DOMAIN SPECIFIC LANGUAGE FOR GENERATING FLOOR PLANS

Alexandru FURDUI[1], Dmitri TRUBCA[1], Iana SPIVAC[1*], Iulia VULPE[1], Valentin DOGARI[1]

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

*Corresponding author: Iana Spivac, e-mail: spivac.iana@isa.utm.md

Abstract: *This article describes a domain-specific language which is designed to transform textual instructions for constructing 2D building plans into a visual representation. It is tried to keep the grammar of this language as simple as possible, so that the average user doesn't get confused among the many different and complex functions, allowing at the same time to create floor plans in an easy way.*

Keywords: *Domain-specific language (DSL), Plan, Syntax, Grammar.*

Introduction

The purpose of the given DSL is to create a friendly and functional environment for people working in the field of construction planning. Although there are many programs dedicated to this, such as AutoCad, none of them offer optimal use of resources and to use them, the programs need to be installed on computers with quite powerful hardware. DSL will be compact and easy to use, even the person who has no programming experience, after a short training will be able to use it.

Because this DSL does not require many hardware resources, that is why it will be cross-platform unlike AutoCad [1] or 3DMax. To represent an apartment plan using this DSL, the program must contain instructions for creating each room of which it consists, as well as some other objects that it may contain.

To display any object, must be specified its position. The user will be able to choose between absolute and relative coordinates. Due to the ability to place objects by entering their relative location (for example, room Y to the right of wall1 of the room with the ID X), the process of drawing the plan will be greatly simplified and accelerated. Each object would have properties which have some predefined default values. For example, for a room it would be possible to set the wall thickness (border), color, floor type. Also, it would be possible to indicate the position for doors, windows or wall opening, so that the plan would be displayed accordingly. To represent the data visually, after parsing, the output will be transformed into a Semantic Model – an object representation of the Abstract Syntax Tree, and executed correspondingly.

The general basic features for the user are:

1. Selecting the number of the edges of the element;
2. Selecting the length,width and angle of the edge;
3. Rotating elements;
4. Selecting the color of the elements;
5. Setting the overlay property for elements;

Semantics and Semantics Rules

A program in the proposed language consists of variable declarations, expression evaluations and struct data types specifications (each one for a different part of the floor plan). In case of struct data types, only their properties must be specified, thus, their values shouldn't be assigned to a variable. Each different type of structure has their own list of mandatory properties, whose presence will be explicitly checked by the interpreter at runtime. Each property must be of the required type and within the given constraints.

Besides the constraints implied by the grammar, we have specified a list of rules that place additional constraints on the set of valid programs in this DSL:

1. No identifier is declared twice;
2. No identifier is used before it's declared;
3. The value of *id* property for struct types has to be unique;
4. For structures that accept the properties *size* and *angles*, the length of the lists specified by their values must be equally;
5. The value of the <id_parent> property, must be equal to the value of <id> property of a *Room* structure declared beforehand.

For consistency purposes, the walls (inside the room struct data type) should be defined in clockwise order.

Data types

Proposed DSL has primitive data types such as: int, float, color, measure. Non primitive data types are: List, Room, Door, Window, Wall and secondary struct data types such as: Table, Wall, Elevator, Bed.

Assignment

Assignment is only used for scalar values. Assignment statement sets and/or re-sets the value stored in the storage location(s) denoted by a variable name, in proposed DSL a variable will receive a value by using "=" , a property inside a structure receives a value by the using ":".

Lexical consideration

In our language keywords are case sensitive. The reserved words [2] are: **Room, Window, Wall, Door, Elevator, Stairs, Bed, Table, Chair, Wardrobe, id, id_parent, size, angles, border, position, wall, start_on_wall, end_on_wall, length, direction, start, end, width, height, distance_wall, layer, visibility, hidden, visible, int, float, color, measure, list, mm, cm, dm, m, km, in, ft, yd, mi.**

Comments are started by // and are terminated by the end of the line. Multiple line comments are started with **/*** and terminated with */

White space may appear between any lexical tokens. White space is defined as one or more spaces, tabs, page and line-breaking characters, and comments.

