DEVELOPING A DOMAIN-SPECIFIC LANGUAGE FOR GEOMETRY PROBLEMS

Andreea BURDUI, Tatiana MIHAILOVA, Ana-Maria TIMCIUC, Eduard SMELOV*, Renat GROSU

Department of Software Engineering an Automation, group FAF-222, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chisinau, Moldova

*Corresponding author: Eduard Smelov, eduard.smelov@isa.utm.md

Tutor/coordinator: **Cristofor FIȘTIC**, assist. univ.

Abstract. *The development of a domain-specific language (DSL) designed for geometric problems aims to streamline the process of formulating and solving mathematical and computational geometry challenges. This paper presents the design, implementation, and applications of DSL, which offers intuitive syntax and functionality to address a wide range of geometric scenarios. The DSL is specifically crafted to accommodate various use cases prevalent in fields such as computer graphics, computational geometry, robotics, and architectural design. The language enables users to express geometric concepts, operations, and algorithms, facilitating rapid prototyping, analysis, and visualization of geometric data. The lexer and parser components handle lexical considerations, providing error handling and efficient parsing of geometric expressions. Furthermore, the language incorporates a rich set of primitives, functions, and operators made for common geometric tasks, including point, line, polygon, and transformation manipulation. This paper details the language's grammar, lexical considerations, lexer, and parser components, offering insights into its design principles and implementation specifics.*

Keywords: *domain-specific, geometry, language, programing, tool, visualization.*

Introduction

A Domain Specific Language (DSL) is a type of programming language that is highly abstract and tailored to address problems within a particular domain effectively. It incorporates the unique concepts and regulations pertinent to that specific field [1].

This DSL is specifically designed to improve geometric problem visualization. This product provides a user-friendly platform for creating and manipulating geometric shapes, making it a unique tool for educational technology and engineering design. Some of the shapes that the tool covers are point, line, segment, triangle, square, rectangle, parallelogram, trapezoid, rhombus, circle, ellipse, etc.

This article will cover the development process of the DSL.

Grammar

Programming language grammar is a set of rules governing how code is structured, akin to grammar in natural languages. It ensures correct arrangement and sequencing of symbols, keywords, and elements. Following grammar rules enables clear communication between developers and computers, reducing syntax errors.

Lexical considerations

In this Geometry DSL, lexical considerations enhance clarity and maintain syntax integrity. Case sensitivity distinguishes entities like 'Shape' and 'shape'. Reserved keywords like 'Circle' are vital and cannot be used as identifiers. Comments improve code readability, supporting both single-line (//) and multi-line (/* */) annotations.

Whitespace is ignored for parsing, ensuring code layout does not affect execution. Identifiers must start with alphabetic characters and adhere to standard programming conventions.

Numerical literals represent sizes and coordinates, following decimal representation for consistency.

Special characters like '=' and '+' have defined roles, aiding geometric calculations. Adhering to these lexical guidelines maintains order and coherence, crucial for educational use and professional modeling tasks.

Terminal Symbols

Terminal Symbols are indivisible elements in the final string produced by formal grammar, forming the concrete content understood by the language. They cannot be altered by grammar rules and constitute the result of recursive rule application. The Start Symbol, a prime non-terminal symbol, marks the beginning of language parsing, initiating syntax tree construction or derivation sequence.

S = {<source code>}

In this geometry DSL, Terminal Elements include:

V_T= {
'=', '+', '-', '*', '/', '->', 'Point', 'Line', 'Segment', 'Triangle', 'Height',
'EquilateralTriangle', 'IsoscelesTriangle', 'Square', 'Rectangle',
Parallelogram', 'Circle', 'Ellipse', 'Rhombus', 'bisector', 'Angle', 'Vertex',
'{', '}','(', ')', 'for', 'while', 'if', 'else', 'true', 'false',
'++', '--', '<', '<=', '>', '>=', '==', '!='
}

Non-Terminal Symbols

Non-terminal symbols are the syntactic variables of your language, representing sequences of tokens and other non-terminals used to define the structure and rules of the grammar. They serve as building blocks for the language's structure. Non-terminal symbols help to organize the grammar into a hierarchy of rules, allowing for the modular design of language constructs. They play a crucial role in facilitating the expansion of the language. As new rules and structures can be added by defining additional non-terminal symbols without altering the existing grammar framework.

