DOMAIN SPECIFIC LANGUAGE FOR CADASTER DOMAIN

Valeria BUTNARU, Andrei CERNEI, Cătălin COȘERU*, Pavel NEAGU, Vladimir RUSSU

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

*Autorul corespondent: Coșeru Cătălin, catalin.coseru@isa.utm.md

Abstract. *The aim of the article presented is to create the language specific to the field called TUMCAD (Technical University of Moldova in Cadaster). This article present a Domain Specific Language (DSL), that will be able to involve some basic tools for creating and editing data, as well as for viewing them. There are lots of applications like CadFunc, AutoCad, Qgis or Inventory. It forms a wide range of programs that an engineer must use to undertake some work. Thus, the idea is to create a series of instructions and functions, which will facilitate the representation, the passage of data, the 2D writing, the possible 3D writing of the points in space. Also, some functions that could achieve the change of different coordinate systems. The TUMCAD DSL will be based on help, fast data work and saving or self-managing things that now need to be done manually. The innovation TUMCAD offers is that each point has some variables that can be changed by the user, in order to add additional information that may be needed.*
In the article is presented the main idea of TUMCAD and some important details such as grammar, syntax and functionalities.

Keywords*: TUMCAD, cadastru, Domain Specific Language, cadaster, engineering.*

Introduction

The cadastral domain is a domain always present in society and deserves to be paid attention when it comes to technology and automation. Thus, was born the idea of creating a DSL that would bring value and streamline the work of people in the cadastral field. Following intensive research, the problems that require a quick, engineering approach have been identified. Therefore, have been created a series of functionalities that TUMCAD could incorporate.

One of the main functionalities is to transform from one coordination system to another, it need time and give problems (solution at this moment is Qgis in terrain cadaster and inventory for 2D of main structures, like homes etc); besides this, area calculation occupies an important place in this field. Therefore, area calculations of a specific terrain, or of a specific space obtained after linking some points (coordinates) is an important feature of DSL.

The TUMCAD is addressed to the cadastral officials, both the district ones and the ones from the villages, and communes. Is proposed a tool to help carry out the work of managing cadastral databases, updating them, and presenting data in a format not only professionally but also socially to show consumers data about plots of land and houses they own, their dimensions and plan.

Also, if for not moving away from the tertiary sector, this article is also for users such as small cadastral companies, who would be happy with a tool that is easy to handle and that has many facilities.

Students and the education sector would be the basis for profitable advertising. Students will want to work with a tool they are already familiar with, which would increase the popularity of this DSL.

Lexical Considerations

To use keywords such as *For, If, Bool* it was decided only use uppercase keywords. Because of this, all words are case-sensitive (to differentiate variables and input data from keywords).

Keywords will be: *Int, Double, Bool, Char, String, Foreach, Array, Map, If, While, Func, True, False, Else, Return, End, Continue, Plot, Coord, Area, CreateBasement* .

For some functions already written or added by the user, it will be possible to execute them by naming them in box II of the application. Any other execution will be done by typing in the main program in box I. To determine the beginning and end of the function is used the terminal symbols {}, also to determine the input data of the functions is used the terminal symbols ().

Comments will be written between / * and * / or using // before.

Free spaces can exist anywhere in the code, they have a syntactic role. In addition to the input parameters or string values (array of char) of a variable. Also from this rule it was deduced that any keyword, function and other thing must be delimited by a free space (or white space). As in c ++ all character values will be taken between "** ... *", and the char type will take the value between '*'. In our case * it can be any ASCII character. (for the use of char values such as ('), ("), (\), (/) etc., they must be preceded by \, also means a new line [1].

Int has values equal to long long int from c ++, which means that it can take values between -(2 ^ 63) and (2 ^ 63) -1. Double will be all floating point numeric data [2].

The main program will start with { and end with }, both terminal variables that will be automatically put in the editing application.

