DOMAIN SPECIFIC LANGUAGE FOR CHILDREN'S MAZE GAME

Stephania MATVEI*, Daniela AFTENI, Ștefan BOICU, Răzvan FIȘER, Serghei COVTUN

Department of Software Engineering and Automatics, Group FAF-203, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova

*Corresponding author: Stephania Matvei, stephania.matvei@isa.utm.md

Abstract. *This article is about a Domain-Specific Language (DSL) developed upon a Problem-Based Learning project, which has the aim to facilitate the process of studying programming through interactive games.*

Keywords: *data structures, derivation tree, education, grammar, interactive learning*

Introduction

The DSL for Children's Maze Game is based on 3 main fields: programming, gaming and education. It represents a game-based language, which can provide user friendly environment for learning, especially for kids, that can start growing their critical skills through interactive games. One of the major reasons is that most school schedules use General Purpose Languages for teaching kids programming, but this is not ideal for several reasons: children have short attention spans, they don't like reading documentation and are not easily engaged in the classic exercises used to teach programming concepts.

The DSL improves upon this in the following ways: DSL is presented as a game which will pique the child's interest, it engages 2 kids at once and uses the very effective element of competitiveness, it has a clear goal which children can understand and work toward so as to win the game, also it encourages optimization of the code, as a sloppy written code will be easily beaten by their opponent.

Data Structures

Arrays are used quite often by games developers to structure and organize data, as well as by us. It is the best to use a data structure that would allow the game to store and update data in a grid, like a two-dimensional array [1]. With the help of an array, it can be done the following:

- Set the size of the grid [1];
- Set the value of data elements at individual locations [1];
- Retrieve the value of data at individual locations [1].

The array uses two numbers to indicate the position of a data element.

Control Structures

The DSL for Children's Maze Game will have two options for control structures:

1. Using arrow keys (up, down, left, right);
2. Using characters (W, A, S, D).

Both options are included because some people prefer to control player characters with the keyboard characters: W, A, S, and D, and others prefer to use arrow keys.

Input and Output

Since the DSL is having its purpose in the field of education, especially in the development of coding skills in young children, the input must be as simple as possible in order to be graspable by the still-in-development minds. The goal is to produce a DSL for an interactive, two-player maze game. The user should be able to type in functions such as "go_up", "go_down", "go_left". In order to add some variation, the player should also be able to use for loops.

In Fig. 1 is presented the syntax of a program:

```
For(10)

START
    {BLOCK OF INSTRUCTIONS}
END
```

Figure 1. Syntax example

The output consists of the maze itself and the representation of the player within the maze. The player moves in real time in accordance with the code, additionally, there are prompts for when the player loses, wins or for situations of stalemates.

Error Handling

Syntax errors are highlighted at compile time which prevent the program from running. The language supports variables and basic mathematical expressions which means that out of bounds errors and division by zero should be accounted for.

Another error that could occur on runtime is when the program enters an endless loop. This can be circumvented by putting a hard limit on the maximum number of iterations.

Reference Grammar

A Programming Language Grammar is a set of instructions about how to write statements which are valid for that programming language [2]. The instructions are given in the form of rules that specify how characters and words can be put one after the other, in order to form valid statements (also called sentences) [2]. The grammar for the domain specific language consists of a 4-tuple G = {S, V_N, V_T, P} where:

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

In Tab. 1 are represented the notations used for grammar specification.

Table 1

Notations used in language grammar definition

Notation	**Meaning**
<hello>	means that hello is a nonterminal symbol
hello	(in **bold** font) means that hello is a terminal symbol
x*	means zero or more occurrences of x
X^+	means a comma-separated list of one or more x's
|	means separate alternatives
→	means deriving
{}	means optional occurrences

S = {<source code>}

V_N = {<source code>, <set of affirmations>, <affirmation>, <function declaration>, <player>, <variable declaration>, < identifier >, <letter>, <digit>, <calculator>, <steps>, <call function>, <conditions>, <number>, <equal to>, <operation>, <direction>, <condition>, <variable>, <map>, <non-zero number>, <break>, <link>, <characters>}

V_T = {0, …, 9, a, …, z, A, …, Z, Start, End, Player, Hunter, var, for, while, break, if, else, case, Step, Right, Left, Up, Down, Function, width, length, labyrinth-layout, “_”, “&&”, “||”, “:”, “=”, “==”, “;”, “{”, “}”, “<”, “>”, “<=”, “>=”, “!=”, “(”, “)”, “+”, “-”, “*”, “/”, “””, ““”, “,”, “.”}

P = {
<source code> → <map> <source code>
<source code> → <function declaration>* **Start** <player> **{**<set of affirmations>**} End**
<function declaration> → **Function** <identifier> **(**<identifier>$^+$**) {**<set of affirmations>**}**
<identifier> → <letter> | {<letter> | <digit> | _} *
<set of affirmations> → < affirmation> | < affirmation><set of affirmations>
<affirmation> → <variable declaration> |
<call function> |
<calculator> |
<steps> |
<break> |
if <conditions> **{**<set of affirmations>**}** |
if <conditions> **{**<set of affirmations>**} else {**<set of affirmations>**}** |
while (<variable> | <number >**) {**<set of affirmations>**} |**
for (<variable> | <number >**) {**<set of affirmations>**} |**
case { <conditions> **:** {{}<set of affirmations> {}} }
<variable declaration> → **var** <identifier>**;**
<call function> → <identifier> **(**<identifier>$^+$**);**
<calculator> → <identifier> **=** <equal to>**;**
<equal to> → <identifier> |
<identifier> <operation> <equal to> |
<number> |
<number> <operation> <equal to>
<operation> → **+ | - | * | /**
<steps> → **Step** <direction>**;**
<direction> → **Right | Left | Up | Down**
<break> → **break;**
<conditions> → <variable> <condition> <variable>
<variable> → <identifier> | <letter>
<condition> → **&& | || | == | < | > | <= | >= | !=**
<number> → <digit> | <digit> <number>
<non-zero number> → <no 0 digit> | < no 0 digit > <number>
<letter> → **a** | **b** | **c** | … | **A** | **B** | **C** | … | **Z**
<no 0 digit> → **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
<digit> → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
<player> → **Player** | **Hunter**
<map> →
import <link> |
{width: <non-zero number>**, length:** <non-zero number>**, labyrinth-layout: “**<number>**”}**
<link> → **“**<characters>*”
<characters> → **0 | … | 9 | a | … | z | A | … | Z | _ | @| # | $ | % | : | = | ; | { | } | < | > | ! | (|) | + | - | * | / | . }**