Our non-terminal symbols:

V_N = {
<program>, <statement>, <commentStatement>,
<functionCallStatement>,<functionCall>, <functionDeclaration>,
<loopStatement>, <forLoop>, <whileLoop>, <ifElseStatement>,
<figureDeclaration>, <variableDeclaration>, <expression>, <type>,
<point>,<comment>,<areaCall>, <perimeterCall>, <diagonalCall>, <areaTriangleCall>,
<areaCircleCall>, <areaSquareCall>, <areaRectangleCall>, <perimeterTriangleCall>,
<perimeterCircleCall>, <perimeterSquareCall>, <perimeterRectangleCall>,
<pointDeclaration>, <lineDeclaration>, <segmentDeclaration>, <triangleDeclaration>,
<squareDeclaration>, <rectangleDeclaration>, <parallelogramDeclaration>,
<circleDeclaration>, <ellipseDeclaration>, <rhombusDeclaration>, <aliasVertex>,
<triangleProperty>, <bisectorDeclaration>,<angleDeclaration>, <heightDeclaration>,
<forInit>, <forCondition>, <forUpdate>, <argumentList>
}

Production Rules

Production Rules form the grammar's backbone, outlining how strings within the language are built from non-terminal symbols as shown in Table. 1. The goal was to create comprehensible Grammar for all users of our geometry DSL.

Table 1

Meta-notation

Element	Description
<test>	Indicates that test is a non-terminal element.
test	Indicates that test is a terminal element.
x*	Zero or more occurrences of x
x+	Indicates one or more occurrences of x
(x)	Groups elements together to treat them as a single unit in expressions or rules, influencing the scope of operators like *, +, and ?.
→	Used to define production rules in ANTLR, showing that a non-terminal can be derived into other non-terminal or terminal symbols.
;	Ends a declaration or statement
|	Separates alternatives
?	Indicates zero or one occurrences of the preceding element, making it optional.
//	Comment section

The following key instructional rules form the basis of our language:

P = {
<program> → <statement>+
<statement> → <figureDeclaration> ';' | <variableDeclaration> ';' | <expression> ';' | <commentStatement>
<figureDeclaration> → <geometryType> <identifier> '(' <argumentList>? ')'
<geometryType> → 'Point' | 'Line' | 'Segment' | 'Triangle' | 'Height' | 'EquilateralTriangle' | 'IsoscelesTriangle' | 'Square' | 'Rectangle' | 'Parallelogram' | 'Circle' | 'Ellipse' | 'Rhombus'
<identifier> → (<letter> | '_') (<letter> | <digit> | '_')*
<letter> → 'a' | 'b' | 'c' | ... | 'z' | 'A' | 'B' | 'C' | ... | 'Z'
<digit> → '0' | <non-zero digit>
<non-zero digit> → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<variableDeclaration> → (<type>?) <identifier> ('=' <expression>)?
<type> → 'num' | 'bool' | 'string' | 'int'
<expression> → <expression> (<operator> <expression>) | <identifier> | <number> | <string> | <boolean> | '(' <expression> ')'
<operator> → '+' | '-' | '*' | '/' | '%' | '<' | '<=' | '>' | '>=' | '==' | '!=' | '++' | '--'
<number> → <numericValue>
<numericValue> → <digits> ('.' <digits>)?
<digits> → <digit> | <digits> <digit>
<string> → "" <string characters>* ""
<string characters> → <characters>*
<characters> → <letter> | <digit> | <special character>
<special character> → any printable character except ""
<boolean> → 'true' | 'false'
<argumentList> → <expression> (',' <expression>)*
<functionCallStatement> → <functionCall> ';'
<functionCall> → <identifier> '->' <functionDeclaration>
<functionDeclaration> → <areaCall> | <perimeterCall> | <diagonalCall> | <areaTriangleCall> | <areaCircleCall> | <areaSquareCall> | <areaRectangleCall> | <perimeterTriangleCall> | <perimeterCircleCall> | <perimeterSquareCall> | <perimeterRectangleCall>
<loopStatement> → <forLoop> | <whileLoop>
<forLoop> → 'for' '(' <forInit> ';' <forCondition> ';' <forUpdate> ')' '{' <program> '}'
<whileLoop> → 'while' '(' <expression> ')' '{' <program> '}'

<ifElseStatement> → 'if' '(' <expression> ')' '{' <program> '}' ('else' '{' <program> '}')?
<commentStatement> → '//' <comment> | '/*' <comment> '*/'
<comment> → <characters>*
}

Lexer

The process of translating high-level programming languages into executable machine code is facilitated by components such as lexers and parsers. Lexers, often referred to as lexical analyzers, parse source code into tokens, which serve as the foundational units for subsequent analysis.

