COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

EDUCATIONAL PROGRAMMING LANGUAGE FOR CREATING INTERACTIVE WEB APPLICATIONS

Diploma thesis

Bratislava, 2017 **Bc.Veronika Pohorencová**

COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

EDUCATIONAL PROGRAMMING LANGUAGE FOR CREATING INTERACTIVE WEB APPLICATIONS

Diploma thesis

Study programme: Applied Computer Science
Field of study: 2511 Applied Informatics
Department: Department of Applied Informatics
Supervisor: Mgr. Pavel Petrovič, PhD.

Bratislava, 2017 **Bc.Veronika Pohorencová**

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Veronika Pohorencová
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Applied Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Educational Programming Language for Creating Interactive Web Applications

Aim: Programming languages and IDEs for creating web applications are not yet sufficient. Designers have to master different technologies that change too fast, few languages are suitable for children. Simplification and unification is expected soon. This thesis is a contribution to this effort. The aim is to design a new language for creating web applications. One language will be used for both server and client components, communicating (sending both data and code) with each other. An API for controlling the rendered webpage will be designed. A full abstraction over HTML and other technologies is assumed, the result being a clean, universal and concise text-based programming language easily understandable to learners. It will be possible to create a complete multi-user web application with persistence and graphical user interface utilizing interactivity in bi-directional communication between the server and the client. The focus will be on designing an easy to understand, elegant, minimal although universal language. The thesis will contain an overview and analysis of suitable technologies and examples of multi-user applications written in the designed language.

Literature:
Young, A. R., Harter M. (2014) Node.js in Practice, Manning Publications.
Blaho, A., et al. (2001) Imagine Reference Guide, Logotron Ltd.
Dart Programming Language Specification (2015), ECMA International.
TypeScript Language Specification (2015), Microsoft Corporation.

Keywords: programming language for children, interactive web application, multi-user

Supervisor:	Mgr. Pavel Petrovič, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	29.09.2015
Approved:	09.12.2015

prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

Student

Supervisor

Declaration of Authenticity

I declare that this Diploma thesis is my own work with using the sources listed in the bibliography and with help of my supervisor.

….....

In Bratislava, 4.5.2017

Acknowledgement

I would like to thanks at first my supervisor Mgr. Pavel Petrovič, PhD. for valuable advices, opinions and inspiration and also my family, friends for mental support.

Abstract

The main goal of our work was to design and develop a new programming language for creating interactive web applications. The result of our work is a simple, yet universal text-based programming language Webi that is easy to understand for children. In our language it is possible to program distributed multi-user web applications with graphical user interfaces that run simultaneously on both a server and user clients. One universal language is used for all server and client components. This unique feature is the main contribution of our work: instead of learning and working with many different chaotically evolving technologies as required to develop web applications today otherwise, the programmer now only needs to learn and use one programming language allowing him to write complete applications running in a distributed execution environment on the web. This work should be understood as a proof of concept of our ideas on how the development of web applications should look like in the future. Our language is object-oriented, statically typed and provides fundamental elements for working with structured data. We tried to avoid the known pitfalls of other languages such as semicolon statement terminators/separators found in Pascal, C, and Java as well as Python-like indenting requirements. It contains additional features, all designed to guide the user in writing clean code. We provide a complete description of this new language, a syntax parser based on complete syntax graphs, builder of an internal executable representation, and an interpreter. The whole process that helps us to accomplish our goal is described in our thesis: analysis of suitable technologies, analysis, design and implementation.

Keywords: programming language for children, interactive web application, multi-user

Abstrakt

Hlavným cieľom našej práce bolo vytvoriť nový programovací jazyk na vytváranie interaktívnych webových aplikácií. Výsledkom našej práce je jednoduchý, univerzálny, deťom zrozumiteľný textový programovací jazyk Webi. Jazyk je určený pre programovanie viacužívateľských webových aplikácii s grafickým požívateľským rozhraním, využívajúcich komunikáciu medzi serverom a klientom. Jazyk je jednotný pre serverovské i klientské komponenty. Hlavným prínosom našej práce je, že namiesto študovania a pracovania s viacero rozličnými technológiami, ako je to dnes vyžadované pri vývoji webových aplikácií, programátor môže teraz využiť len jeden programovací jazyk. Naša práca by mala poukázať aj na to, ako by mohlo vyvíjanie webových aplikácii vyzerať v budúcnosti. Náš jazyk je objektovo orientovaný, so statickou typovou kontrolou a poskytuje základné prvky pre prácu s dátovými štruktúrami. Pri návrhu jazyka sme sa snažili vyhnúť nástrahám iných jazykov ako napríklad koncový/oddelovací znak bodkočiarka, ktorý nájdeme v jazyku Pascal, C, Java alebo nutné odsadzovanie v jazyku Python. Jazyk obsahuje aj ďalšie vylepšenia, ktoré majú naviesť užívateľa k písaniu čistého kódu. Vytvorili sme podrobný popis nášho jazyka, parser založený na syntaktických diagramoch, builder internej repzrezentácie a interpreter. V práci je opísaný celý proces, ktorý nám pomohol naplniť náš cieľ: analýza, návrh a implementácia. Súčasťou práce je aj prehľad a analýza vhodných technológií.

Kľúčové slová: detský programovací jazyk, interaktívna webová aplikácia, viac-užívateľská

Content

1 Introduction

Nowadays, the computer programming is increasingly important. More code will be needed in the future, but one problem is, that the world doesn't have enough programmers. Due to the progress of computer technology, the programming skills of some sort will soon be used in everyday life by all people. Most adults still do not know what is programming and it is still quite recent that programming was added into schools and it is true only in some countries. To inspire kids to develop coding skills and start programming is a way to get them on good career path. A great way how to start is to learn languages that were developed for kids. A lot of kid's languages are designed using the idea that the children are learning programming with dragging bricks and dropping them into some workspace. When the children progress in this way of programming, the good way how to continue is to start learning text-based programming languages. But, there is not a lot of programming languages for kids that are text-based. Every few years a new hardware or software platform is introduced to the market. Programming applications that are designed for a particular platform is not going to be a very smart option in the future, and it is not one already today. The most flexible platform are applications running in the web browser, the most wide-spread application that is available virtually on all platforms used today. However, programming web applications even quarter a century after the large world-wide-web revolution still requires the use of multiple cumbersome quickly changing technologies and thus becomes a nightmare even for a seasoned IT professionals not to mention novice programmers. We feel a strong need to fill this gap with a simple elegant universal computer language that contains all technologies needed to write multi-user web applications running in a distributed computational environment. Our webi programming language give a new possibility for youngers, but also adults to start learn text-based programming language that is similar to "real" languages. In our language the user can program multi-user web applications with graphical user interfaces that use communication between server and client. Our language is a good jumping point for the future programmers.

2 Theoretical Background

This chapter includes a theoretical background that will be needed and helpful to achieve the goals of our thesis. First, we describe how works compilers and interpreters and then we analyse similar systems and suitable technologies for creating web applications.

2.1 Programming Languages

This chapter is written based on reading the books **Compilers Principles, Techniques & Tools** (1) and **Basics of Compiler Design** (2).

2.1.1 Compilers and Interpreters

Programmers write a code (source program) in programming languages understandable for humans, in high-level languages. But the source code needs to be converted to machine code, understandable for computers and this is accomplished by compilers and interpreters. A compiler is kind of language processor that reads a program in a source language and translates it into an equivalent program in a target language. An interpreter is another kind of language processor that reads and executes the instructions specified in the source program of language. Programming languages like C, C++, Pascal use compilers and languages like Python, Ruby use interpreters. Compilation and interpretation may be combined, as the example is Java language, where the compiler converts the source program into an intermediate form called bytecode. Then the bytecode is interpreted by a virtual machine and converted into a machine code. The next possibility is that parts of a program are compiled to machine code, some parts are compiled to intermediate form that is interpreted at runtime while other parts, for instance a syntax tree, may be kept and interpreted directly.

A compiler is composed of three main phases: frontend, middle part and backend. Frontend consists of lexical, syntax and semantic analyzer. In the middle part, the frontend generates an intermediate representation. The last three phases register allocation, machine code generation, assembly and linking belong to the backend. The order, combination or split of phases, may be different in some compilers.

An interpreter composes of lexical, syntax and semantic analyzer as compiler. But then instead of generating code from the syntax tree, an interpreter's backend is processing the syntax tree to execute the program (evaluate expressions and execute statements).

