DOMAIN SPECIFIC LANGUAGE FOR GRAPHIC REPRESENTATION

Laurentiu CIOCAN, Alexandru CASAP, Valeria SVET, Ion CEREMPEI

Technical University of Moldova

***Abstract**: This paper surveys the available literature on the topic of domain-specific languages as used for the construction and maintenance of software systems. Although many articles have been written on the development of a particular DSLs, there is very limited literature on graphic DSL development methodologies.*

***Key words:** Domain Specific Language, Graphic, Grammar, Finite Automata, ANTLR4.*

Introduction:

The computer graphics industry had a long development way. From the set of all changes that this industry was affected by, one of most significant was the creation and implementation of graphic domain specific languages namely the shading one. The reason why the development of such type of language became needed consists in offering a method of obtaining the same set of functionalities as it was possible to get by using assembly language, but in an intuitive and relatively not complex way.

When it comes to code writing using modern graphic libraries, there are several issues that should be considered. The first one is related to the difficulties met by the people that have the desire to become graphic programmers, without prior experience in programming. When the first issue is solved by the easy to learn means of working with computer graphics, the second issue appears. It deals with the programmer's obstacle to optimize his code written in a simple language.

As such, there is a need for a graphic domain specific language with an easy to understand and use syntax; a language that will make possible to access a large area of functionalities, similar to the already existing languages.

1. Development of Graphic DSL

In the second chapter are described the steps how to design and implement grammar for graphical domain specific language, and tools which were used. The DSL design includes several stages. First of all, definition of the programming language grammar $L(G) = (S, P, V_N, V_T)$ [1]:

- S - is a start symbol;
- P – is a finite set of production of rules;
- V_N – is a finite set of non-terminal symbol;
- V_T - is a finite set of terminal symbols.

The rule to start a language grammar is engage the parser first. Any rule in the grammar can act as a start rule for the following parser. Start rules don't necessarily consume all of the input. They consume only as much input as needed to match an alternative of the rule. For example, consider the following rule that matches one or few tokens, depending on the input:

<program> → <start><programName>**LCBRACE<**statements>**RCBRACE**<endProgram>

Where "<program>" is the start symbol. Non-Terminal symbols are: V_N = {program, programName, statements, action, argument, size, shape, color, position, id, alphaNum, start, endProgram}, and Terminal symbols are V_T = {LPAREN, RPAREN, PANDC, DRAW, MOVE, SCALE, ROTATE, SIZE, COLOR, AT, DIGIT, CIRCLE, SQUARE, BLACK, BLUE, BROWN, GREEN, RED, ORANGE, PUPLE, YELLOW, WHITE, COMMA, ALPHA, BEGIN, END, LCBRACE, RCBRACE, WS).

Second step is to define the reference grammar.

Table 1: Meta-notation

<foo>	means foo is a nonterminal.
foo	(in **bold** font) means that **foo** is a terminal i.e., a token or a part of token.
x^*	means zero or more occurrences of x.
\`(\` and \`)\`	braces in quotes are terminals.
|	separates alternatives.

<program> → <start> <programName> **`{`** <statements> **`}`** <endProgram>
<programName> → <id>
<statements> → (<action> **`(`** <argument> **`)`** **`;`**)*
<action> → **draw** | **move** | **scale** | **rotate**
<argument> → <shape> <id> **`size`** <size> **`color`** <color> **`at`** <position>
→ <id> **`at`** <position>
→<id> **`size`** <size>
→ <id>
<size> → <digit>
<shape> → **circle** | **square**
<color> → **black** | **blue** | **brown** | **green** | **red** | **orange**| **purple** | **yellow** | **white**
<position> → <digit> **`,`**<digit>
<id> → <alpha> <alphaNum>*
<alpha> → **a** | **b** | … | **z** | **A** | **B** | … | **Z**
<alphaNum> → *<alpha> <digit>*
<digit> → **0** | **1** | **2** | **3** | … |**9**
<start> → **begin**
<endProgram> → **end**

Third step in DSL development is definition of regular expression [2]. Similar to use of finite automata for recognition of strings patterns, regular expressions are used to generate patterns of strings. A regular expression is an algebraic formula which value is a pattern consisting of a set of strings, called the language of the expression.

Operands in a regular expression can be:

- characters from the alphabet over which the regular expression is defined;
- variables whose values are any pattern defined by a regular expression;
- epsilon which denotes the empty string containing no characters;
- null which denotes the empty set of strings.

For an easier reading of regular expression - derivation of the non-terminal variable "argument" was written separately:

RE = BEGIN(ALPHA(ALPHA + DIGIT)*) LCBRACE ((DRAW + MOVE + SCALE + ROTATE) LPAREN (argument) RPAREN PANDC)* RCBRACE END

argument → ((CIRCLE + SQUARE)(ALPHA(ALPHA + DIGIT)*) SIZE (DIGIT) COLOR (BLACK + BLUE + BROWN + GREEN + RED + ORANGE + PURPLE + YELLOW + WHITE)AT (DIGIT COMMA DIGIT))
| ((ALPHA(ALPHA + DIGIT)*) AT (DIGIT COMMA DIGIT))
| ((ALPHA(ALPHA + DIGIT)*) SIZE (DIGIT))
| (ALPHA(ALPHA + DIGIT)*)

The next step is to build the finite automata in JFLAP [2, 3]. A finite automaton (FA) is a simple idealized machine used to recognize patterns within input taken from some character set (or alphabet) C. The task of an FA is to *accept* or *reject* an input depending on whether the pattern defined by the FA occurs in the input.

