DOMAIN SPECIFIC LANGUAGE FOR LINDENMAYER SYSTEMS

Patricia CAPITAN*, Marius BADRAJAN, Victor FLORESCU, Mihai MUȘTUC

Department of Software Engineering and Automatics, FAF-201, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Moldova

*Corresponding author: Patricia Capitan, patricia.capitan@isa.utm.md

Abstract. *The article addresses the process of creation of a Domain-Specific Language (DSL) that will generate and design fractals, fractal plants, and architectural geometries of different colors, dimensions, using Lindenmayer systems, otherwise known as L-systems. L-systems can be used as a powerful design tool. Minimal inputs are able to create a spatially complex output. Since L-system is a topic of interest for people like botanists, biologists, architects, etc., these not necessarily possessing advanced programming skills, a DSL to graphically represent patterns created using an L-system would be quite of use for their works and researches.*

Keywords: *fractals, fractal plants, DSL, L-system, design tool*

Introduction

A domain-specific language (DSL) is a computer language that is specific to a particular software domain. There are many different DSLs, ranging from the languages widely used for common domain names to the languages used by some software [1].

L-systems were first used to simulate the development of basic multicellular organisms in terms of cell division, growth, and death. The purpose of modular modeling is to explain the overall development of a plant, and in particular the appearance of plant morphology, as the integration of individual unit development. Now they are used to generate geometric structures (fractal-like objects). L-systems can describe the structure of a fractal as an axiom, an alphabet of symbols and a set of productions using those symbols. Each symbol in the grammar is given a visual interpretation so an arbitrary string created by an L-system can be converted to a picture. The string after each iteration gives a fractal-like form when each symbol is given a visual representation [2].

There are very few tools targeted specifically towards representing L-system patterns, let alone the logic behind it and the implementation. A DSL would solve this issue and help people native with this domain during their working process of modeling the growth of biological systems. Moreover, for architects it would represent a powerful design tool. In regards to programming, the DSL might turn out handy and convenient for students who are interested in studying fractal-like objects.

The paper describes the process of generating a domain-specific language, whose objectives are the following:

- to offer better understanding of the performance and to document the requirements and behavior of L-systems;
- to develop a tool which will help people native with this domain in their works;
- to generate graphic images of objects created using L-systems, such as fractals, etc.;
- to solve the issue representing the lack of graphic tools for L-systems.

Language Design

The generated domain-specific language is based on a sequential mathematical model of computation, specifically a finite automaton that will operate on predefined grammar rules for different types of graphic outputs. In the frontend of programming language compilers, finite automata are frequently utilized [3]. From a sequence of characters, the lexical analyzer generates a sequence of language tokens (such as reserved words, literals, and identifiers), which the parser utilizes to generate a syntax tree. The lexical analyzer and parser handle the regular and context-free aspects of the computer language's grammar.

The DSL holds 3 data types: integer, character, and string, and one composite data type - functions, which represent the main way of generating a graphical output. Strings literals have some restrictions on their definition: they cannot contain symbols other than those accepted by the function type, meaning for certain graphical designs there are certain rules for characters and symbols to be followed. Functions save more data about L-system items. To keep the DSL simple, the inner implementation of the functions is hidden from the user.

The criterion for any source code is that a set of statements separated by ';' is expected. A variable definition or a function call can be used in the statement. A variable name, assignment symbol, and a call to a function that returns an object are all required for the variable definition command. In terms of syntax, a function call offers more options. Arguments should be enclosed in braces and separated by a comma if a function requires them. For its arguments, each function has some specific constraints or limitations.

The DSL works as follows: the code is separated into tokens, which are then processed by a parser to locate parser rule matches. Rules and tokens are processed by ANTLR to generate lexer, parser, and listener files. The program executes each command one by one, one at a time, from top to bottom. The program will provide an appropriate error notice if there is an unknown syntax or semantic validation issue.