In summary, the main component of the compiler's backend is generator and the main component of interpreter's backend is executor.

2.1.2 Lexical Analysis

The first phase of the frontend, lexical analysis, is also called scanning. The main role of this part is to read the source program and break it up into meaningful sequences of characters called lexemes. For each lexeme, the lexical analyzer produces a token that is sent to syntax analyzer, when requested. Before the lexical analyzer will send a token, it may need to look ahead some characters to recognize the correct token. The lexical analyzer produces tokens in the following form: (token-name, attribute-value), where token-name is an abstract symbol (used in the syntax analysis) and attribute-value points to the symbol table (data structure, which holds the information about the specified token). If a token does not need to hold attribute-value in symbol table then it is omitted (i.e. token "=" does not need to hold attribute-value, but token that represent an identifier needs to hold information about its name or type). The symbol table is created and used during lexical, syntax and semantic analysis. Data structure of symbol table should be designed to find and save record quickly. If the lexical analysis during scanning detects that the source program is incorrect, then it generates an error message to inform the programmer. The lexical analysis may also perform other tasks, i.e. skipping comments and whitespaces (or other characters that separate tokens), counting the number of tokens and newlines that may be used it to associate error messages with line numbers.

2.1.3 Syntax Analysis

The second phase of frontend, syntax analysis, is also called parsing. The parser uses the string of tokens, produced by the scanner, verifies them syntactically, generates a parse tree (or syntax tree) and reports syntax errors.

2.1.3.1 A Context-Free Grammar

A context-free grammar, or simply grammar is used to describe the syntax of programming language. The grammar is specified by listing of production rules. Every production rule consists of a start symbol, terminals and nonterminals. For example, assume we have the *while* statement with the following form:

while (expression) { statement }

When we use *expr* to denote an expression and *stmt* to denote statement then we can get the following rule:

stmt → **while** (*expr*) { *stmt* }

that is called a production with nonterminals (called left side or head), an arrow and sequence of terminals/nonterminals (called right side or body). In this example a keyword *while*, parentheses and curly brackets are terminals (elementary symbols of the language). The nonterminals are *expr* and *stmt* (a set of strings of terminals) and the start symbol is *stmt*. The empty string, that has zero terminals, is written in grammar as "e". Productions with the same head can have their bodies grouped, separated by the symbol |, which means "or". For example expressions that consist of digit and plus/minus signs, the production with grouped bodies is

list → -+ { *list* + digit | *list* - digit | digit }

where *list* represents a list of digits separated by plus or minus.

To generate a sequence of terminals from a defined grammar we begin with the start symbol and then we replace a nonterminal by the body of a production rule specified for that terminal. We repeat this process of replacing a nonterminal until all nonterminals will not be replaced by terminals. String of terminals that cannot be derived from the start symbol of grammar is not recognized by the grammar.

2.1.3.2 Parser Trees

A parse tree graphically represents the process of deriving a string of terminals from a context-free grammar. The tree consists of one or more labeled nodes, typically the label corresponds to grammar symbols. The root of the tree is labeled by the start symbol, leaves are labeled by a terminal (or by empty symbol "ε") and interior nodes are labeled by a nonterminal. The leaves, from left to right represent the *yield* of the three (string derived from the nonterminals). As example the derivation of 9-7+1 (according to the grammar that was explained in section above) is represented by the tree shown at Figure 1.

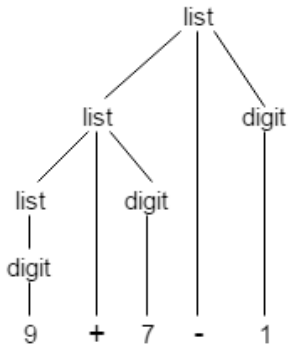

Figure 1: Parse Tree

where the root is labeled by *list* (the start symbol in production), the children of the root are labeled by *list, +, digit* (from left to right) etc. The yield of this three is 9+7-1. This process of finding a parse tree for specified string is called parsing. If a grammar produces more than one parse tree generating a given string, it is called ambiguous. It means that we can find string that is a yield of more than one parse tree. It is important for most parsers that grammar is unambiguous, so we need to design unambiguous grammar or we can add rules to resolve the ambiguities of an ambiguous grammar.

2.1.3.3 Abstract syntax trees

An abstract syntax tree or simply syntax tree consists of nodes, where interior nodes represent an operator and children of the nodes represent the operands for operator. Difference between syntax and parse trees is that the interior nodes of syntax trees represent programming constructs and in the parser trees represent nonterminals. This nonterminals are also constructs, but some of them are helpers that typically are not needed in syntax trees. On the Figure 2 is shown the abstract syntax tree for expression 9+7-1.

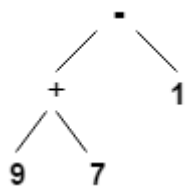

Figure 2: Syntax Tree

2.1.3.4 Parsing methods

There are three general parsing methods - universal, top-down and bottom-up. Universal methods are e.g. Cocke-Younger-Kasami algorithm and Earley's algorithm. But most parsing methods belongs to group of top-down or bottom-up methods, because are efficient. In top-down parsers construction of parser trees starts from the root to leaves, contrary in bottom-up parsers it is from leaves to root.

2.1.4 Semantic Analysis

The last phase of frontend, semantic analyzer, checks the source program semantically using the syntax tree and the symbol table. An important role of semantic analyzer is type checking. Semantic analyser needs to check if type of identifiers is the same as type of values that are assigned to them, if identifiers are declared before are used, if the returned value of function is same like in function definition, if the source code does not include two function definitions with the same name, if the structure includes the

proper type etc. To check the source program the semantic analyzer uses more than one symbol tables, i.e. one table which hold variables with their types, another table for holding method's names with types of returned values etc.

2.2 Similar Systems

2.2.1 Imagine Logo

One of the most popular children programming langauges still used in Slovakia is **Imagine Logo** (3). It is a Logo dialect, which itself is a dialect of old-fassion LISP language. Thus it is limited to list data type for expressing any non-trivial data, which appears to be an unpleasant limitation that we will try to avoid. On the other hand, the language is free from semicolon separators/terminators and allows the use of an arbitrary whitespace. We would like to adopt the same strategy. A strong disadvantage of Imagine is that it is bound to the Windows platform, our language will be web-based and thus available to users with all devices. Imagine Logo contains some support for object-oriented programming, but it does it in somewhat unconventional manner. Our language will be pure object-oriented language (in a sense of SmallTalk, where everything is an Object), but it will resemble Java programming language due to its use of static typing, classes, and interfaces. Imagine Logo is very inspiring because it is based on graphics - the graphical canvas is in the center of the interest of the user/programmer, and we feel this is the right choice and follow the same idea. Figure 3 shows how looks IDE of Imagine Logo.

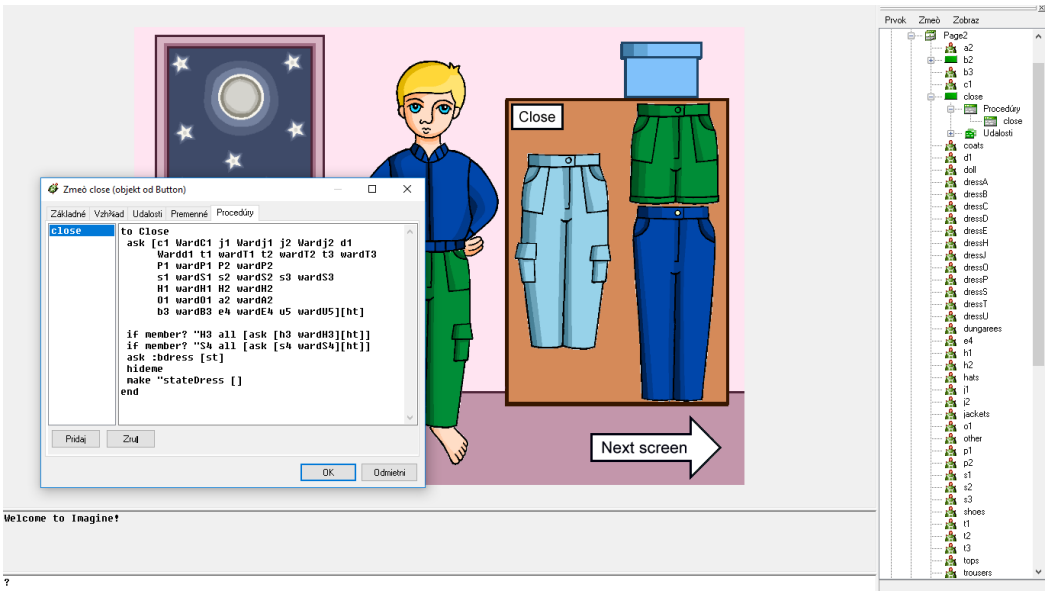

Figure 3: Imagine Logo IDE

2.2.2 KidsRuby

KidsRuby (4) is a free software that offers a version of the Ruby programming language for younger users and a set of tutorials. It is an object-oriented programming language. Kids Ruby is inspired and follows Hackety-Hack, an older program for kids to learn Ruby. Main window consists of command line structure, where the user writes code and see numbers of lines and submenu with sections Help, Output and Turtle. Section Help includes a list of lessons, sections Output and Turtle show result of program and Output also shows error messages. When the user run code, an output is visible in real-time. The user can also save or open your program and clear your code.