Parsing example

```
“
import “map.json”
Start Player {
        var six
        six = 6
        For(six) {
            Step Right
        }
}
”
```

<source code> → <map><source code> → import <link> <source code> → import “map.json” <source code> → import “map.json” Start <player> {<set of affirmations>} End → import “map.json” Start Player {<set of affirmations>} End → import “map.json” Start Player {<affirmation><set of affirmations>} End → import “map.json” Start Player {<variable declaration><set of affirmations>} End → import “map.json” Start Player {var <identifier><set of affirmations>} End → import “map.json” Start Player {var six <set of affirmations>} End → import “map.json” Start Player {var six
<affirmation><set of affirmations>} End → import “map.json” Start Player {var six <calculator><set of affirmations>} End → import “map.json” Start Player {var six <identifier>=<equal to><set of affirmations>} End → import “map.json” Start Player {var six six = <equal to><set of affirmations>} End → import “map.json” Start Player {var six six = 6 <set of affirmations>} End → import “map.json” Start Player {var six six = 6 <affirmation>} End → import “map.json” Start Player {var six six = 6 for<number>{<set of affirmations>}} End → import “map.json” Start Player {var six six = 6 for six{<set of affirmations>}} End → import “map.json” Start Player {var six six = 6 for six{<affirmation>}} End → import “map.json” Start Player {var six six = 6 for six{<steps>}} End → import “map.json” Start Player {var six six = 6 for six{Step <direction>}} End → import “map.json” Start Player {var six six = 6 for six{Step Right}} End

Derivation Tree

In Fig. 2 is presented the derivation tree based on the parsing example presented in the previous section.

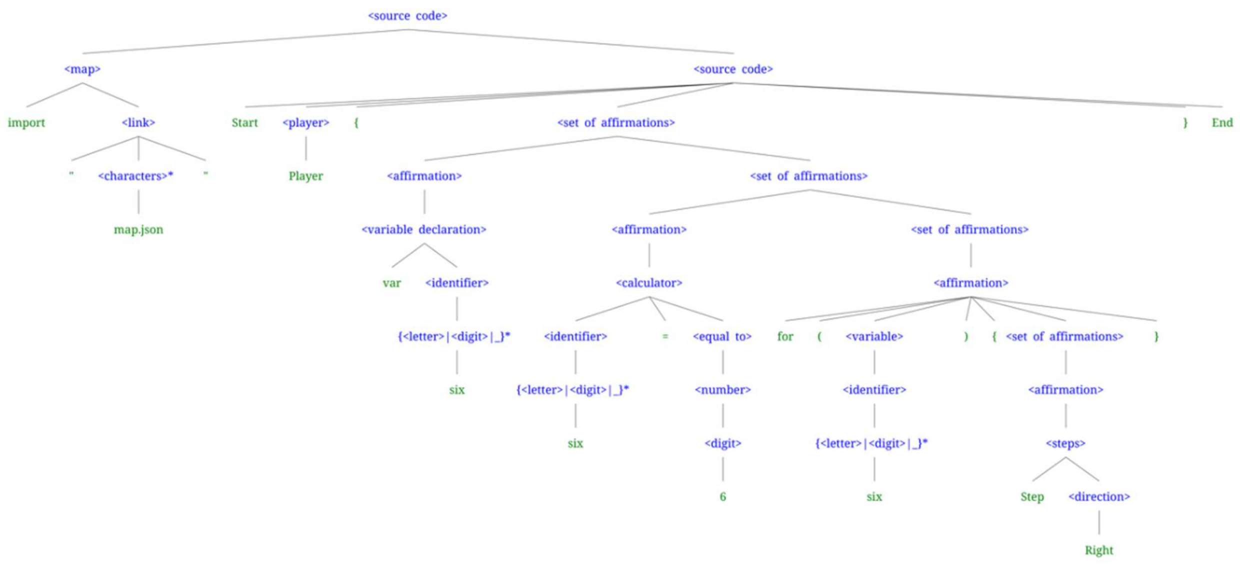

Figure 2. Derivation tree

Conclusions

The purpose of this project is to show the concepts of a DSL which can be used in educational purposes. The presented DSL intended for Children's Maze Game differs from others by its simplicity and functionality. The grammar of the language is intuitive and user-friendly, which is one of the main cafeterias to keep children's attention. Combining programming with gaming and education provides an effective and fun environment for learning.

References

1. Data Structures. Using a two-dimensional array for a game. [online]. [accessed 15.02.2022]. Available: https://www.bbc.co.uk/bitesize/guides/z4tf9j6/revision/7
2. Compilers. What is a programming language grammar? [online]. [accessed 16.02.2022]. Available: https://pgrandinetti.github.io/compilers/page/what-is-a-programming-language-grammar/