A finite automaton consists of:

- a finite set S of N states;
- a special start state;
- a set of final (or accepting) states;
- a set of transitions T from one state to another, labeled with chars in C.

The FA shall be execute using an input sequence as follows:

- begin in the start state;
- if the next input char matches the label on a transition from the current state to a new state, go to that new state;
- continue making transitions on each input char:
  - if no move is possible, then stop;
  - if in accepting state, then accept.

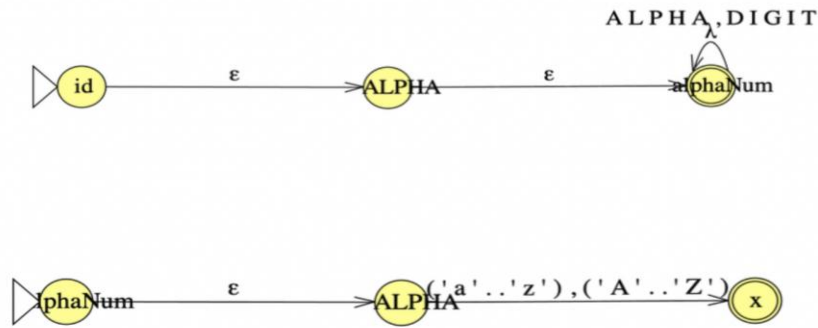

Figure.1 Finite Automata examples in DSL.

The final step is to develop a DSL using ANTLR4 [3, 5]. The ANTLR Tool generally is used to generates a parser and lexer (native is written in Java, but optionally can be modified in other languages) that you can compile. ANTLR generates the following files:

- ExprParser.java: the recursive-descent parser from the grammar.
- Expr.tokens: the list of token-name, token-type assignments.
- Expr_.g: the automatically generated lexer grammar that ANTLR derived from the combined grammar. The generated file begins with a header: lexer grammar Expr;.
- ExprLexer.java: the recursive-descent lexer generated from Expr_.g.

In the Figure 2 is shown the parse tree and in the Figure 3 is shown the hierarchy of the following version of a program:

```
BEGIN
Program
{
  DRAW(CIRCLE C1 SIZE 1 COLOR ORANGE AT 1,1);
}
END
```

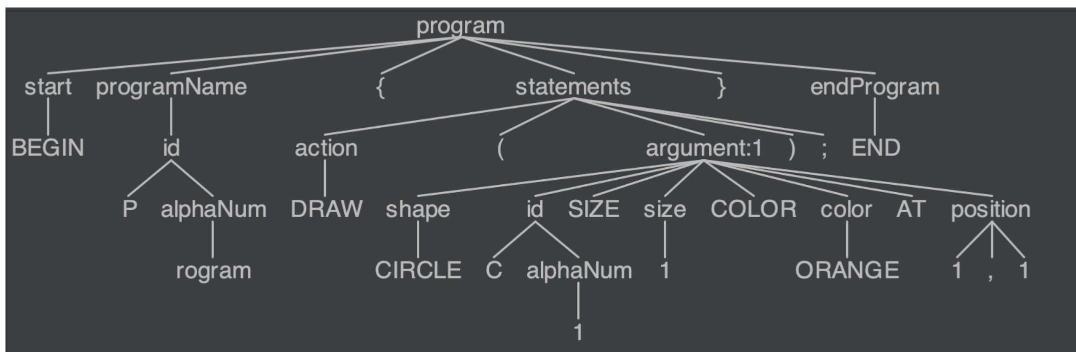

Figure 2 Program parse tree.

Figure 3 Program hierarchy.

Conclusion:

The syntax for this DSL allows for an easy, yet reliable way to manipulate graphics without requiring any programming experience. Only having statements (as it is the only element a program in our DSL can consist of) makes the language as basic as possible. The shapes can be named, thus giving the user the ability to separate and interact with any object individually. The backend for this language is the WebGL API, making the content perfect for browsers, so there shall be no need for any special or sophisticated environment that would be hard for the user to set up. The 'problem' with the WebGL API is that it requires a ton of experience and knowledge to use. That's why, with this DSL, the aim is for the end-user of our product to be children that want to learn the basics of programming from the comfort of their favorite web browser, as drawing and manipulating graphics is one of the better ways to keep a child entertained. Given that the product will be as simple and straightforward to use as possible, there are some features that have to be considered, even if it would slightly enhance the DSL's complexity. A great addition to this DSL would be an update function so the scene could change dynamically, which would also come with a few new statements to support it. Although it's not planned yet, it's definitely in the developer's mind as a future improvement. Other ideas that are worth our consideration are if-statements, lighting and perspective manipulation. Again, should that be implemented, it would be as simple as it can possibly be, for the sake of the end-user.

References:

1. E. H. John and M. J. D. U. Rajeev, Introduction to Automata Theory, Languages, and Computation, 2001, p. 169.
2. M. P. Andrew, "Lectures Notes on Regular Languages and Finite Automata," 2010.
3. L. Peter, An Introduction to Formal Languages and Automata, vol. Fifth Edition, 2012.
4. T. Federico, "www.tomasseti.me," [Online]. Available: https://tomassetti.me/antlr-course/. [Accessed 26 February 2019].
5. T. Parr, The Definitive ANTLR Reference, Building Domain-Specific Languages, 2007.