The section Help consists of items:

- **Start Here:** explains what is Ruby and how software works
- **KeyBoard Shortcuts:** includes a list of default keyboard shortcuts
- **Hackety-Hack:** includes an edited version of the lessons from Hackety-Hack, that explain what is programming, how to draw shapes with turtles, basics of Ruby: displaying on the screen, strings, numbers, objects, variables etc.
- **Ruby Warrior:** explains how to play a game Ruby Warrior designed to teach Ruby, where the user is a warrior and need to write a Ruby script to instruct the warrior.
- **Make Games With Gosu:** includes a series of lessons to show how to create game using Ruby and a game package called Gosu
- **Robots:** Kids Ruby allows program the Sphero - robotic ball, this section includes short tutorial and a list of main commands to control sphero
- **Glossary:** includes list of definitions that the language uses

All lessons in KidsRuby consist of short explanation on given theme and also example of code that can be switched to text mode and then the user can easily copy code.

Figure 4 shows how looks game which user can program using a series of lessons from section Make Games With Gosu. These lessons in steps explain how to open the window, draw a player, move player when arrow keys are pressed, limit player's movement to bounds of window, creating multiple balls, move balls, stop game when the ball hits a player and reset the game. Finally the user can play dodge ball with your player.

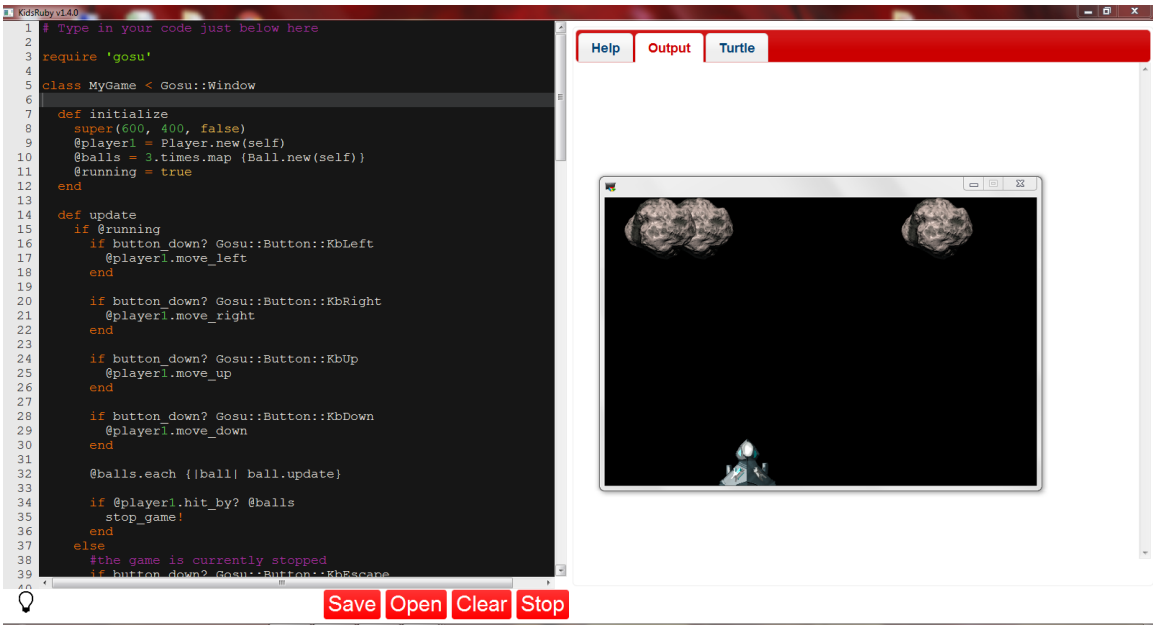

Figure 4: KidsRuby – game with Glosu

KidsRuby is an interesting software that can help younger users start programming. An advantage is a set of lessons that explain language fundamentals and show them on examples of code. The section, which shows how to create a simple game, is very helpful. Users are motivated and entertained while they learn how to create their own game. Another advantage is that the software allows programing a robotic ball and it can be interesting for children to see their code run on a real robot. A disadvantage is that error messages don't work during the programming with turtle. Then, while the program starts running it takes longer time and the user doesn't see any informative message to know that something is happening. The software includes a lot of text that can cause children lose interest and could be better structured with better graphics. The main inspiration for us will be the section "Make Games With Gosu", it is a good idea to add something like this to our software.

2.3 Related Technologies

2.3.1 JavaScript

JavaScript (5) is a high-level, prototype-based, untyped and dynamic scripting programming language for developing web applications. JavaScript is used to program how the web applications behave. It runs on the client side, but also can be used on server side, for example in Node.js. It gives the possibility to code in object-oriented style (supports most of the object oriented concepts) and also in functional programming style

(function can be assigned to a variable, can return another function, can be passed to another function as parameter).

2.3.2 Web Workers

In JavaScript, multiple scripts cannot run at the same time, but we have option to achieve parallelism by using Web Workers. **Web Workers** (6) represent parallel threads in Javascript and specially are used to allow execute the long task without yielding or to process a big amount of data. They allow that long-running scripts are not interrupted by scripts that respond to user interactions. One limitation is that it is not allowed to access DOM elements inside Web Workers.

2.3.3 TypeScript

TypeScript (7) is a programming language used to develop JavaScript applications for client and server side (Node.js) and compiles to Javascript. It is typed superset of Javascript, any JavaScript programs are also valid TypeScript programs. It starts from the same syntax and semantic, but adds a new features such as: type annotations, interfaces, enums, tuples, namespaces etc. TypeScript is more similar to languages like Java, C# than to Javascript.

2.3.4 Dart

Dart (8) is an application programming language that runs fast in browsers, on the command line, on servers and on mobile devices. It is designed to catch errors easier with earlier error detection. Some of major features are: Dart is object - oriented, everything can be assigned to a variable is an object, even numbers, functions, null are objects, adding static types to variable declarations is optional, every application must have a top-level function main(), which is the entry point to the application. Easier way to start using Dart is DartPad, open-source online tool to help developers learn about Dart language and libraries.

2.3.5 Node.js

Node.js (4) is a JavaScript runtime environment for writing server side Javascript applications. It uses an event-driven, non-blocking I/O model, that makes it perfect for real-time applications. In contrast to blocking I/O model, the non-blocking model allows for multiple I/O operation to be performed independent of other system operations and it allows to build efficient applications.

2.3.6 Socket.io

Socket.io (10) is one of the most powerful, speed Javascript frameworks, that enables bidirectional event-based real-time communication. It supported on every platform, device and browser.

2.3.7 Bootstrap

Bootstrap (11) is a framework for developing responsive web applications, includes languages HTML, CSS and JavaScript. It is simply, intuitive, easy to learn and helps developers to programming front-end faster. An advantage is that bootstrap supports responsive design, and makes the application look good on any screen (desktop, tablet, mobile phone). The layout of the application is performed dynamically. Resizing, hiding or moving content depends on which device is an application running. Nowadays, Bootstrap offers a lot of templates of different styles that is easy to use or rearrange.

2.3.8 Technologies for distributed systems and remote procedure calls

Java Enterprise Edition is a wide collection of technologies, libraries, tools, and API for building large enterprise solutions. As such it provides multiple technologies that are of a high interest to our project. In particular it defines API for remotely calling methods of objects instantiated on a different computational node - such as JAX-WS that is based on SOAP protocol and allows automatic generation of plain Java classes that represent web services on the server. The programmer of the client cannot distinguish between calling methods of local and remote objects. In a similar way, JAX-RS allows the same type of access to REST-based web services. This is exactly what our programming language will provide, however without the burden of complicated configuration of these technologies.