Reference Grammar

< > - nonterminal param;

bold – terminal param;

% … % - optional;

x* -zero or more occurrence of x ;

P={<program> → <block>
<field_declaration> → ε | <type> <name> |<type> <name> **‘[‘** <int_lit> **‘]’ %‘[‘** <int_lit> **‘]’%**
<var_declaration> → ε | <type> <name>
<method_declaration> → ε | **Func** [<type> | **void**] <name> (<param_in>) <block> % **Return** <expr>%
<param_in> → ε | <var_declaration> <param_in> $^+$ | <field_declaration> <param_in> $^+$
<block> → **‘{‘** <statement>* **‘}’**
<statement> → <loc> <assign_op> <expr>;
| <var_declaration> ;
| <method_declaration>;
| <field_declaration>;
| <method_call>;
| **If ‘(‘** <expr> **‘)’‘{‘** <statement>$^+$ **‘}’** % **else ‘{‘** <statement>$^+$ **‘}’** % ;
| **Foreach** <expr> **in** <expr> **‘{‘** <statement>$^+$ | **End** | **Continue ‘}’ ;**
| **While ‘(’**<expr>**’)’ ‘{‘** <statement>$^+$ | **End‘}’ ;**
| **Coord ‘(‘**<expr>$^+$ **‘)’ ‘{‘** <statement> **‘}’**
| **Plot ‘(‘** <expr>$^+$ **‘)’**
| **Area ‘(‘** <expr>$^+$ **‘)’**
| **CreateBasement ‘(‘** <expr>$^+$ **‘)’**
| <statement>
<assign_op> → = | += | -=
<method_call> → <name> (%<expr>$^+$%)
<loc> → <name> | <name> **‘[‘** <expr> **‘]’** |<name> **‘[‘** <expr> **‘]’ ‘[‘** <expr> **‘]’**
<expr> →<loc>
| <method_call>
| <literal>
| <expr> <bin_op> <expr>

| - <expr>
| ! <expr>
| (<expr>)
<callout_arg> → <expr> | <string_lit>
<bin_op> → <arith_op> | <comp_op> | <cond_op>
<arith_op> → + | - | * | / | %
<comp_op> → < | > | <= | >= | == | !=
<cond_op> → && | ||
<literal> → <int_lit> | <char_lit> | <double_lit> | <bool_lit> | <string_lit>
<name> → <alpha><alphanumb>*
<alphanumb> → <alpha_string> <digit>
<alpha> → ε | a | b | .. | z | ;
<alpha_string> → <alpha>$^{+}$<digit>*
<digit> → 0| 1| 2| 3| .. | 9
<int_lit> → <digit>*
<double_lit> → <int_lit>* **'.'** <int_lit>*
<bool_lit> → **True | False**
<char_lit> → ' <char> '
<char> → **a | b | …**
<string_literal> → " <char>* "
<type> → **Int | Double | Bool | Char | String | Void | Map** }

Data types

The data types used in TUMCAD are *Int, Bool, Double, Char, String, Void* and *Map*.

Control Structures

If blocks, and *For* and *While* cycles, are also statements, so they are assigned to this non-terminal variable. *If* is composed of the keyword *If* with the first letter in UpperCase, followed by an expression, and the next <block> (string of statements) to be executed. In the case of the false expression, the else block can be optionally written, which can also execute the statements [3].

The *For* cycle consists primarily of the segment of the keyword *For*, followed by the name of the metered variable, with the application of its assignment, to a certain expression. And the execution of the cycle until the value in the following expression after the comma is reached. The possible string of affirmations is executed, with access to the *End* and *Continue* terminal parameters. The role of *End* is to exit the cycle, and Continue is used to increment the metering operator and skip this round of the cycle.

While, the simplest cycle, begins with the keyword *While*, followed by a *Boolean* expression, the value of which involves the repeated execution of the string of statements inside the block. It is possible to execute the keyword *End*. Which also has the role of getting out of the loop.

Operators

The assignment operators are of 3 types, 1 the usual assignment, 2 the increment of the target variable by adding with the value of the variable or a <loc> from the right side of the assignment [1]. And of course the value of the variable on the left decreases, as well as a ratio equal to the value of the variable on the right.