Techniques for Efficient Lexical Analysis

Almost every lexer technique involves two primary stages: a scanner and an evaluator. First of all, the scanner, is often based on a finite-state machine (FSM). It has encoded information regarding the possible sequences of characters that can be found within any of the tokens it handles, known as lexems. What is a lexeme? The lexeme is simply a sequence of characters recognized as a particular type. To create a token, the lexical analyzer requires a subsequent stage, called the evaluator, which examines the characters of the lexeme to assign a value. It is the combination of the lexeme's type and this value that forms a token, which is then suitable for submission to a parser [2].

Parser

Parser Functionality

Parsing involves crucial functionalities. First, it scrutinizes code syntax provided by the lexer, ensuring adherence to programming language grammatical rules. Second, it constructs a parse tree or an abstract syntax tree (AST) capturing program hierarchical structure. Lastly, it handles error reporting and recovery, conveying syntax issues to developers and attempting error recovery to parse remaining code, enabling multi-error detection in a single pass.

Advanced Parsing Techniques

Various advanced techniques and tools address parsing challenges. Parser generators like Yacc, Bison, and ANTLR automatically generate parser code from formal grammar specifications. Predictive parsing, used by LL parsers, involves looking at upcoming tokens to make parsing decisions efficiently. Generalized parsing approaches, like Generalized LR (GLR) parsing, handle all context-free grammar, producing multiple parse trees in case of ambiguities [3].

Graphical Representation of Parse Trees

Parse trees visually represent code interpretation and structure based on language underlying grammar. For example, the respecting parse trees represent a point declaration (Fig. 1) and a triangle declaration (Fig. 2) within the DSL. These parse trees display the hierarchical nature of language syntax and parsing various statements and declarations within the DSL framework [4]. This method of visualization is crucial for understanding the flow and the logical structuring of code, which can help in debugging and optimizing the code. Moreover, it helps developers in grasping the abstract concepts of a DSL more effectively by providing a clear and concrete representation of how each component relates to others within the program's architecture.

The code ***Point B(100, 500);*** initializes a point, B, with coordinates (100, 500). This line of code is used to define the position of point B in a two-dimensional space. Below, a parse tree is provided to visually depict the structured analysis of this declaration (Fig. 1)

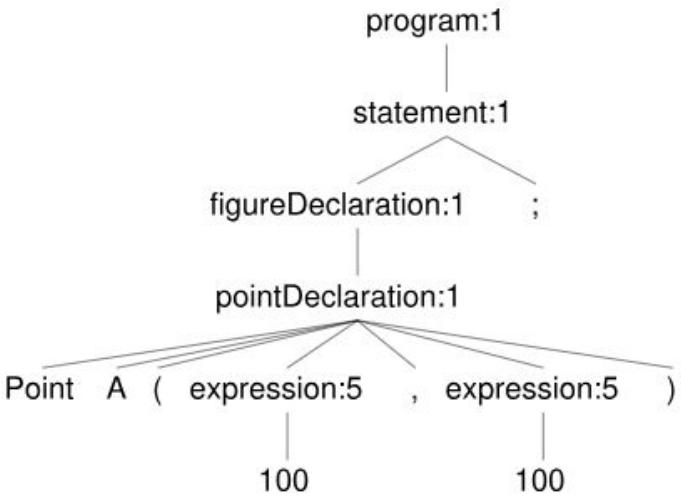

Figure 1. Point Declaration Parse Tree

The code ***Triangle t1(S : 200, D : 300, E : 400)*** initializes a triangle with specified parameters, where **S**, **D**, and **E** define the properties of the triangle such as side lengths or angles, depending on the context. This code is designed to generate triangles with these attributes. Below, a parse tree is provided to demonstrate the structured analysis of this declaration (Fig. 2)