Similar packages can be found in Node.js, worth mentioning are the NodeRMI and RMI.js packages.

3 Specification of Webi Programming Language

The language is developed for creating a complete multi-user web applications with implicit bi-directional communication between the server and the client. The same language is used to write code for both server and client components that run in distributed environment and communicate with each other. It is a clean, universal and concise text-based programming language, easily understandable for children. This chapter includes detailed specification of language - what the language offers, syntax of language, rules and examples of code written in our language.

3.1 Program Code

The program consists of a series of statements. Use of the whitespace (space, tabs, new lines) is arbitrary. Statements can contain expressions. Every expression evaluates to some value. Statements are of the following kinds: variable declarations, assignment, print, program flow statements (conditions, loops), return statement, and calling of built-in and user methods that do not return values. Expression types include: arithmetical, comparison, logical, and string expressions, variables, direct values, and calls of built-in and user methods that return values. The language is case-sensitive and the language-defined identifiers and keywords must be spelled properly. Variables are distinguished from all other expressions by capitalizations: variable names may not contain lowercase letters. Other identifiers can use an arbitrary capitalization (except of all capitals), but the convention, which is used in the built-in language features, suggests that class and interface names start with a capital letter and method names start with a lowercase letter. For both, the Pascal Casing it used. Two tokens must be separated by whitespace if their concatenation could be interpreted as a different token otherwise.

3.2 Comments

The comment is a part of code that is ignored by the interpreter. Comments start and end with character „#“.

3.3 Data Types

The basic data types are Number, String and Boolean. Strings are mutable (as contrasted to Java strings, see Section 4.5). In our language the numbers are implemented as instances of Number class. The class Number provides several useful methods that operate on numbers. This also apply for strings and booleans, they also have their

associated classes String and Boolean. The number type represent any real number. It can have an integer value or float value and the programmer does not specify the type of number exactly. The string type represents values formed as a sequence of arbitrary UTF-8 characters. The string is recognized in the code by double quotes “ ”. A string can be empty. The boolean type has two values true and false. The built-in structured mutable data types are: List, Dictionary, Json and Image. Values of these data types are objects and have their associated classes. Elements of a List can be of a different type. A direct List value is specified as ordered sequence of comma-separated values surrounded by square brackets. The elements of List can be accessed directly by the index, see below for more options. A Dictionary is a set of key: value pairs, it is indexed by a key. In the direct Dictionary value, each pair is surrounded by curly braces. All keys within the same Dictionary need to be of the same type. An attempt to access a non-existent element of a structured type value results in a run-time error and enters a debugger. A Json type is useful to hold data with a more complex structure without having to specify a special class for them. Json values can be easily saved/retrieved to/from files. An advantage is that Json can be viewed and modified by a human in a text editor. Json direct value is written as a list of name : value pairs separated by commas surrounded by curly braces. Name is an arbitrary string and value is a direct value of any of the following types: Number, String, Boolean, another Json or a List. List, Dictionary, and Json can be empty. An Image represents graphical images. It can be loaded by specifying a file path or URL. Its width and height in pixels can be retrieved. Unlike Java and some other languages, our language does not contain null values. If a variable is declared without initialization, it contains a so-called “default” value. For built-in types, these are: 0, false, zero-length empty string, empty list, dictionary, json, and 0x0 image. For other types, they are objects, where all object variables have default values.

3.3.1 Data Types Examples

List

```
>> List VALUES = [1, 2, 3, 4]
>> print VALUES[0]
1
>> print VALUES[VALUES.length() - 1]
4
>> print VALUES[1:3]
```

[2,3,4]

>> print VALUES[:2]

[1,2,3]

List modification

>> VALUES[1] = 5

>> print VALUES[:2]

[1,5,3]

Dictionary

>> Dict NAMES = {1: ”Emil”, 4: “Karol”}

>> print NAMES[1]

Emil

>> print NAMES[4]

Karol

Dictionary modification

>> NAMES[4] = “Adam”

>> print NAMES[4]

Adam

Json String value

>> Json NAME = {“firstname” : ”John”}

>> print NAME.firstname

John

Json Number value

>> Json INFO = {“age” : 15}

>> print INFO.age

15

Json Boolean value

>> Json PERSONAL = {“married” : true}

>> print PERSONAL.married

true

Json List value

>> Json PERSONS = {

"doctors": ["John", "Adam"],

"teachers" : ["Peter", "Pavol", "Jan"]

}

```
>> print PERSONS.doctors
John, Adam
>> print PERSONS.teachers[0]
Peter
```

Json Object value

```
>> Json EMPLOYEES = {
"doctor": {"firstname":"John", "lastname":"Snow", "age":30}
}
>> print EMPLOYEES.doctor.age
30
```

3.4 Variables

Variable represents a storage for a value. It's name is a sequence of uppercase letters and digits, but digits can only be used as a suffix. Variables must be declared before they are used and always have a type. Declared variables are automatically initialized to default values, which is 0, false, or empty string, list, dictionary, json or an image. The value can be stored in the variable using assignment operator "=" in an assignment statement. There are three types of variables: object properties, object shared properties, and local variables. Similarly to the language Java, there are no global variables. Object variables are stored inside objects and they can be accessible by the methods called on their own object only. The language offers also constant variables. The value of a constant variable cannot be changed, it is specified in the initializing declaration.

3.4.1 Examples of expressions and statements with variables

Declaration of variable

```
>> Number A
>> String WORD
```

Declaration with initialization

```
>> Boolean LOG = true
```

Assign value to variable

```
>> VALUE = 5.2
>> A = -10
>> B = VALUE + A
>> WORD = "ahoj"
```

Constant variable

>> constant Number A = 2

3.5 Operators

The language contains arithmetic, string, assignment, comparison and logical operators, and structural data indexing operators. Operators take two (binary) or one (unary) arguments. The operands for arithmetic operators are numbers and the returned value is also a number. The operators ++ and -- are only shortcuts of a combination of assignment statement with addition/subtraction expression, giving an impression of built-in increment/decrement statements. They return no value and they cannot be used in expressions. The string concatenation operator "+" is used to add strings or a string with a number. It returns a new string without modifying the original values. When the concatenation operator is applied on a pair of operands of types string and number, the number is always automatically converted to a string first. The assignment operators assigns the value of its right operand to its left operand. The comparison operators compare values of two numeric, or string operands and return the result as a boolean value. The operators "==" and "!=" allow operands of any type, if their type is the same. They compare values, not references, by calling method *Object.equals(Object other)* or overridden method. References can be compared using method *Object.is(Object OTHER)*, which returns true, if other contains a reference to the very same object. Logical operators return boolean value and are used with boolean operands. All operators are evaluated from left to right, if they have the same precedence. Otherwise, an operator with the higher precedence is evaluated first. Figure 5 below lists all operators in the order of their precedence from the highest to the lowest. Operators in the same line have equal precedence. The parentheses may be used to modify the default order of evaluation.

Operators	Name
.	object's method and variable access
[]	structured data indexing operator
* / %	multiplication, division, modulo
+ -	addition, subtraction
< > <= >= == !=	comparison operators
&& || !	logical operators
= += -= *= /= %=	assignment operators

Figure 5: Precedence of operators

3.5.1 Operators Examples

Arithmetics operators +, -, *, /, %, ++, --

>> 5.2+4

9.2

>> 5%2

1

String operator +

>> "hello" + " " + "Anna"

hello Anna

>> "5" + 2 + 3

523

Assignment operators =, +=, -=, *=, /=, %=

>> A = -10

>> A += 5

>> print A

-5

Comparison operators <, > , <=, >=, ==, *!=*

>> 5 <= 4

false

>> String X = "4"

>> String Y = "hello"

>> X != Y

true

Logical operators &&, ||, !

```
>> true || false
true
```

Evaluation of expressions

```
>> 5+2*4
13
>> (5+2)*4
28
```

3.6 Statements

Statements are instructions that perform a specific action. The language contains simple statements and structured statements. Structured statements may contain sequences of statements as their bodies. The language includes simple statements: variable declarations, assignment, program flow statements (conditions, loops), return statement, and calling of built-in and user methods that do not return values.

The statement return ends the execution of a method and optionally returns a value. The returning value must be of the type specified in the method header.