Grammar

G = (V_N, V_T, S, P):

V_N = { **<program>, <ls freestyle>, <ls tree>, <ls dragon>, <define>,**
<type>, <alphalower>, <identifier>, <value>,
<parameters>, <axiom>, <applies>, <angle>, <length>, <rules>,
<start>, <short>, <long>, <directions>,
<num>,
<rule>, <first>, <second>,
<for>, <A>, <B>, <C>, <X>, <Y>,
<if>, <expression>, <statement> }

V_T = { [a-z], [A-Z], [0-9], [+ | -], ls freestyle(), ls tree(), ls dragon(), int, char, string, _, ”, ;, for(;;), if(){}else{}, [_ | ++ | -- | = | < | > | <= | >= | == | !=] }

S = **<program>**

P = { **<program>** → **<ls freestyle>**; | **<ls tree>**; | **<ls dragon>**; | **<define>**; | **<for>**; | **<if>**;

<define> → **<type> <identifier>** = **<value> | <type> <identifier>**
<type> → int | char | string
<identifier> → <alphalower> | <alphalower><num> | <alphalower>_<alphalower> <alphalower> → a|...|z | **<alphalower>** a|...|z
<value> → <num> | <alphalower>

<ls freestyle> → ls freestyle(**<parameters>**)
<parameters> → ”**<axiom>**”, **<applies>**, **<angle>**, **<length>**, {**<rules>**}
<axiom> → <start>
<start> → <short> | <long>
<short> → [a-z] | [A-Z] | [0-9]
<long> → **<short> | <short><long> | <long><directions> | <directions><long> |**
<long><directions><long>
<directions> → [+ | −] | [+ | −] **<directions>**
<applies> → [1-9] | [1-9]**<num>**
<num> → [0-9] | [0-9]**<num>**

<angle> → [0-9] | [1-9][0-9] | [1-2][0-9][0-9] | 3[0-5][0-9] | 360
<length> → [0-9] | [1-9][0-9] | 100

<rules> → <rule> | <rule>,<rules>
<rule> → ”**<first>**”:”**<second>**”
<first> → <short>
<second> → <long> | [**<second>**] | **<long>**[**<second>**] | [**<second>**]**<long>** |
<long>[**<second>**]**<long>**

<ls tree> → ls tree(**<num>**)

<ls dragon> → ls dragon(**<num>**)

<for> → for**(<A>**;**<B>**;**<C>**){}
<A> → **<X>**=**<Y>**
<B> → **<X>**<**<Y>** | **<X>**>**<Y>** | **<X>**<=**<Y>** | **<X>**>=**<Y>**
<C> → **<X>**++ | **<X>**--
<X> → [A-Z] | [a-z] | [A-Z]**<X>** | [a-z]**<X>** | _**<X>** | **<X><num>** | **<X><num><X>**
<Y> → **<num>**

<if> → if(**<expression>**){} | if(**<expression>**){};else{}
<expression> → **<identifier>**[< | > | == | <= | >= | !=]**<statement>**
<statement> → **<statement>** | **<num>** | ”**<X>**” | ’**<short>**’
}

Grammar Description

- **define variables:**
  <type> - data type of variable
  <identifier> - name of variable

- **ls freestyle**: function to design plants using L-systems, according to user's personal preferences
  <axiom> - The starting string of the L-System.
  <applies> - The number of times the rules are applied to the string.
  <angle> - The angle to apply for the turning commands, in degrees.
  <length> - The length of each f (move) command, in pixels.
  <rules> - A dictionary of character and substitution strings.

  Supported commands:
  f : move forward
  + : turn angle right
  - : turn angle left
  [: start branch
  ] : end branch

- **ls tree/ls dragon:** functions that generate 2 examples of L-systems designs
  <num> - The number of recursions.

