacity.
- Hypergraphs: offer flexibility for modeling complex relationships beyond simple pairwise connections, applicable in areas such as co-authorship networks or actor collaborations.

Amidst this utility, DSLs emerge as pivotal in addressing the intricacies of graph-related operations, streamlining the creation, management, and analysis of graph data. DSLs, with their targeted functionality and simplified syntax, cater directly to the domain of graph theory, offering a more intuitive approach for users to engage with complex data structures. This shift towards specialized tools signifies a broader move to make technological solutions more accessible and efficient, particularly in fields where data complexity can quickly become overwhelming.

Despite the utility of graphs, the manual creation, management, and analysis of large-scale graphs present significant challenges. The complexity of these tasks grows exponentially with the size of the graph, introducing inefficiencies, potential for errors, and limitations in the application of graph theory to real-world problems. These challenges underscore the need for automated solutions to efficiently manage and analyze complex graph data.

Solution Proposal

In response to these challenges, GraphExpress is proposed as a DSL designed to simplify the creation, manipulation, and analysis of graphs. By offering an intuitive syntax and a comprehensive suite of functionalities, GraphExpress aims to democratize access to advanced graph operations. Its design focuses on automating the tedious aspects of graph management, incorporating powerful algorithms for analysis, and facilitating data visualization and integration.

Table 1

SWOT Analysis of GraphExpress

Strengths	Intuitive syntax, Comprehensive tools, Automated construction & maintenance, Integrated visualization
Weaknesses	Narrow focus, Initial learning curve, Performance limits for large datasets
Opportunities	Growing data needs, Role in AI and ML applications, Educational resource
Threats	Competitive market, Rapid technological changes, Need for strong user community

Lexical Considerations

In constructing GraphExpress, a set of lexical considerations was meticulously formulated to enhance clarity, ensure consistency, and maintain functionality within the language syntax. These guidelines are crucial for the effective parsing and interpretation of the DSL scripts.

GraphExpress is designed to be case-sensitive. This distinction applies to both keywords and identifiers, which means 'draw' is considered different from 'Draw' or 'DRAW', and similarly, 'Node' differs from 'node'. This sensitivity impacts how keywords are recognized and how identifiers are differentiated, ensuring precise interpretation of language constructs.

Certain terms are reserved as keywords within our grammar, including 'draw', 'Binary', and 'Bipartite'. These keywords play pivotal roles in the language syntax and semantics, thus they are prohibited from being used as identifiers for nodes or other elements within the DSL scripts. This reservation prevents ambiguity and enhances the structural clarity of graph definitions.

In GraphExpress, comments are implemented as single-line comments starting with '//' and extending to the end of the line, allowing for annotations and explanations within scripts. This approach ensures clarity and maintainability, as users can include descriptive notes directly alongside their code.

Identifiers are essential for naming graph nodes and must start with an alphabetic character, followed by a combination of alphanumeric characters. This requirement ensures that all node names are distinguishable and adhere to typical programming naming conventions, facilitating clear and meaningful graph representations.

In our GraphExpress, numerical literals, especially when representing weights in connections between nodes, are restricted to non-negative decimal digits (0-9) and can include decimal points for floating-point values. This approach aligns with the grammar's simplicity and directness, catering to the common use cases in graph specifications.

Specific symbols such as '-', '<', '>', '*', and '.' are endowed with particular roles within our DSL, primarily concerning the definition and attributes of node connections. Their utilization is bound to explicit patterns established in the grammar, ensuring the syntactic integrity and interpretative clarity of the language.

Creating a Grammar

This section details the formal grammar for this DSL, designed to provide a structured way to represent various types of graphs. The grammar is defined using a series of nonterminal and terminal symbols, production rules, and meta-notation to describe how users can construct graph descriptions.

The start symbol of our grammar is ⟨program⟩, which sets the foundation for any graph description. A program comprises a series of commands followed by the draw keyword, indicating the end of the graph specification.

Terminal symbols:

- <, -, >: these symbols are used to establish connections between nodes in the graph. They represent different types of edges, such as undirected (-), directed towards (>), and directed from (<).
- draw: this keyword signifies the end of the graph description, prompting the visualization of the defined graph.
- *: this symbol indicates that a node is a final node, akin to final states in finite automata, affecting its representation in the graph.
- -->: this symbol denotes the starting node of the graph, similar to the initial state in finite automata.
- Binary, Bipartite: these specify the type of the graph being defined, altering how the graph is interpreted and displayed.
- ., representing the decimal point used in floating-point numbers within graph weights.

Start = {<program>}

V_T = {-, <, >, draw, *, -->, Binary, Bipartite,.}

V_N = {<program>, <set of commands>, <type>, <connection list>, <comment>
<connection>, <node>, <start>, <final>, <number>, <alpha>, <digit>, <char>}

P = {

<program> → <set of commands> **draw**

<set of commands> → [<type>] <connection list>

<type> → **Binary | Bipartite**

<connection list> → [<comment>]<connection>[comment][<connection list>] | ε

<comment> → // <char>*

<connection> → [<start>]<node>[<]-[<number>]-[>]<node>

<node> → <alpha><char>*[<final>]

<start> → -->

<final> → *

<number> → <digit>+ | <digit>+.<digit>+

<alpha> → A | B … Z | a | b … z

<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<char> → A |B … Z |a |b … z | 0 | 1 … 9

}

Figure 1. GraphExpress grammar

Non-terminal symbols:

