DOMAIN-SPECIFIC LANGUAGE FOR GRAPHICAL REPRESENTATION OF MATHEMATICAL EXPRESSIONS

Dumitru PUȘCAȘ[1], Antonela MALÎI[1*], Bogdan PICIRIGA[1], Nicolae URSU[1], Alexandru DANILESCU[1]

[1]Technical University of Moldova, Faculty of Computers, Informatics and Microelectronics, Department of Software Engineering and Automatics, group FAF-192, Chisinau, Moldova

*Autorul corespondent: Antonela Malîi, malii.antonela@isa.utm.md

Abstract. *In this article is described a Domain Specific Language for graphical representation of mathematical expressions. It is named Maf(s). The grammar of this domain specific language is simple, so that it cannot confuse the user with many different functions and tricks. It is focused on modelling graphically represented mathematical models together with a simulation of the step-by-step resolution of some of these expressions or equations.*

Key words: *DSL, grammar, functions, graphically, Maf(s), equations, functions.*

Introduction

A domain-specific language (DSL) is a computer language designed to be used in a particular field to solve a specific class of problems. It can be used in different contexts and by different kinds of users. Some DSLs are intended to be used by programmers, and therefore are more technical, while others are intended to be used by someone who is not a programmer and therefore they use less geeky concepts and syntax. The main idea of a DSL is to offer means which would allow the specialist of a particular domain to compute the solution using idioms and terms that he operates with. Some well-known examples of DSL are: HTML, CSS, SQL, MATLAB, XML, UML etc. [1].

The main advantages of a DSL in comparison with general-purpose programming languages are:

1. It can be used by people that don't have programming experience, as it is human-readable language and everything can be understood intuitively.
2. A DSL requires much less computational resources and memory.
3. All the elements of a DSL are the basic and most important ones, so the user won't have to dive into a big variety of different sets of commands and structures as in the case of a general-purpose language [2].

However, the easier is the language to be understood by a user the harder it is for the programmer to develop it, because the DSL developer must have an advanced level of knowledge in several fields: the field his DSL is based on, computer languages, compilers, language processors etc.

The DSL for Graphical Representation of Mathematical Expressions

Ma*f*(s) is a DSL with the main purpose to offer the users an easy way to represent different sorts of: equations, inequations, integrals etc.The DSL is flexible and requires a little computational resources. As the language is based on Mathematics, the main problem that it was decided to focus on is writing formulas and analyzing them, the people that have done at least once know what it is like. It seems like an easy task, but when it comes to integrals or differential equations it can get pretty tedious to write all the formulas and keep the track of correctness and rigorosity of the written equations. That's where Ma*f*(s) DSL comes into play. This language will

provide an easy and efficient way to formulate the equations and make an in-depth analysis. With the uninterrupted development of the Ma*f(*s) DSL, it is possible to implement many functionalities such as showing the solution step by step, which will be handy in understanding the results for the domain-experts and extremely useful for the students that want to get a better grasp of what's happening. Focusing more on the computational model, it should be said that our modelling language design consists of two parts: the first part provides domain concepts adapted for mathematical expressions modelling as well as the concrete syntax of the language, the second one aims at establishing a set of semantic operations which will execute our specified features of the DSL.

Grammar Rules

Grammar of a programming language is a technical way of describing a set of formal rules that overviews how the programming language is constructed and presents the valid tokens and lexemes. The code of any programming language can be implemented without any errors and troubles according to a specified valid grammar that should respect a set of predefined rules of definition of the grammar in general [3]. The grammar for the domain specific language consists of a 4-tuple $G = \{T, N, P, S\}$ where:

- "*T*"-set of terminals.
- "*N*"-set of non-terminal.
- "*S*"- start symbol.
- "*P*" – the set of production rules for generating valid sentences of the language.

Tabel 1

Meta-notation

<foo>	means foo is non-terminal
foo	(in **bold** font) means that **foo** is a terminal
x*	means zero or more occurrence of x
$x+$	a comma-separated list of one or more x's
|	separates alternatives

N = {<source code>, <program>, <block>, <var_decl>, <type> , <var> , <digit>, <digit*>, <number>, <statement>, <function>, <equation>, <inequation> , <expression>, <system>, <method>, <math_func>, <pow>, <sqrt>, <rel_operators>, <condition>, <findValue>}

T = {0.9, aA.zZ, + - < > <= >= = * / () [] {} # ^ ,pow, rad, solve, represent, if, end}

S = {start}

Rules:

