DOMAIN SPECIFIC LANGUAGE FOR GRAPHIC DESIGN INDUSTRY

Alexandru ROȘCA[1], Dinara BUCILA[1*], Lina CIPCIU[1], Constantin TOLICI[1], Eric-Vadim CEBANU[1], Igor OVADENCO[1]

[1] *Technical University of Moldova, Faculty of Computers, Informatics and Microelectronics, Department of Software Engineering and Automation, Group FAF-193, Chisinau, Republic of Moldova*

[*]Corresponding author: Dinara Bucila, bucila.dinara@isa.utm.md

Abstract*. This article describes a Domain Specific Language (DSL) explicitly created for grapical industry area. The main purpose of the language is the automate abstract image generation, used in various artistic aims when creating a large scale project. The language grammar is constructed in a straightforward manner, in order to be understood by professionals from distinct correlative aesthetic fields. The main DSL focus consists of image producing according to the input commands introduced by the user. The language operates with a specific C++ graphic library, which proides various fascinating opportunities in terms of graphical manipulations. The grammar was defined manually, without using additional tools, for simplistic purposes. Additionally, the ANTLR tecnoogy was used for abstract syntax tree generation and code analysis.*

Keywords: *abstract image, texture, grammar, ANTLR, abstract syntax tree, pattern*

Introduction

There is no doubt that choosing strong optics sets the stage for the overall success of a design. Usually professionals in this field can spend a huge amount of hours looking for the exclusive illustrations for websites or projects. Thus, images are certainly a straightforward key to a successful final result. The proposed DSL solves the problem of difficult appropriate image searching, further to be used in various projects, including different areas, which are strongly interconnected with graphic design. To put it another way, the main purpose of the DSL is characterized by abstract image generation for first-class visual content processing. The general mechanism of the language is the following: the source code is being divided into tokens and then processed by parser in order to find matches of the parser rules [1]. All rules and tokens are described in special grammar file, which was processed by ANTLR and resulted into generation of lexer, parser and listener files [2]. The flow of the language is always the same – it executes every command from top to bottom, one by one, taking one command at a time. In case of unrecognized syntax or semantic validation issue, the program will throw an appropriate error message.

Computational Model

The behavior expressed with a DSL must of course be aligned with the needs of the domain. In many cases, the behavior required for a domain can be derived from behavioral paradigms, with slight adaptation or enhancements, or simply by interacting with domain-specific structures or data. The imperative paradigm was used for DSL designing. Imperative programs consist of a sequence of statements, or instructions, that change the state of the program [3]. This state may be local to some kind of module (e.g., a procedure or an object), global (as in global variables) or external (when communicating with peripheral devices). Procedural and object-oriented programming are both imperative, using different means for structuring. Because of aliasing and side effects, imperative programs are expensive to analyze. Debugging imperative programs is straightforward and involves stepping through the instructions and watching the state change. The imperative paradigm is characterized by the finite state machine computational model. A Finite State Machine is a model of computation based on a hypothetical machine made of one or more

states. Only one single state of this machine can be active at the same time [4]. It means the machine has to transition from one state to another in order to perform different actions. The important points which describe the Finite State Machine and also are respected in the DSL implementation are the following ones: there is a fixed set of states that the machine can be in; the machine can only be in one state at a time; a sequence of inputs is sent to the machine; every state has a set of transitions and every transition is associated with an input and pointing to a state [5].

Reference Grammar

In Table 1 are represented meta-notations used for grammar specification.

Table 1

Meta-notations used in language grammar definition

Meta-notation	Significance
<x>	means x is a nonterminal
x	(in **bold** font) means that **x** is a terminal i.e., a token or a part of token
x*	means 0 or more occurrences
{ }	large braces are used for grouping
|	separates alternatives
[x]	means 0 or no occurrences of x, x is optional
V_T	set of terminals
V_N	set of nonterminals

Below are represented some of the grammar productions for the Domain Specific Language:

V_T = {{1.1000}*, square, circle, triangle, {a.z}*, Julia_set, Fibonacci_word, #808080, #ffffff, #000000 **,** /path/image.png, def, :, =, .,), (, ;}

V_N = {<source-code>, <create_canvas>, <name>, <width>, <height>, <digit>, <int>, <canvas_color>, <canvas_gradient>, <hex_color>, <type>, <color_list>, <create_pt>, <sides_number>,<sides_dimension>,<pt_color>,<create_line>,<broken_line>, <broke_number>, <waved_line>, <wave_number>, <create_point>, <radius>, <pt_distance>, <image_bg>, <path>, <create_fractal>, <pt_remove>, <save>, <blure>, <intensity>, <darken>, <lighten>, <grain>, <b_w>, <lens>, <mirror>, <string>, <pointillism>, <result>, <phyllotaxis>, <object_name>}

P = {

<source-code> —> **def** <object_name> **=** <function>**:;** | <object_name>**.**<function>**:;**

<function> —> <create_canvas> | <canvas_color>* | <canvas_gradient>* | <create_pt>*