Reference Grammar

Table 1

Meta-notation

Token	Description
<foo>	means foo is a nonterminal.
foo	(inbold font) means that foo is a terminal; i.e., a token or a part of a token.
[x]	means zero or one occurrence of x, i.e., x is optional; note that brackets in quotes ′[′ ′]′are terminals.
x*	means zero or more occurrences of x.
x + **,**	a comma-separated list of one or more x's.
{}	large braces are used for grouping; note that braces in quotes ′{′ ′}′are terminals.
|	separates alternatives.

S = {<source_code>}

V_N={<statement>,<variable_name>,<variable_value>,<arithmetic_operation>,<variable_declaration>,<elementar_structure_type>,<room_structure_declaration>,<room_structure_declaration>,<window_structure_declaration>,<door_structure_declaration>,<wall_structure_declaration>,<elementar_structure_declaration>,<structure_declaration>,<direction_type>,<visibility_type>,<data_type>,<alpha>,<digit>,<number>,<alpha_num>,<float_literal,<color_hex>,<arithmetic_terms>,<measure_literal>}

V_T = {:, (,), ;, , ,/*,*/,//,{,},#,+,-,/,'*',=,Room, Window, Wall, Door, Elevator, Stairs, Bed, Table, Chair, Wardrobe, id, id_parent, size, angles, border, position, wall, start_on_wall, end_on_wall, length, direction, start, end, width, height, distance_wall, layer, visibility, hidden, visible, int, float, color,measure, list,0.9,a.z,A.Z,mm, cm, dm, m, km, in, ft, yd, mi}

P = { <source_code> → <statement>*

<statement>→<variable_declaration>;|<elementar_structure_declaration>;|<room_structure_declaration>;|<window_structure_declaration>;|<door_structure_declaration>;|<wall_structure_declaration>;|<arithmetic_operation>;|<commentary>

<elementar_structure_type> → **Elevator** | **Stairs** | **Bed** | **Table** | **Chair** | **Wardrobe**

<room_structure_declaration> → **Room(id:** <id>**, size:** {<measure_literal> +, | <variable_name>}**,angles:**{<float>+,|<variable_name>}**,border:**{'**{**'<measure_literal>**,**<color_hex>'**}**'|<variable_name>},{**position:**{'**{**'<measure_literal>**,**<measure_literal>'**}**'|<variable_name>}| **position:** '**{**'<measure_literal>**,**<measure_literal>**,**<measure_literal>**,**<measure_literal>'**}**' | <variable_name>}**)**}

<window_structure_declaration>→**Window(id:**{<id>|<variable_name>}**,id_parent:**{<id>|<variable_name>}**,wall:**{<number>|<variable_name>}**,**{**start_on_wall:**{<measure_literal>|<variable_name>}|**end_on_wall:**{<measure_literal>|<variable_name>}}**,length:**{<measure_literal>|<variable_name>}**)**

<door_structure_declaration>→**Door(id:**{<id>|<variable_name>}**,id_parent:**{<id>|<variable_name>}**,wall:**{<number>|<variable_name>}**,**{**start_on_wall:**{<measure_literal>|<variable_name>}|**end_on_wall:**{<measure_literal>|<variable_name>}}**,length:**{<measure_literal>|<variable_name>}**,direction:**{<direction_type>|<var iable_name>}**)**

<wall_structure_declaration>→**Wall(start:**{'**{**'<measure_literal>,<measure_literal>'**}**'|<variable_name>}**,end:**{'**{**'<measure_literal>**,**<measure_literal>'**}**'|<variable_name>}**,border:**{'**{**'<measure_literal>, <color_hex>'**}**' |<variable_name>})