- ⟨program⟩: represents the entire graph description, consisting of a set of commands terminated by the draw keyword.
- ⟨set of commands⟩: may start with a graph type (Binary or Bipartite) indicating the nature of the graph. If no type is specified, the graph is treated as a standard graph. It also includes a mandatory ⟨connection list⟩, detailing the graph's connections.
- ⟨connection list⟩: details the user-defined connections between nodes. It allows an arbitrary number of connections and incorporates optional comments. Defined recursively, it enables a sequence of connections, interspersed with comments, ending when the list is empty (denoted by ε).
- ⟨comment⟩: begins with // and can be followed by any characters except newline, allowing users to include explanatory text within their graph descriptions.
- ⟨connection⟩: defines the connections between nodes, which may include optional start symbols (-->) and must include two nodes. Connections can represent different types of relationships between nodes, including unidirectional, bidirectional, or weighted edges.
- ⟨node⟩: represents the identifiers for graph nodes. A node starts with an alphabetic character and can be followed by additional alphanumeric characters. A node can be marked as final by appending *.
- ⟨number⟩: describes numbers used in the graph, which can be integers or decimals, allowing for the specification of weights in the graph connections.
- ⟨alpha⟩, ⟨digit⟩, ⟨char⟩: represent alphabetical characters, numerical digits, and a combination of both, respectively. These are used in constructing node names and numbers

The production rules (P) define how the nonterminal symbols are constructed from both nonterminal and terminal symbols. They form the core structure of the grammar, dictating how users can combine different elements to describe their graphs.

- The ⟨program⟩ rule establishes that a graph description must consist of a sequence of commands followed by draw.
- ⟨set of commands⟩ allows for an optional graph type followed by a list of connections, defining the graph's structure.
- ⟨connection list⟩ facilitates the definition of node connections and supports interspersed comments for clarity and documentation purposes.
- ⟨connection⟩, ⟨node⟩, and related rules specify the syntax for creating and naming graph elements and their interconnections.

Parsing example:

```
A-8-B

B-->C

B<-->D*



draw
```

Figure 2. Graph express code example

This code represents a graph definition in GraphExpress. The parsing according to our grammar rules and explain the role of each component is next:

1. A-8-B: this line defines a connection between two nodes, A and B, with a weight of 8. In terms of our grammar:
   - A and B are recognized as <node> elements.
   - -8- between A and B specifies the weight of the connection using the <number> nonterminal.
   - The entire structure A-8-B follows the <connection> rule, representing a weighted undirected edge between nodes A and B.
2. B-->C: this line defines a directed connection from node B to node C. According to our grammar:
   - B and C conform to the <node> definition.
   - --> between B and C indicates a directed edge starting from B and pointing towards C.
3. B<-->D*: this line represents a bidirectional connection between nodes B and D:
   - Again, B and D are identified as <node> elements.
   - The < and > around the - symbol indicate a bidirectional connection.
   - The * at the end of D indicates that D is a final node.
4. draw: this terminal symbol marks the end of the graph definition.

Starting with the <program> nonterminal, the parser recognizes a sequence forming <set of commands> followed by the terminal draw. The <set of commands> for this example does not explicitly define a graph type (e.g., Binary or Bipartite), so the default assumption is a general graph structure.

The <connection list> within <set of commands> comprises three <connection> entries, separated by newlines. Each connection utilizes different aspects of the <connection> grammar, demonstrating the versatility of our grammar in defining various types of node relationships.

The parsing process follows the formal rules set by the grammar, ensuring that the input code correctly represents a graph according to the language's syntax and semantics. The result of this parsing process would be an internal representation of the graph described by the input, ready for further processing, such as visualization or analysis.

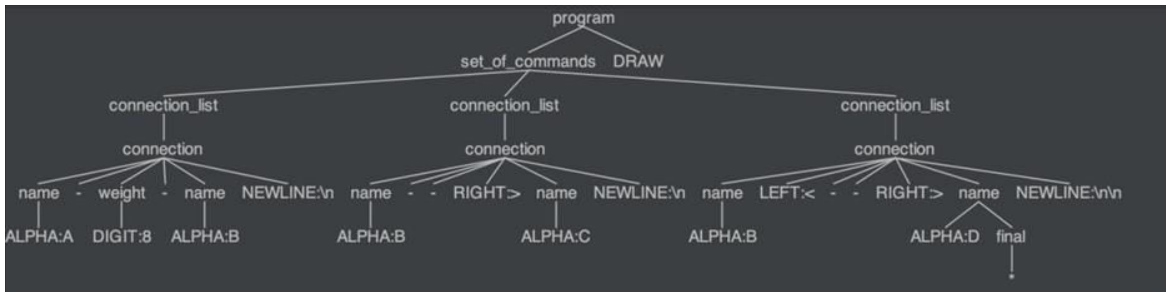

Figure 3. Parsing tree example

Conclusions

GraphExpress, as a DSL tailored for graphs, embodies an advancement in the field of graph analysis. Its development is a response to the evident need for specialized tools capable of simplifying graph management and analysis tasks. By integrating intuitive syntax, automated graph management, advanced analysis tools, and dynamic visualization, GraphExpress has significantly reduced the barriers to engaging with complex graph data. Its introduction promises to enhance productivity, foster deeper insights, and facilitate innovation in various domains where graph analysis is essential.

References:

[1] "Graph Theory Defined and Applications." [Online]. Available https://builtin.com/machine-learning/graph-theory
[2] "Graph Theory | Problems & Applications." [Online]. Available https://www.britannica.com/topic/graph-theory
[3] R. Diestel, *Graph Theory*. Springer, 2017
[4] M. Fowler, *Domain-Specific Languages*. Addison-Wesley Professional, 2010