<object_name> —> <string>

<create_canvas> —> {<width>, <height>}

<canvas_color>—> <hex_color>

<canvas_gradient> —> {<gtype>, <color_list>*}

<create_pt> —> {<ptype>, <pt_color>, <sides_dimension>}

<sides_number> —> <int>

<sides_dimension> —> <int>

<height> —> <int>

<width> —> <int>
<color_list> —> {<hex_color>*}
<gtype> —> <digit>
<int> —> **{(1.1000)}***
<string> —> **{(a.z)}***
<hex_color> —> **(#ffffff | #000000 | #808080)**
<digit> —> **{(1.1000)}***
<ptype> —> (**square | circle | triangle)**
<path> —> **/path/image.png**
<ftype> —> (**Julia_set | Fibonacci_word)**
<string> —> **{(a.z)}***}

Semantics and semantic rules

The main rule for the whole source code is expecting a list of statements separated by ‘;’. The statement can be either a variable definition or a function call. Variable definition command basically requires a ‘def’ keyword, variable name, assignment symbol and a call of a function, which returns an object. A function call has a more diversity in terms of syntax. Basically, it can be a variable name along with a ‘.’ and a function name with a double dot sign and with or without parameters. If a function requires arguments, they should be enclosed in braces and separated by comma. There are general functions which doesn’t belong to an object type, and work like constructors in programming languages. They are used as a part of variable definition statement since they return an object (such as pattern or canvas) which can be assigned to variable. Function argument might be one of the following statements: another function call, string literal, integer literal. However, each function has some specific requirements or bounds for its arguments. In this perspective, a certain function can receive only some specific string values.

Data Types

The datatypes in Graphical DSL can be partitioned into 2 categories: literals and objects. Literals contains only 2 types – strings and integers. Strings literals have some restrictions on their definition: they cannot contain single quote character and their length cannot exceed 255 symbols. The integer by itself has range from -2,147,483,648 to 2,147,483,647. Objects are more complex data structures which might hold a big amount of information about an entity in the program. In this DSL, object expose to the user a list of methods which can change the object state or provide some output related to this object. The inner implementation of the objects is hidden from the user, in order to simplify the DSL.

Variable scopes and rules

The DSL provide a single scope for the variables – the variable is available right after its definition (e.g., all lines which are below the definition line), and cannot be accessed before definition or used in another file. The rules for a variable definition are straightforward. First of all, the keyword ‘def’ must be used before variable name. Then, after variable name there must be an assignment symbol and the function call, which will return the object.

Example of code and parse tree in ANTLR

In the following piece of code two variables ‘can’ and ‘pt’ are defined and after that the method ‘add_pattern’ is used to add the figure on the canvas. Additionally, the source code is parsed by abstract syntax tree construction. The result is presented in Fig. 1 (Parse tree in ANTLR).

```
def can = create_canvas:(500, 200);
def pt = create_pt:(‘triangle');
can.add_pt:(pt);
```

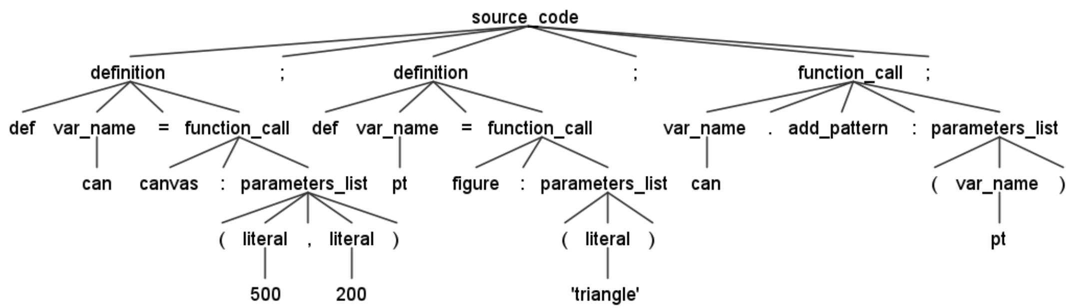

Figure 1. Parse tree in ANTLR

Conclusion

The DSL will differ from other existing languages due to the fact that it will contain various functions, ranging from specifying the object's type (e.g., a point, a line or a square) to the ability of adding various effects (e.g., blur or fade). On the future perspective it is established to implement the code interpreter and to create some additional functionality. Also one of the main goals would be to improve the universality of the DSL by usage scale enlarging.

References:

1. MARTIN FOWLER, REBECCA PARSONS. *Domain-Specific Languages.* Addison-Wesley Professional; 1st edition (September 23, 2010).
2. TERENCE PARR. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf; Second edition (February 5, 2013).
3. MARKUS VOELTER. *DSL Engineering Designing, Implementing and Using Domain-Specific Languages.* CreateSpace Independent Publishing Platform (January 23, 2013).
4. JOHN CARROLL, DARRELL LONG. *Theory of Finite Automata with an INTRODUCTION to FORMAL LANGUAGES.* Pretince Hall (February 1, 1989).
5. ROBERT B.Reese, JUSTIN DAVIS. *Finite State Machine-Datapath, Design, Optimization and Implementation.* Morgan and Claypool Publishers; Illustrated edition (February 27, 2008).