<elementar_structure_declaration> → <elementar_structure_type>**(id:**{<id>|<variable_name>}**,**

layer:{<number>|<variable_name>}**, visibility:**{<visibility_type>|<variable_name>}**,** **width:**{<measure_literal>|<variable_name>}**,height:**{<measure_literal>| <variable_name>}**,**{**position:**{'**{**'<measure_literal>**,**<measure_literal>'**}**'|<variable_name>}|**id_room:**{<number>|<variable_name>}**,wall:**{<number>|<variable_name>}**,**{**start_on_wall:**{<measure_literal>|<variable_name>} | **end_on_wall:** {<measure_literal>|<variable_name>}}**, distance_wall:** {<measure_literal>|<variable_name>}})

<id> → <alpha><alpha_num>*

<commentary> → // <alpha_num>*

<variable_declaration> → <data_type> <variable_name> =<variable_value>

<arithmetic_operation> → **+** | **-** | ***** | **/**

<arithmetic_operation>→<variable_name>=<arithmetic_terms>[<arithmetic_operation> <arithmetic_terms>]

<arithmetic_terms> → <variable_name>|<number> +

<data_type> → **int** | **float** | **list** | **color** | **measure** <variable_name> → <alpha><alpha_num>*

<variable_value>→<number>+|<alpha_num>+|<color_hex>|<float_literal>
+|<measure_literal>
<alpha> → **a**|.**z**|**A**|.**Z**
<digit> → **0**|.|**9**
<number> → <digit> +
<float_literal> → <digit> + . <digit> +
<measure_literal> → {<float_literal> | <number>} <measure_unit>
<measure_unit> → mm | cm | dm | m | km | in | ft | yd | mi
<alpha_num> →<alpha>|<digit>
<list> → '**{**'{<list_item>{, <list_item>}*}*'**}**'
<list_item> → <color_hex> | <number> | <float_literal>
<color_hex> → #<alpha_num> +
<visibility_type> → **hidden** | **visible**
<direction_type> → **in** | **out**
}

Code example and syntax tree

Below is an example of how to declare a Room structure and its parse tree Fig. 1:

Room(id: 1a, size: {10m,10m,10m}, angles: {60.0,60.0,60.0}, border:{10cm,#000000},position: {10.0m, 10.0m});

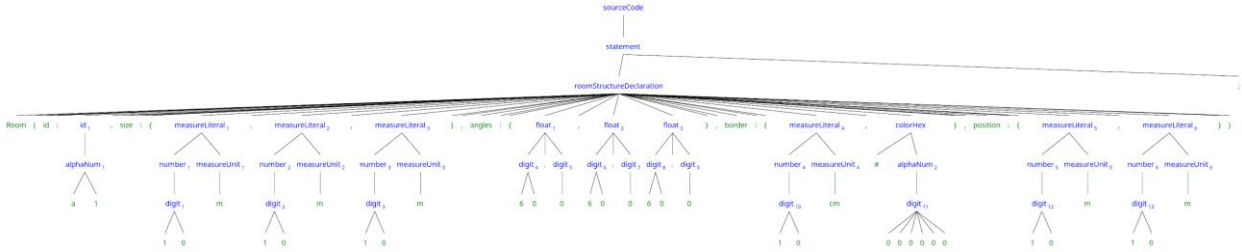

Figure 1. Parse tree of declaration of Room structure

Conclusions

The syntax of the proposed language, allows for an easy, yet reliable way to draw floor plans, without requiring any additional visual editor. All statements are intuitive, and can be understood instantly by a domain expert. It provides a rich functionality that allows users to have the best experience while designing the plan for their house. To test the grammar, we used ANTLR [3] parser generator tool. As development progressed, a new lexical analyzer and parser was implemented using a recursive descent approach and by writing a routine for each non-terminal in the grammar. The parser is written in C++, a language that allows low level manipulations and offers a fast processing speed. After the parse tree is created, the Semantic Model representing the parsed data is populated and the graphics are displayed using OpenGL.

Reference:

1. AutoCAD.[online].[accessed 03.02.2021] Disponible:https://www.autodesk.com/products /autocad/overview?term=1-YEAR&support=null
2. HARRISON, MICHAELl A. *Introduction to Formal Language Theory.* Boston:Addison-Wesley, 1978.
3.The ANTLR [online].[accessed 20.02.2021] Disponible: https://tomassetti.me/ antlr-mega-tutorial/