The structured statements are if-else, for, foreach, while, with-do statement. The if-else statement is used for conditional execution. If a specified condition evaluates to true, then the specified sequence of statements is executed. If the condition evaluates to false, the other, optionally specified, sequence of statements after the else keyword is executed instead. The if statement is terminated by the end keyword. For brevity, the structured command on the left can be written as shown on the right with the help of the elseif keyword:

```
if condition1                    if condition1
 statements1                     statements1
else                              elseif condition2
 if condition2                    statements2
  statements2                   else
 else                               statements3
  statements3                   end
```

end

end

Remaining statements are for creating loops, to perform a sequence of statements repeatedly. The for statement executes a sequence of statements number of times. The foreach iterates over the iterable object (string, list). The while statement is executed while a condition evaluates to true. The condition is an arbitrary expression that evaluates to a boolean. The with-do statement tries to match the value of a specified expression with the value of one of the provided alternative expressions. It executes statements of every matched option.

3.6.1 Statements Examples

Statement return

```
>> Number A = 5
>> return A + 4
9
```

Print the values

```
>> Number A = 5;
>> print “the number ” + A + “is greater than ” + (10 - 6)
the number 5 is greater than 4
>>print A
5
```

If-else statement

```
>> String A = “ano”
>> if A == “nie”
        A = “ano”
   elseif A == “ano”
        A = “nie”
   else
        A = “neviem”
    end
>> print A
“nie”
```

For statement

```
for i in [0,10,2]     0-start, 10- end, 2 - step….return 0,2,4,6,8,10
```

for i in [10,0,-2] 10-start, 0- end, -2 - step….return 10,8,6,4,2

for i in [4,10] 4-start, 10- end...return 4,5,6...10

for i in [10] 10-end….return 0,1….10

```
>>  Number A = 0
>> for I in [0,10,2]
A += I
   end
>> print A
30
```

Foreach statement

```
>> foreach I in “aha”
          print I
   end
a
h
a
```

While statement

```
>> Number A = 0
>> while A < 5
          A += 2
   end
>> print A
6
```

With-do statement

```
>> TEST = 0
>> List VALUES = [14,4,8]
>> with VALUES[0] as X do
          on [X < 10] TEST ++
          on [X / 2 == 0] TEST += 2
          on [X == 6] TEST --
          on [X / 7 == 0] TEST += 3
    end
>> print TEST
5
```

3.7 Class

The objects are defined by its class. The class defines a type of objects, i.e. a set of attributes that characterize its objects and set of methods that operate with the data that object represents. Attributes store data for the object and methods are functions and procedures that allow performing actions with the object. Our language does not allow creating functions without classes, the only type of functions are methods that are called on objects. The source code of the program consists of at least two class declarations. Each such declaration creates a new definition of a class. The name of the class follows the keyword class. The class and object variables that store data for object must be declared in lines that follow the class definition before the first method definition. To distinguish class and object variables, the keyword shared is added in front of the type of the class variables - i.e. those whose value is shared among all instances of that class. The method with the same name as the name of its class is called constructor. This special method is called when creating a new instance of specific class. Every class must include a constructor, otherwise the program will not be compiled. One class can have more than one constructor. If the class has multiple constructors, they must be distinguishable by the number or types of arguments. A method definition consists of the return type, method name, and comma-separated list of arguments - each preceded by its type and a space. The methods that do not return values ("procedures") are defined with the return type Void. All methods are always public, they can be accessed from their own class, but also from the other classes. To create an instance of a class, call its constructor and pass the required arguments. If an object variable is declared without initialization and constructor call, its variables are filled with default values. However, it is recommended to aVoid this practice except of the cases where the variable would contain a null value, which is not part of this language[1]. To access an object's variable use operator dot with object and with the name of the variable. To access the class's variables from inside or outside the class use the class name with operator dot and with the name of a variable. Object and class variables can always only be accessed from within the methods of its own class or its subclass. Objects are removed from the memory automatically, when they are no longer being used.

[1] In fact, objects with the default values can internally be implemented as a null, until their first modification takes place.

3.7.1 Example of Class

```
>> class Student
        shared Number COUNT = 0
        Number AGE
        String NAME
        String CLASSROOM

        Student(String NAME, Number AGE, String CLASSROOM)
                NAME = NAME
                AGE = AGE
                CLASSROOM = CLASSROOM
                COUNT += 1
        end

        Void changeName(String NEW)
                NAME = NEW
        end

        Boolean isOlder(Student ANOTHER)
                return AGE > ANOTHER.AGE
        end
>> Student S = Student(“Veronika”, 23, “A”)
>> print S.AGE
23
>> Student P = Student(“Janko”, 31, “B”)
>> print COUNT
2
>> S.isOlder(P)
false
```

3.8 Passing by Value or Reference

When we call class methods and pass arguments into these methods, here are two ways how they can be passed. The arguments can be passed by value or by reference. In our language arguments are always passed by reference. It means that the local variable in

the method contains a reference to the same object that was passed in. Then changes on object inside a method modify the object that was passed in. But if the programmer needs to create a new object as a copy of an existing object, he may do it by calling a copy constructor. The inherited copy constructor Object(Object OTHER) performs a deep copy of all fields. If a different behavior is prefered, another copy constructor has to be provided and implemented in the respective class. If it is not, a copy constructor of the superclass that is closest in the hierarchy, is used instead, eventually, the one from the Object class. The parameter of this constructor is a reference to an object of the same type as is the type of the object that will be created. As every other constructor, a copy constructor is responsible for filling all fields of the new instance. If it leaves some fields untouched, they will contain the default values. An example below explains this case using the already defined class Student from section Class.

3.8.1 Passing by Value or Reference Examples

Object passed as reference

```
Void  assignMyClassTo(Student ANOTHER)
        ANOTHER.CLASSROOM = CLASSROOM
end
>> print S.CLASSROOM
A
>> print P.CLASSROOM
B
>> S.assignMyClassTo(P)
>> print P.CLASSROOM
A
```

Implementation of copy constructor

```
Student(Student NEW)
                NAME = NEW.NAME
                AGE = NEW.AGE
                CLASSROOM = NEW.CLASSROOM
end
>> Student S = Student(“Veronika”, 23, “A”)
>> Student T = Student(S)   … a copy constructor call
>> print T.NAME
```

Veronika

>> T.changeName(“Adam”)

>> print T.NAME

Adam

>> print S.NAME

Veronika

3.9 Inheritance

Instead of creating every new class from the beginning, a new class (child class) can be based on another existing class (parent class) and use its already defined attributes and methods. It’s called the inheritance. Every class except of the Object class, which is at the root of the hierarchy, inherits from exactly one class. The child class can be created by deriving from the existing class by adding the keyword “is a kind of” with the name of the parent class in its declaration. Otherwise, its parent is the Object class, but it does not have to be specified. Attributes and methods of the parent class can be used and called as if they were members of the child class. If the name of a method in the child class is the same as in the parent class, the parent method is overridden (see Polymorphism below). If the name of an attribute in the child class is the same as in the parent class, the parent attribute is shadowed. In both cases the methods of the child class can use keyword parent with the usual dot notation to access the parent field or method. The constructor of the child class must contain a call of the parent constructor somewhere in its body otherwise the program will not be compiled. It is not allowed to derive new classes from built-in classes (e.g. String), except of the Object class. One of the strong benefits of the inheritance is that a variable with a type A, which is some class can hold values of type B, if B is a direct or indirect child class of class A. More about this is described in the section Polymorphism.

3.9.1 Example of Inheritance

```
>> class Person
        String FIRSTNAME
        String LASTNAME

        Person(String NAME, String SURNAME)
                FIRSTNAME = NAME
                LASTNAME = SURNAME
```

```
    end


    String fullName()
        return FIRSTNAME + " " + LASTNAME
    end
>> class Student is a kind of Person
    Number ID

    Student(String FIRSTNAME, String LASTNAME, Number STUDENTID)
        Person(FIRSTNAME, LASTNAME)
        ID = STUDENTID
    end

    String getStudent()
        return fullName() + ", " + ID
    end
```

3.10 The Object Class

Every class inherits some methods from the Object class, which is an ultimate superclass of all objects. It does not have to be declared using the "is a kind of" keyword. If no class is specified as the parent class, then Object is the parent class. It contains several built-in methods that are useful in different situations:

- Object(Object OTHER) is a copy constructor that performs a deep copy of all fields. It returns a new object of the same type as passed in the argument OTHER and all its fields will contain identical values - deeply copied, i.e. they will not share anything.
- String toString() evaluates the object, converts it to a string, it is used to obtain the string representation of the object to be printed. All built-in classes override this method to provide a reasonable string interpretation of its value.
- Boolean equals(Object O) makes a deep comparison of the object with another object.
- Boolean is(Object O) compares the reference of this object with the reference of the specified object O.