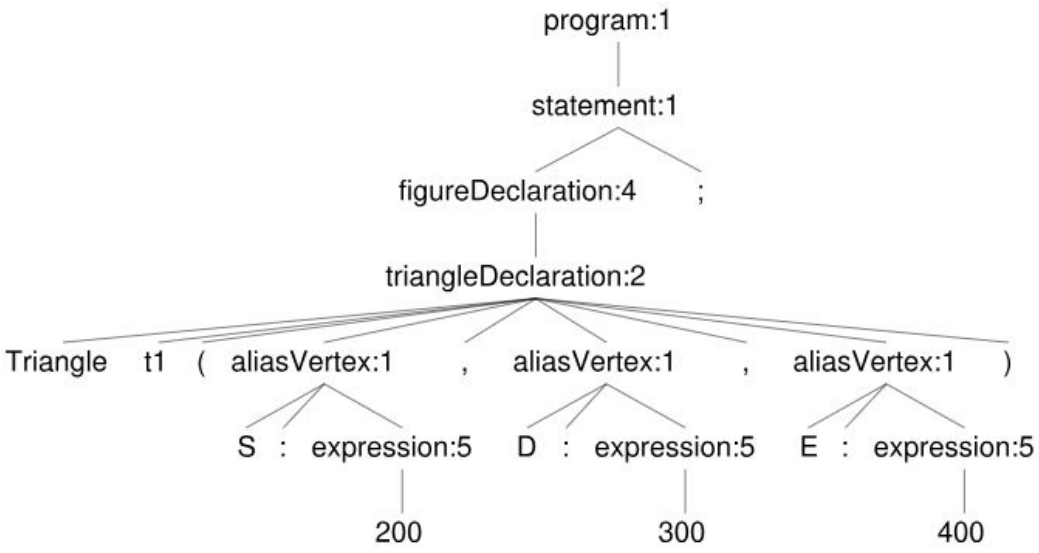

Figure 2. Triangle Declaration Parse Tree

The code ***for (int i = 0; i < 10; k++) { Point A(i, i+1); }*** generates a loop that iterates as long as i is less than 10. During each iteration, a new Point object named A is created with coordinates that incrementally increase by 1. Below is a parse tree (Fig. 3) illustrates the breakdown and hierarchical organization of this loop and the instantiation of Point objects inside it.

