THE DEVELOPMENT OF A DOMAIN SPECIFIC LANGUAGE FOR MATRIX OPERATIONS

Vlada MAGAL*, Iurie CIUȘ, Ana COROLEȚCHI, Xenia-Qin Li WU, Adrian GHERMAN

Department of Software Engineering and Automation, Group FAF-203, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova

*Corresponding author: Vlada MAGAL, vlada.magal@isa.utm.md

Abstract. *Working with matrices can be difficult, as any mistake leads to failing to get the correct answer. It even becomes more complicated when we think of matrices of bigger sizes. All of the rules and formulas could be converted into something more user-friendly, a language in which more format inputs could be accepted with less rigid syntax, and which would perform lots of matrix operations given by quite simple commands.*

Keywords: *Domain-Specific Language (DSL), linear algebra, parse tree, grammar, matrices.*

Introduction

Matrix analysis is an important aspect of linear algebra. The majority of abstract linear algebra's characteristics and operations may be represented in terms of matrices In graph theory, incidence matrices, and adjacency matrices, matrices are crucial. There are many computational problems linked with numerical analysis that are reduced to a matrix computation to be solved. That involves often computing with matrices of massive dimensions.

Linear algebra, more than any other undergraduate mathematics course, offers greater potential usefulness for people studying in a variety of scientific and business sectors [1]. It is no understatement to proclaim the applications of linear algebra have revolutionized the world. They range from computer graphics to modeling current flow through electrical networks to machine learning [2].

Even though computing power is more accessible than ever before, linear algebra is still considered a subject only within the reach of brilliant theoretical mathematicians, and university courses do not particularly encourage students to discover mathematical applications besides theoretical considerations. A solution for this problem is a domain-specific language (DSL) for matrix operations.

The DSL will be a useful, free, and open-source tool for beginners to experiment and learn more about linear algebra and its applications without worrying about mindless calculations, and for experts in the field to automate the mind-numbing part of crunching numbers without having to deal with strict syntax. High school students, university freshmen, and all beginners in linear algebra would benefit from this DSL. Furthermore, it could also be used by teachers and mentors to demonstrate different operations to students or to check if their answers are correct. The domain of linear algebra would also benefit from this DSL, firstly because even experts could use a tool to simplify their work, secondly because it will attract more people to the domain.

Language overview

Working with matrices is, of course, most convenient when dealing with arrays. Arrays will be formed either of integers or floats. Other data that might serve as input or output data will be booleans, necessary for control structures. ANTLR is a powerful parser generator that can read, process, execute, or translate structured text or binary files. ANTLR generates a parser from syntax that constructs and walks parse trees.

Implementing a DSL includes multiple steps and vigorous research and planning. Grammar is made up of a set of production rules, each of which has a term and a description of how it is broken down. It does not tell anything about its semantics, that is, what an expression means. [3]. The first step, then, of creating a DSL is creating these production rules that will populate the syntax tree.

A formal grammar is in the form: G = {V_N, V_T, S, P}. Another step in creating the DSL would be defining the rest of the rules that "validate" a program besides the ones imposed by the grammar. Also, tokens need to be defined: reserved keywords based on the grammar and implement the lexer. The "nested expressions" take the form of an Abstract Syntax Tree. The parser validates the syntax tree, or grammar, and performs the target-language code generation.

Grammar design

In Tab. 1 the meta-notation used in the Extended Backus-Naur Form is described, often used to describe grammar.

Table 1

EBNF Meta-Notation

<x>	means x is non-terminal
x or 'x'	means x is a terminal
[x]	means x is optional (0 or 1 occurrences of x)
x*	means 0 or more occurrences of x
x+	means 1 or more occurrences of x
"|"	separates alternatives
"{" and "}"	are used for grouping alternatives

The grammar of the DSL is G = {V_N, V_T, S, P} where:

V_T = {<program>, <statements>, <statement>, <nosemicolon_statement>, <semicolon_statement>, <ctrlflow_statement>, <block>, <comment>, <return_statement>, <expression>, <assignment>, <for_statement>, <if_statement>, <while_statement>, <declaration>, <function_dec>, <parameter>, <variable_dec>, <variable_init>, <type>, <scalar_type>, <multid_type>, <function_call>, <prefix_expression>, <infix_expression>, <postfix_expression>, <bracket_expression>, <paranthesis_expression>, <identifier>, <number>, <integer>, <double>, <character>, <digit>, <nonzero_digit>, <operators>, <infix_op>, <postfix_op>, <prefix_op>, <assignment_op>},

V_T = {';', //, [,], {, }, ',', +, -, ++, --, +=, -=, !, ==, %, *, /, vector, matrix, int, longint, bool, double, break, for, if, else, while, void, returns, '_', a, b, c, … z, A, B, … Z, 0, 1, … 9},
S = <program>,
P = {

STATEMENTS:

<program> ::= <statements>*
<statements> ::= <statement>*
<statement> ::= <nosemicolon_statement> | {<semicolon_statement> ';' }
<nosemicolon_statement> ::= <ctrlflow_statement> | <block> | <comment>
<semicolon_statement> ::= <declaration> | <return_statement> | <expression> | <assignment> | **break**
<return_statement> ::= **return** <expression>
<ctrlflow_statement> ::= <for_statement> | <if_statement> | <while_statement>
<for_statement> ::= **for (** <variable_dec> **;** <expression> **;** <expression> **)** <statement>
<if_statement> ::= **if (**<expression>**)** <statement> [<else_statement>]
<else_statement> ::= **else** {<if_statement> | <statement>}
<while_statement> ::= **while (**<expression>**)** <statement>
<comment> ::= '//' <character>*
<block> ::= **{** <statements>* **}**
<return_type> ::= <type> | **void**
<assignment> ::= <identifier> <assignment_op> <expression>