3.11 Interface

The language doesn't allow multiple inheritance, but it provides a lightweight alternative: interfaces. An interface describes a collection of abstract methods. A class that implements an interface must define all methods that are declared in that interface, but it is free to also implement any other methods. Each class can inherit from only one class, but it can implement any number of interfaces. A difference between implementing an interface and inheriting from a class is that the interface specifies what the implementing class should know to do, but does not contain any attributes. We do not even allow default implementations of methods in the interfaces. Two classes may define two different ways of doing the same thing if they implement the same interface. The interface has a simple syntax, it is similar to the class definition. The keyword "interface" is followed by the name of the interface and the list of method headers without their bodies. The interface doesn't contain any constructor and thus it cannot be instantiated. When a class wants to implement some interface, the keyword "looks like a" with the name of the interface is used in its declaration. To declare a class that implements more than one interface, separate the names of interfaces by commas. Interfaces can have parent interfaces using the same mechanism as used for class inheritance: using the keyword "is a kind of" with the name of the parent interface. Every interface can stand as a type of a variable. In that case, the variable may contain an object of any class that implements that interface. If it is declared without initialization, it contains an empty instance of class Object. Calling interface-defined methods on such an object results in a warning message printed into the text console.

3.11.1 Example of Interface

```
>> interface Figure
        String move()
        String collect()
>> class Player looks like a Figure
        String NAME

        Player(String N)
                NAME = N
                end
```

```
Void changeName(String NEW)
        NAME = NEW
end


String move()
        return NAME + " is moving"
end


String collect()
        return NAME + " collect the diamond"
end
```

3.12 Polymorphism

If several classes implement the same interface or inherit from the same parent class, it means that they are similar in some way. They must define the same methods declared by the interface. In case of the class inheritance, they may override some method of the parent class each with its specific implementation. The two methods with the same name follow the same logic, have the same name. Their return value and the arguments have the same type. They have different implementations, use different internal representations, etc. When calling such methods, we don't need to know the class of the object exactly: we can use a variable with the type of the parent class or the implemented interface. Calling a method through such a variable makes the system recognize the true class of the object stored in the variable. The proper method is called, depending on the class of the stored object. This is the concept of polymorphism. In our language we can use polymorphism by implementing the interface or deriving a new subclass. Polymorphism can save a lot of time and work of the programmer. Assigning a value V stored in a variable declared as type A into a variable of type B is allowed even if the type B is a direct or indirect subclass of class A[2]. However, the programmer takes the responsibility that the real type of V is the type B or its direct or indirect subclass. If it is not, a warning message is printed on the text console. When attempting to call a non-existent method on that object later, again a

[2] Unlike Java and other languages, no explicit type-casting is required, it is implicit.

warning message is printed on the text console. An example below explains the use of polymorphism with an already defined class Player and interface Figure in section Interface.

3.12.1 Example of Polymorphism

```
>> class Monster looks like a Figure
        String NAME

        Monster(String N)
                NAME = N
        end
        String move()
                return NAME + " made a step"
        end
        String collect()
                return NAME + " has stolen the diamond"
        end

>> List FIGURES = [Player("player1"), Monster("monster1"), Monster("monster2")]
>> FIGURES[0].move()
"player1 is moving"
>> FIGURES[2].move()
"monster2 made a step"
```

3.13 Error Handling

To make programs that the programmer writes behave properly when encountering an unexpected problem, he or she can use error handling. For example if someone forgets to pass a correct argument into a method, a statement error "description of error" can be used. It makes the interpreter to stop, interrupt the execution, print the description and location of the error and give the user a possibility to examine or modify the current state of execution with immediate commands. To resume the execution with the next statement, the programmer can use the statement resume.

3.13.1 Example of Error Handling

```
>> String lastLetter(String WORD)
        if WORD.length > 0
                return WORD.get(WORD.LENGTH - 1)
        else
                error “Cannot return the last letter of empty String”
        end
end
>> lastLetter(“ ”)
Cannot return the last letter of empty String
```

3.14 Execution Model

The applications programming in our language will be very event-based. It means that the code is based on events. For instance, when the user clicks on a button, some event is triggered. To develop an application in our language the programmer will use many functions that are called after some interaction of users with GUI, functions that are called when some system service completes, or functions that are called every X milliseconds. These functions are callback functions, and they are executed after some event happens. As the events occur in the system, they enter a queue, and they are processed one by one. Therefore, the event callback functions, which in fact constitute the whole program should return quickly. Javascript language use a mechanism with asynchronous callback functions. In Javascript almost all I/O is non-blocking and this applies to our language too. Our language could be implemented using the execution model of Javascript, therefore it will inherit most of its properties - concurrency model (12) including its event loop and message queue.

3.15 Execution Environment

For creating a multi-user web applications the language uses communication between server and client. The programmer does not need to care about implementation of this communication, it happens automatically inside the system. The programmer writes one program running on all computational nodes using one consistent language. The methods of various objects call each other, and when they are located at different computational nodes, the communication is handled by the execution environment

automatically. In principle, the programmer does not even know where the object on which a method is called is currently located. Of course, a careful, and more advanced programmer will know, and he or she will design the code in order to minimize the required communication.

Every application written in our language includes at least one class called Startup, and one class called User. When the programmer creates a new empty project, the structure of code for these two classes is already prearranged. The automatically created code for each new application looks as follows:

```
class Startup
List USERS
System SYSTEM

Startup()
end

Void userConnected(User U)
end

Void userDisconnected(User U)
end
```

```
class User
Startup APPLICATION
System SYSTEM

User()
end
```

When the code of application starts, at first an object of class Startup is created automatically. The Startup object is localized on the server side. Then for every user who opens the application via a browser a new User object is created on the client side.

Both of these classes include a constructor. The programmer fills the constructors with an initialization code - to initialize all the object attributes, to perform all other required initializations, and to set up event callbacks. The line "Startup APPLICATION" in class User allows the User objects to call methods of Startup object. The variable APPLICATION is automatically initialized with a remote reference to the Startup object that lives on the server. Reversely, the line "List USERS" in class Startup is an object variable USERS containing list of remote references to all existing user objects. Its contents are maintained automatically. Both classes contain "System SYSTEM", an automatically maintained variable SYSTEM of the built-in class System. It provides the interface to all the built-in functionality of the system. For instance, obtaining the id of the

user, a reference to its graphical user interface on the client side, the user name, it allows working with the file system on the server side etc.

When a new user connects to the application, the system ensures that the reference to a constructed user object is added into the list USERS and the method *userConnected(User U)* is called. Reversely, when some user leaves the application, his reference is removed from list USERS and the method *userDisconnected(User U)* is called. These methods are called from inside the system, not by the programmer. It is up to the programmer to program these methods. After creating objects - Startup and User, the application runs, but it has no thread of execution. It relies on events being triggered by the user interaction, system events occurring, or timer. The respective part of the code runs at the same time on both sides. Each side manages its objects. All classes are known to both the server and client sides, and thus both the server and the client can create objects of any class defined in the program. Often, information, i.e. an object has to travel from client to server or in the reverse direction. All objects are normally sent as a (remote) reference[3]. Calling a method on a remotely referenced object is handled by the interpreter. It uses the socket.io library to send an internal request for a call of the required method to another node. After the method completes, the return value (typically another remote reference) is sent back. The programmer does not care about this internal communication, he or she can simply assume that method call works the same regardless of the type of reference, the system will know what to do. If the programmer wishes that a local copy of remotely referenced object be created, he or she may use a copy constructor. The new instance returned by the copy constructor will reside on the same side where the copy constructor was called. This can result in a more efficient code.

3.16 Built-in Classes

Our language offers build-in classes: Number, Boolean, String, List, Dictionary, Json, Image, System, Math and Object. These classes provide useful methods to create and access numbers, booleans, lists, dictionaries, jsons, images. The class System provides system methods and cannot be instantiated by the user. Parent class of all classes (not only built-in) is the Object class. It provides common methods that all objects implement. All

[3] Remote references work like usual references. Internally, they are represented as a pair of values (x,y), where x is the id of the computational node, where the object was instantiated, and y is an id of the remotely referenced object. The system keeps a list of all remotely-referenced objects to allow access. Computational nodes are numbered from 0 - the server, and then the clients in the order the users connected to the application.

methods of the mentioned classes are explained in document Webi Language Specification API.