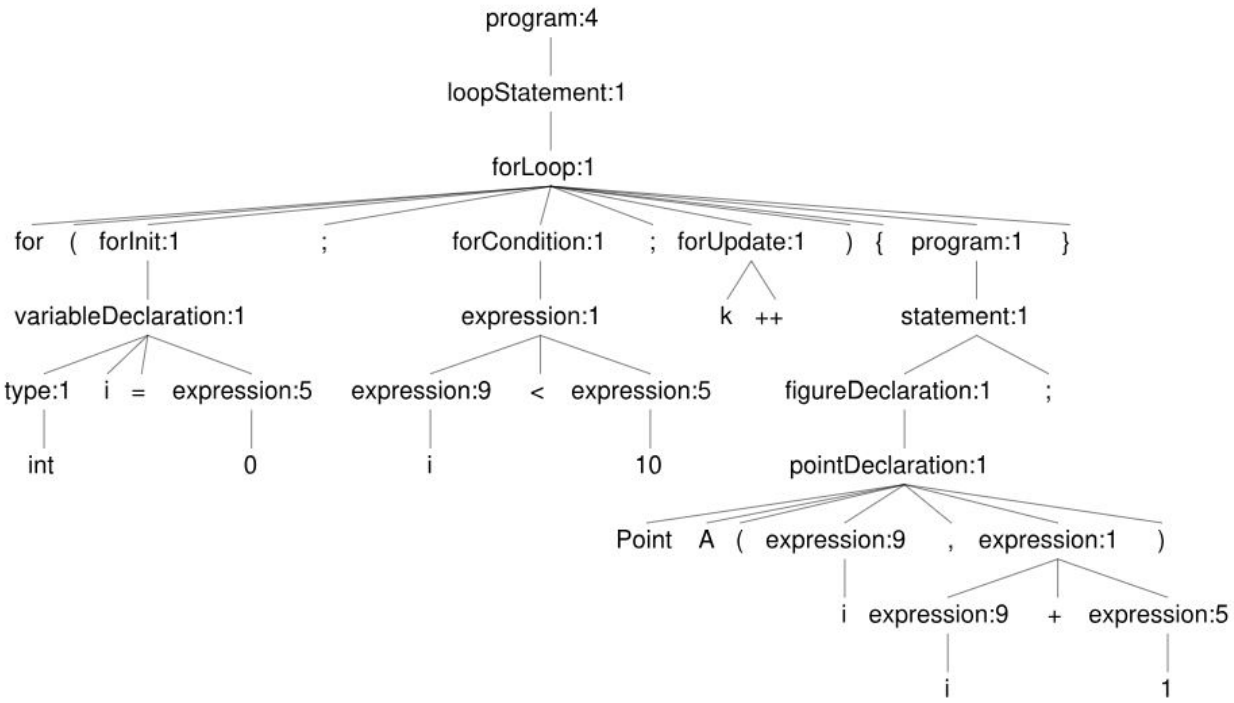

Figure 3. For-loop Declaration Parse Tree

The code ***if (k % 2 == 0) { Square sq(k); } else { Rectangle rect(k, k+1);}*** creates geometric shapes based on the value of k. If k is even, a square with side length k is created. If k is odd, then rectangle with width k and height k+1 is instantiated. So, this logic demonstrates conditional branching in programming. Below, a parse tree illustrates the hierarchical parsing of this conditional statement (Fig. 4)

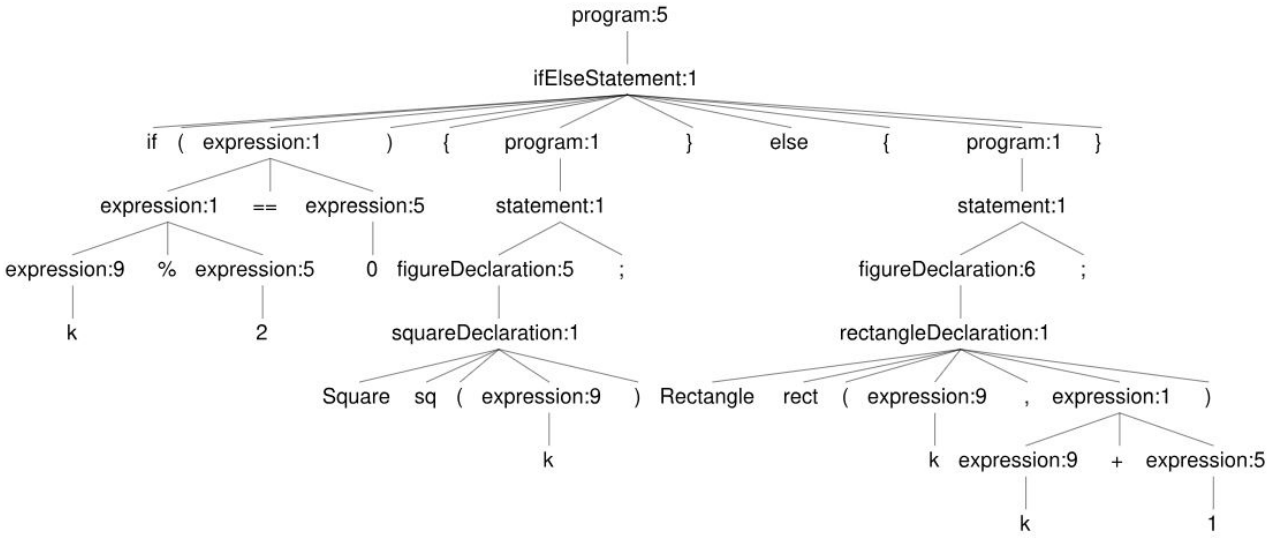

Figure 4. If-else statements Parse Tree

Conclusions

In this paper, we introduced the DSL made for geometric problems. We discussed its intuitive syntax, robust functionality, and diverse applications across fields like computer graphics, computational geometry, robotics, and architectural design. Throughout, we highlighted key features such as its flexible grammar, careful lexical handling, and comprehensive set of geometric tools. By delving into its design and implementation, including lexer and parser components, we emphasized the language's potential to streamline geometric computations. The language offers a

powerful solution for rapid prototyping, analysis, and visualization of geometric data. We believe its adoption will significantly enhance productivity and foster innovation in computational geometry, paving the way for future advancements.

Bibliography

[1] JetBrains s.r.o., *Domain Specific Languages.* [online] [accessed 26.03.2024] Available: https://www.jetbrains.com/mps/concepts/domain-specific-languages/#dsl
[2] GeeksforGeeks, *Introduction of Lexical analysis*. [online] [accessed 26.03.2024] Available: https://www.geeksforgeeks.org/introduction-of-lexical-analysis/
[3] Terence Parr, *Parser Rules.* [online] [accessed 26.03.2024] Available: https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md
[4] Cojuhari Irina, Duca Ludmila, Fiodorov Ion, *Formal Languages and Finite Automata.* [online] [accessed 26.03.2024] Available: https://else.fcim.utm.md/pluginfile.php/110458/mod_resource/content/0/LFPC_Guide.pdf