P= {

<source code> → **start** <program> **end**

<program>→<block>*

<block>→ **#** <var_decl>^+ <procedure>^+ <statement>^+ **#**

<var_decl>→<type> <var>

<type> →**int | float**

<var>→ <char><digit>*

<char>→**a|b|…z|A|B|.Z**

<digit>→**0|1|.|9**

<digit*>→**1|2|3.|9**

<number>→<digit*><digit>*

<statement>→<function>|
<equation>|
<inequation>|
<expression>|
<system>|
<method>
<function>→ f(<var>^+)=<expression>
<equation>→ <expression> = <number>
<inequation> → <expression> <rel_operators> <number>
<expression>→<number>*<math_func>*<operators>*|
(<number>*<math_func>*<operators>*)
(<number>*<math_func>*<operators>*)<number>*<math_func>*<operators>*|
{(<number>*<math_func>*<operators>*)<number>*<math_func>*<operators>*}|
{(<number>*<math_func>*<operators>*)<number>*<math_func>*<operators>*}<number>*<math_func>*<operators>*
<system>→ '{' <equation>*
<math_func>→<pow>|
<sqrt>|
<var>
<operators>→ **+** | **-** | **/** | *****
<pow>→ **pow** (<var>,<number>)
<sqrt>→ **rad**(<var>, <number>)
<rel_operators> → < | > | <= | >=
<method>→if (<equation> | <inequation>) <condition>
[else <condition>]
<condition>→ <expression> | <equation> | <inequation>
<procedure> → **solve** | **represen**t | <findValue>
<findValue> → **findValue** (<var>= <number>)*
}

The code syntax is going to be checked by the parser and this will be done with the help of ANTLR tool. The script will run successfully if all the specifications from the grammar will be respected. Otherwise, it will display a message that something went wrong [4].

Ma*f*(s) have three basic data types: string, float and integer. The DSL gives the three main possibilities of statement procedures usage: "represent", "findValue" and "solve".

To explain how the tokenization will be realised, it will be represented the process according to a few different examples:

a)

```
start
#
int a
represent
2*pow(a,2)+a+7=0
#
end
```

b)

```
start
#
int x
findValue(x=3)
f(x)={[(x+7)+5]/10}
#
end
```

Figure 1 a) Input example nr.1 b) Input example nr.2

For the code from the Figure 1 a) and Figure 1 b) it can be elaborated the semantic implementation based on the rules defined earlier:

1. <source code>→ **start** <program> **end**→ **start** (<program>→ <block> → # <var_decl <procedure <statement **#) end**→**start** (# <var_decl> →((<type>→ **int**)

(<var>→(<char><digit>*)→**a**)) <procedure> → **represent** ,<statement>→(<equation> →(<expression> = <number>) →((<number>* ((<math_func>*)→pow(<var>,<number>)) <operators>*) = <number>)→ **2*pow(a,2)+a+7=0)** #) **end**

2. <source code>→ **start** <program> **end**→ **start** (<program>→ <block> → # <var_decl>^+ <procedure>^+ <statement>^+ #) **end** →**start** (# <var_decl> →((<type>→ **int**) (<var>→(<char><digit>*)→**x**)) (<procedure> → findValue→((<var>= <number>)→x=3)), <statement>→(<function> →(f(<var>^+)=<expression>) → (f(<var>^+)={[(<number>* <math_func>* <operators>*)<number>* <math_func>* <operators>*]<number>* <math_func>* <operators>*}))→ **f(x)={[(x+7)+5]/10}**) # **end**

Conclusion

In this paper it was analyzed and described what is a DSL and what are the main advantages of it. Additionally, it is presented the Ma*f*(s) language, which was developed to let people describe and visualize mathematical expressions. The grammar of language is intuitive and pleasurable to use, so it's convenient for a large group of people. To create Ma*f*(s) it will be developed a parser. When Ma*f*(s) code runs, the lexer is being created, then the token object shows-up, afterwards the parse comes into play and finally the tree is being parsed. Actually, all these actions depend on the type of input, for different inputs the order can be different.

References

1. FEDERICO TOMASSETTI, *The complete guide to external DSL*, 2019.
2. THORSTEN BERGER, *Domain-Specific Languages concepts, examples*,.2017.
3. PETER SEWELL, *Semantics of Programming Languages*, England: Cambridge University. 2008-2009.
4. *Lexical analysis and Grammar analysis*.[online].[access 25.02.2021] Available:teachcomputerscience.com/lexical-analysis/