3.17 Importing Existing Code and Using Multiple Source Files

The import keyword at the very beginning of the program source code can be used to import libraries written by others or by the programmer himself or herself. All built-in classes are available as part of the language. The keyword import is followed by a string specifying the name of file that contains class definitions that are to be included with the program.

3.18 Graphical User Interface

In this section, options how to create a basic GUI are explained. For programmers to build own GUI the language offers to use graphical classes. All classes are explained below and all useful methods that this classes includes are explained in document Webi Language Specification API.

3.18.1 Class Page

The page represents the fundamental space for all graphic components. The project can contain more than one page where the content is changing. An absolute Cartesian coordinate system is using on the page. The start position [0,0] is situated in the upper left corner of the screen. The position X increases from left to right and the position Y increases from top to bottom. All coordinate, size values used on the page are specified in pixels and negative values are not allowed. The color can be specified with its name or hex value. On the page the visual elements can be placed, i.e. buttons, tables, texts, navigations and playground for drawing various shapes. The class Page offers useful methods to style and manipulate with window.

When the programmer used the method *setFont* in the code, the each element (that includes using text) created after and until end of code or until the next call of this method will apply specified font.

To place visual elements into the page, the programmer will use the method *createElement.* The our language offers the following elements : *button, checkbox, input, link, navigation, select, table, image, video, text* and *textarea*. The programmer does not need to care about constructing of elements, this method automatically creates and places it on the page.

3.18.1.1 Class Page Example

>> Page MAIN = Page(“Game”)

>> MAIN.bgColor(“blue”)

>> Element B = MAIN.createElement(“BUTTON”)

The class Page includes also events where some method will be executed after some event occurs, i.e. after the user clicks, presses a key on the page or resizes the page. To set these events is used method *“addEvent”*. The events have also properties that can be useful for programmers. The programmer can get the value of property, if inside the body of called method use **event**.*nameOfProperty.*

3.18.1.2 Add Event Example

>> Page MAIN = Page(“Game”)

>> MAIN.addEvent(“click”, “wasClicked”)

3.18.2 Class Element

The class Element represents visual elements that can be placed on the page. How to create the elements was explained in section class Page. The each element has visual properties *background color, visibility*, *position* and *space.*The class Element offers also events and their properties to work with elements like in class Page. The each element has also own properties that can be set up.

The method *“setPos”* sets the position of the element, but it doesn’t mean that if this method is not used, the element will not be placed on the page. The method *“createElement”* of class Page automatically placed each element, in order how were created.

3.18.2.1 Class Element Example

>> Page MAIN = Page(“Game”)

>> Element T = MAIN.createElement(“TEXTAREA”)

>> T.setVisible(false)

3.18.3 Class Playground

The class Playground represents a rectangular area for drawing graphical shapes. The playground must be placed on the page. One page can contain more than one playground. The playground can be set anywhere on the page, it has own position and size.

The playground uses the same coordinate system like is used on the page. This class offers useful methods to set style of playground and for drawing shapes. The class Playground offers also events and their properties like class Element.

The various shapes can be drawn inside the playground. The class Playground offers methods to create basic shapes, but the programmer can combine this methods and create own shapes. The methods startPath() and closePath() are used to create closed path and specify where the path start and where end. The method drawLine() is used to draw this path. When the programmer want to make the path filled, he or she use method fillPath() and put this method before method closePath(). In the example 2.18.3.1 are used this methods to draws filled triangle into the playground. When the programmer use method *drawText()* to draw text into the playground and want change the style of font he or she use method of class Page *setFont().*

3.18.3.1 Class Playground Example

```
>> Playground GROUND = Playground(50,50,250,150)
>> GROUND.startPath()
>> GROUND.drawLine(100,70,150,70)
>> GROUND.drawLine(150,70,125,20)
>> GROUND.fillPath("blue")
>> GROUND.endPath()
```

3.18.4 Class Figure

The class Figure offers simply way how to create figures like players, monsters, cars, diamonds etc. The figure is represented by image and the figure can be placed at specified position inside the playground. The class Figure also includes useful methods.

3.18.4.1 Class Figure Example

```
>> Figure PLAYER = Figure("hero.png",150,100,)
>> PLAYER.move("left",10)
```

4 Design of the Webi Language

4.1 Outline

In following sections we will describe the process of compiling, interpreting and executing of the source code. The compiling takes places only at the server, but interpreting and execution process is identical on the server and the client sides. The only exception is that the server computational node can have a direct remote reference to any other computational node (clients). On the contrary, clients that refer to objects that are instantiated on different client computational nodes must refer to them through the server internally, see Figure 6. The remote references will be handled by socket.io communication and will be described in detail in section 4.6. Figure 7 outlines the overall architecture of the compilation, interpretation and execution process, while Figure 8 show process of compilation and Figure 9.W show process of interpretation and execution.

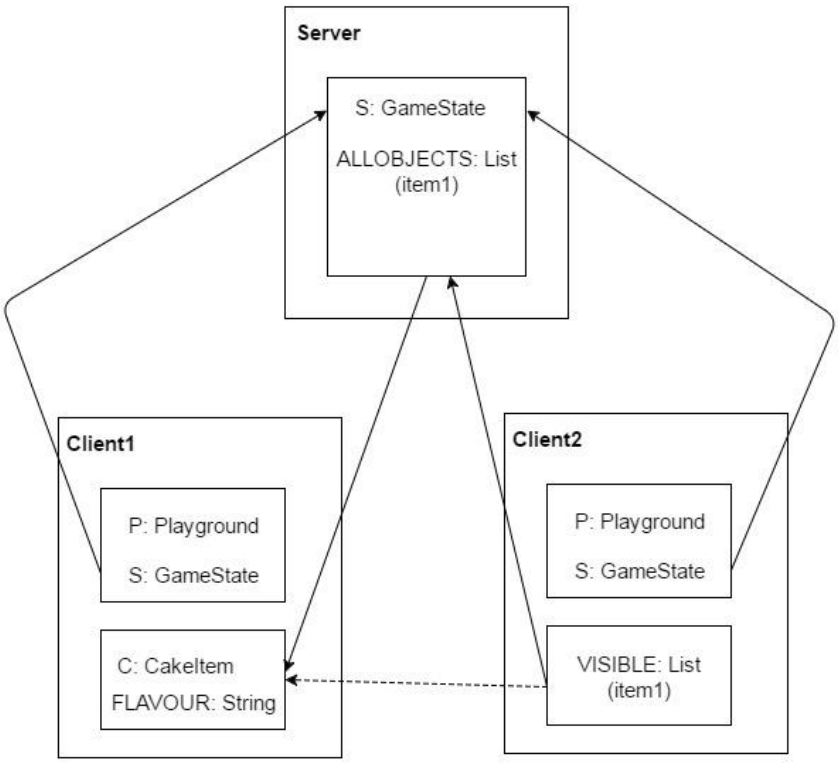

Figure 6: Remote references. In this example the object S of class GameState was instantiated on the server. Therefore there is only one copy of this object, located in the memory of the server computational node. It is referred from two objects of class Playground, both stored in some object variable P, each in one of the computational client nodes. The Client1 instantiated object C of class CakeItem, when the user at Client1 baked a new cake. The cake is in the visibility of user playing at Client2 and therefore there is a remote reference to this object at the computational

node Client2. At the same time the GameState refers to all the objects created in the game. The server computational node can contain a direct remote reference whereas the computational node Client2 must utilize indirect remote reference through the server.