Binary operators are of the same importance and execution as assignment ones. The only difference is that there are only two, representing the keywords *True* and *False* [1].

Arithmetic operators are +, -, *, /, % depending on the type of variables they may be optional [2]. For example, double variables cannot be divided by % (mode) to get the rest of the division.

Digit and <alpha> are non-terminal parameters that represent letters and decimal digits. Char_lit are char words that can be composed of any ASCII character that must be written between 'and'. The string is a string of such characters written between "..." [3].

Execution

For the execution of functions and procedures is used a simple non-terminal parameter <method_call> where it writen only the name (which must be equal to the name of a method stated above) and the string of input parameters (in our case of expressions <expr>) which is optional.

Expressions can be both variable names and array and array2D variable values. (these variables are stored in the non-terminal parameter <loc>)[1].

Example of code and Parse tree in ANTLR

```
{
Int value223
Double var22
Func void coord(  Int var2 ; ) {
value223 = var22    }
coord ( var2 = 10 )
Bool b
b = True
}
```

Figura 1. Example of code

In the piece of code represented above are declared 3 variables: value223 of int data type, var22 of double data type and a Boolean b with the value assigned “True”. There is also the initialization and the call of the function “coord”.

Next, Figure 2 shows the parsing of the previously presented piece of code.

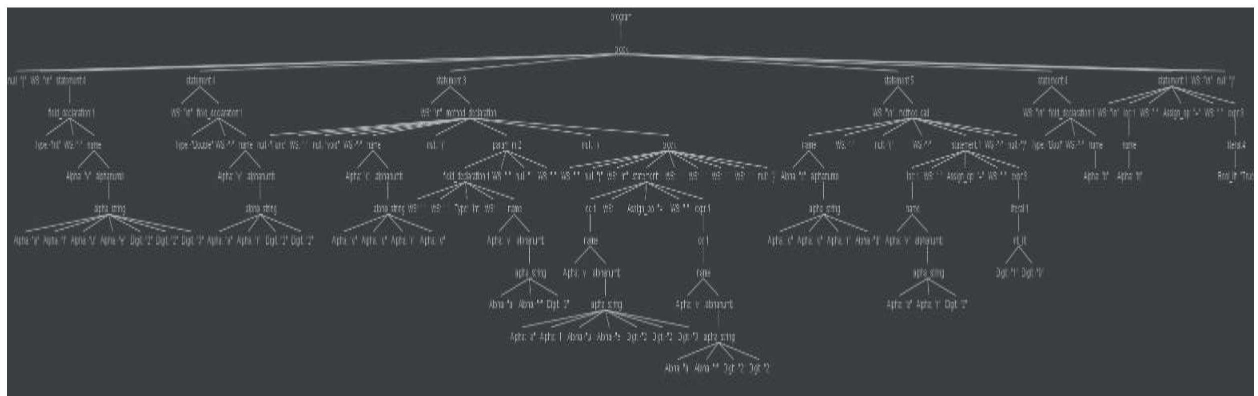

Figura 2. Parsed TREE

Conclusion

Domain specific languages are languages created to support a particular set of tasks, as they are performed in a specific domain. The purpose of this paper was to present a DSL that will facilitate the work of employees in the cadastral domain, who have to perform complex calculation and transformation tasks. It will automatize the prolems they encounter daily, saving time and making the process more interesting.

The proposed DSL is user friendly, intuitive and easy to understand even by those who don't have programming skills. All those combined together with a simple user interface will increase the popularity and the development of the DSL.

References

1. PETER SEWELL, *Semantics of Programming Languages*., 2009.
2. KALEIDOSCOPE *Introduction and the Lexer*. [online]. 01.03.2022, [acc. 02.03.2022].Available:https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html
3. ANTLR: Documentation. [online]. 04.02.2019, [acc. 02.03.2022].Available: https://github.com/antlr/antlr4/blob/master/doc/index.md