DECLARATIONS:

<declaration> ::= <function_dec> | <variable_dec>
<function_dec> ::= **function** <identifier> **(**<parameter>***) returns** <return_type> <block>
<parameter> ::= <type> <identifier> [‘,’]
<variable_dec> ::= <type> <identifier> [<variable_init>]
<variable_init> ::= ‘=’ <expression>
<type> ::= <scalar_type> | <multidim_type>
<scalar_type> ::= **int** | **longint** | **bool** | **double**
<multidim_type> ::= <scalar_type> {**matrix** | **vector**} <bracket_expression>*

EXPRESSIONS:

<expression> ::= <identifier> | <number> | <prefix_expression> | <infix_expression> |
<postfix_expression> | <bracket_expression> | <paranthesis_expression> | <function_call>
<function_call> ::= <identifier> **(** [<expression> ‘,’] **)**
<prefix_expression> ::= <prefix_op> <expression>
<postfix_expression> ::= <expression> <postfix_op>
<infix_expression> ::= <expression> <infix_op> <expression>
<paranthesis_expression> ::= **(**<expression>**)**

IDENTIFIERS, NUMBERS, OPERATORS:

<bracket_expression> ::= [<expression>] ‘[’ <expression> ‘]’
<identifier> ::= <character> {<character> | <digit>}*
<number> ::= <integer> | <double>
<integer> ::= <nonzero_digit><digit>
<double> ::= <integer> ‘.’ <digit>*
<character> ::= **a** | **b** | **c** … **z** | **A** | **B** … **Z** | **_**
<digit> ::= **0** | **1** | … | **9**
<nonzero_digit> ::= **1** | … | **9**
<operators> ::= <infix_op> | <prefix_op> | <postfix_op>
<infix_op> ::= **+** | **-** | **&&** | || | **%** | **==** | **/** | ***
<prefix_op> ::= ++ | -- | !
<postfix_op> ::= ++ | --
<assignment_op> ::= **+=** | **-=** | **=**
}

DSL Program Example

Following is a simple example program for calculating the determinant in the DSL and the parsing tree. In Fig. 1 is represented the parse tree for the example program.

```
//hi
int matrix mx1 = (
                    1, 2;
                    2, 3;
                  );
double d;
d = get mx1 determinant;
```

As can be seen in the example program, unique elements of syntax will be introduced by the DSL, such as the “get” command, which will allow the user to call many operations, such as calculating determinants, eigenvalues, etc.

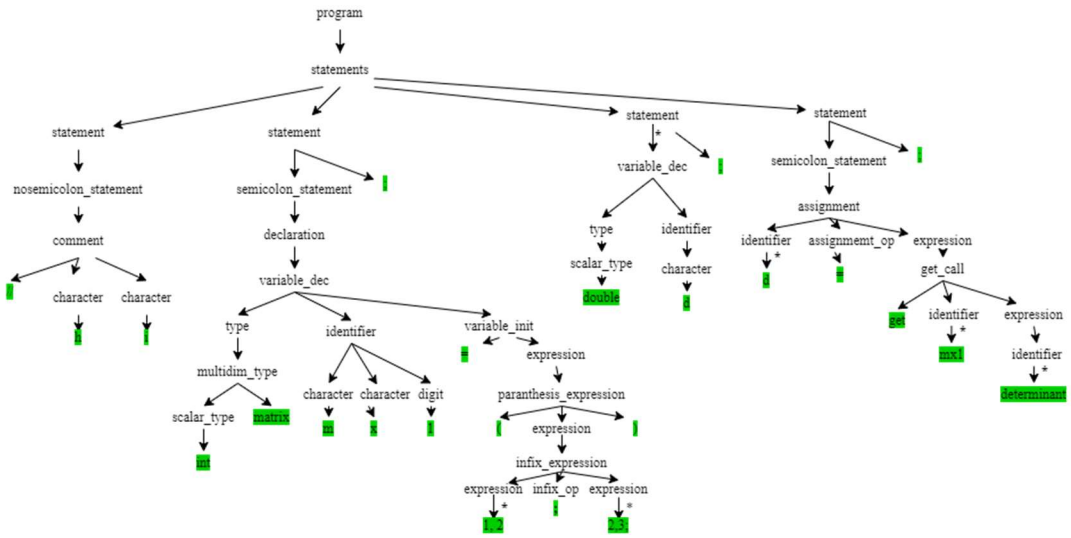

Figure 1. Parsing tree for program example

Conclusion

Domain-specific languages are languages that are designed specifically for a certain application domain. They have many potential advantages in terms of software engineering, ranging from increased productivity to the application of formal methods. This paper introduces a new DSL concept, which is centered around matrix operations. After implementing the language it is essential to consider in the future how intuitive it might be for the end-user, along with making it more specific by adding new variables depending on what works best for the domain that was chosen.

References

1. LAY, D., LAY S., and MCDONALD J. *Linear Algebra and Its Applications*. Washington: Pearson Education, Inc., 2016.
2. BROWNLEE, J. *10 Examples of Linear Algebra in Machine Learning* [online]. 2018.[Accessed: 2022-02-07]. Available: https://machinelearningmastery.com/examples-of-linear-algebra-in-machine-learning/
3. FOWLER, M. *Domain-Specific Language*s. Boston: Addison-Wesley Professional, 2010.