- **for statement:**
  <A> - Initialization
  <B> - Condition
  <C> - Increment/Decrement

- **if statement:**
  <expression> - Condition

Syntax Example and Parsing Tree

Syntax: ls freestyle(axiom, applies, angle, length, rules);

Input: ls freestyle(”X”, 6, 25, 10,{”X”: ”f−[[X]+X]+f[+fX]−X”, ”f”: ”ff”});

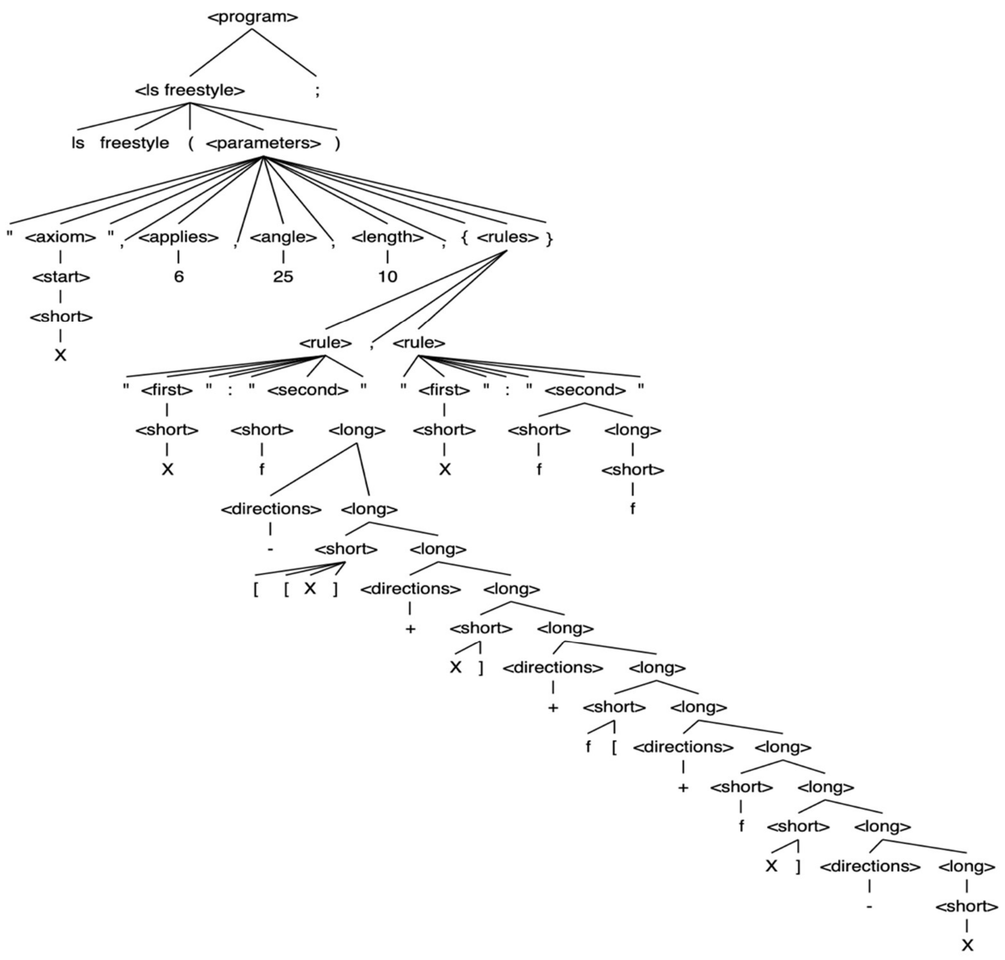

Figure 1. Parse Tree Example

Conclusion

A domain-specific language designed specifically towards representing graphical outputs created using L-systems would result in a powerful tool for people interested in this scientific sphere. Further, using the generated DSL would be a great way to learn the behavioral features of L-systems and how these work in different aspects of graphics. Students, along with others possessing little to no programming and coding experience, may use this tool in their personal works and/or for research purposes. The DSL would bring recognition to L-systems, their representation and relevance in diverse domains.

References

1. Domain-specific language [online]. [accessed on February 27, 2022]. Available at: https://en.wikipedia.org/wiki/Domain-specific_language
2. PRUSINKIEWICZ, P., HANAN, J., HAMMEL, M., MECH, R. *L-systems: from the Theory to Visual Models of Plants* [online]. [accessed on February 27, 2022]. Available at: http://algorithmicbotany.org/papers/sigcourse.2003/2-1-lsystems.pdf
3. Finite-state machine [online]. [accessed on February 27, 2022]. Available at: https://en.wikipedia.org/wiki/Finite-state_machine