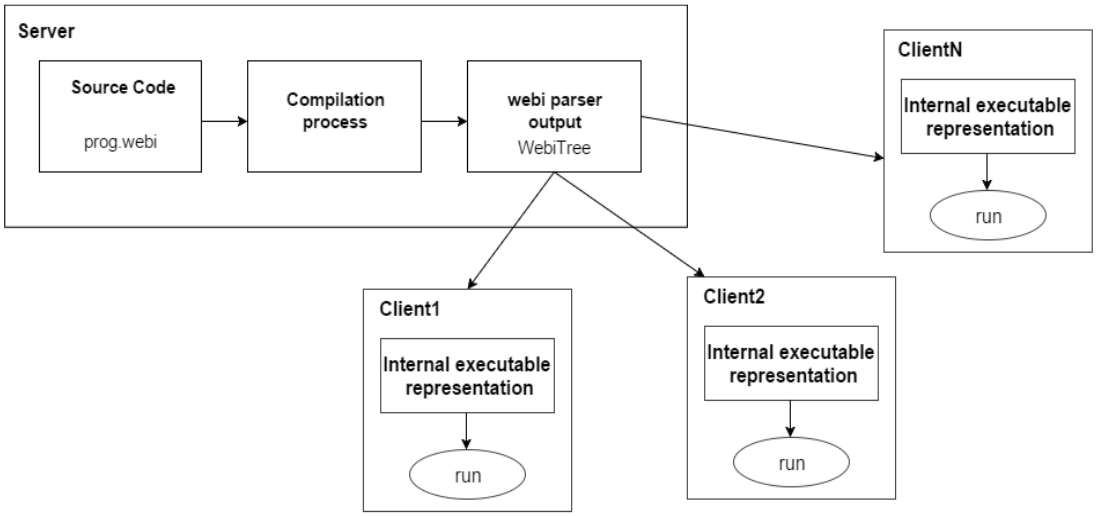

Figure 7: Overall architecture. This figure represents the overall architecture of the compilation, interpretation and execution process. The compilation process run on the server side, where from the source code (prog.webi) during the compilation is created webi parser output *WebiTree.* The server sends the *WebiTree* output to each client (Client1, Client2, ClientN, etc.) Then, on the side of each client is built internal executable representation, which after building starts run.

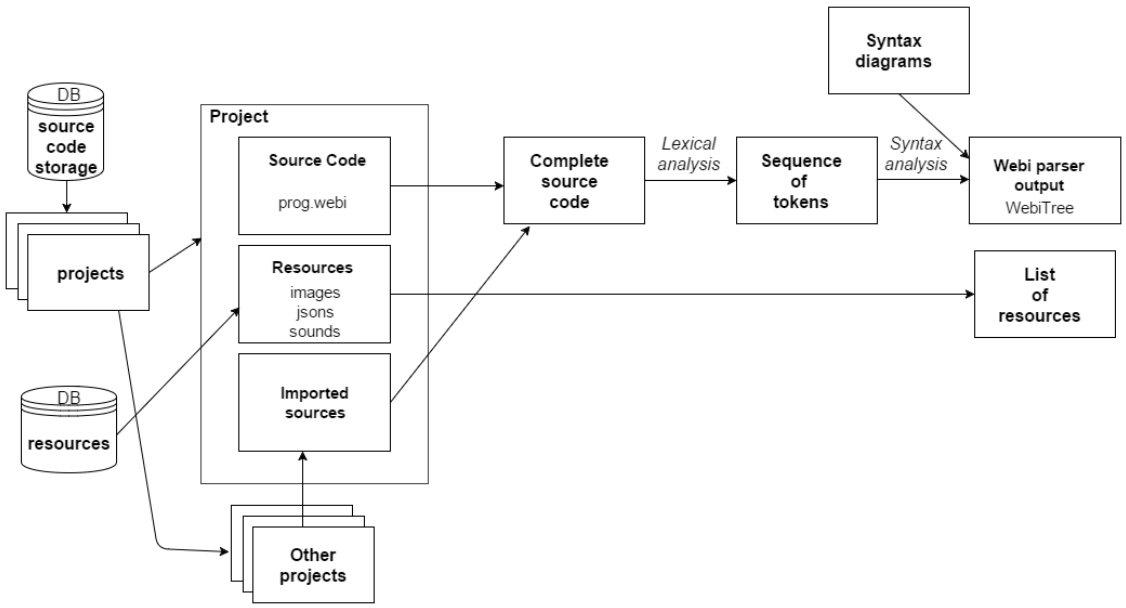

Figure 8: Process of compilation. The project in our language consists of source code (prog.webi), resources (images, jsons, sounds) and imported sources (other projects that can be

imported), that are loaded from database. When we have the complete source code the process of compilation start with lexical analysis which helps to syntax analysis break up the source program into tokens. Then the syntax analysis with using of syntax diagrams constructs webi parser output - WebiTree. In the end of compilation is created also list of resources.

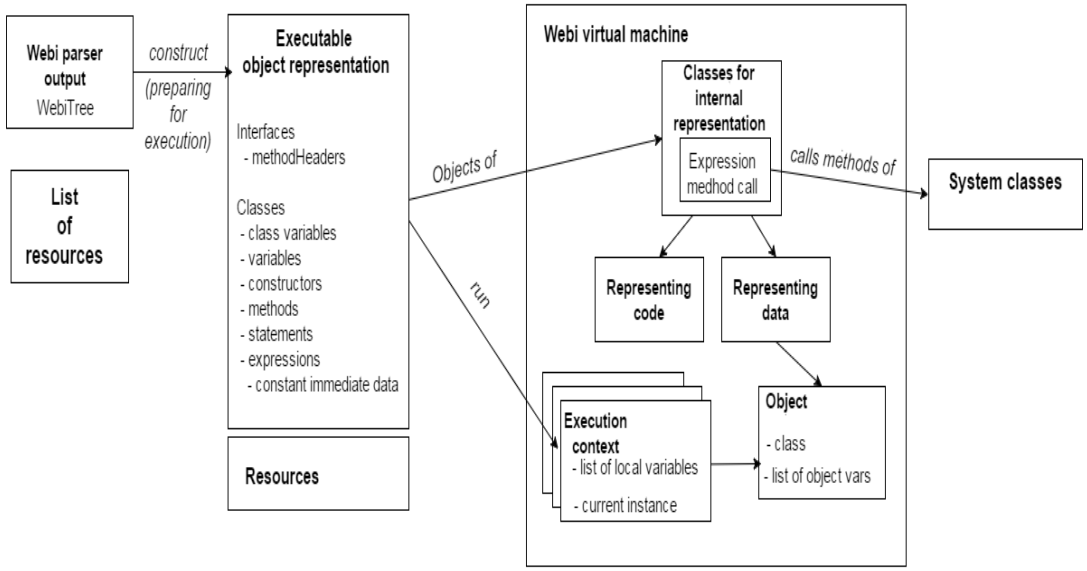

Figure 9: Process of interpretation and execution.

4.2 Webi Lexical Analysis

Our Webi lexical analyzer (scanner) breaks up the source program into sequences of characters - tokens. The scanner is represented by the method *scan()* which uses additional methods *next(), isNumber(), isLetter().* The method *next()* has no parameter and no return value. In each call, the method takes one character of the source program, that the global variable *input* holds and assigns it to the variable *character*. When all input characters get consumed, the ending character "\0" is assigned to the variable *character.* Then the method *scan()* will know that the whole input was processed. The methods *isNumber()* and *isLetter()* check if the character is number or letter. The method *scan()* also has no parameter and no return value. It puts together characters and generates a sequence of tokens - numbers, words (keywords, variables), symbols (operators, brackets), that are assigned to the variable *token* and skips spaces, newlines and comments. The method *scan()* should recognize also the more complex tokens. Our syntax of the language includes following more complex tokens: += , -= , *= , /= , %=, <=, >=, ==, !=, &&, ||, ++,

--, numbers with floating point, strings (sequence of characters surrounded by “” or ‘’). The method also sets the position of the current token to the variable *position*. It may be used during reporting of an error message.

4.3 Webi Syntax Analysis

Our Webi syntax analyzer (parser) has several responsibilities. The parser takes tokens and tries to recognize and check the program constructions with help of syntax diagrams that are explained in section 4.3.1 and method *next()* which was explained in section above. When the parser finds that the program is syntactically wrong, it then reports an error message with a number of line where the problem has occurred. This can help the programmer to easily find and correct syntax errors in a program. During the syntax analysis, the parser is continuously producing *webi parser output* that is explained in section 4.3.2. The parser is represented by the method *scan()*. This method has no parameters and it returns *true* if the program is syntactically correct or *false* if it is not. In the latter case the parser also prints an error message into an output window. Otherwise, if the returned value is *true*, then a complete *webi parser output* is created and followed by an interpretation process of constructing an *internal executable representation* that is explained in section 4.4.

4.3.1 Specification of Webi Syntax Diagrams

Webi syntax diagrams graphically represent the syntax of our language and help the parser recognize and check the program constructions. We have chosen a common syntax diagram formalism with a few modifications and restrictions to make the implementation easier. The restrictions are as follows: 1) when the diagram is branching, each branch must contain only a single node and 2) branching must be followed by a terminal node of that diagram. It can be easily seen that these restrictions do not change the expressive strength, they only require to rearrange the drawing in this particular manner. The following figures depict all webi syntax diagrams.

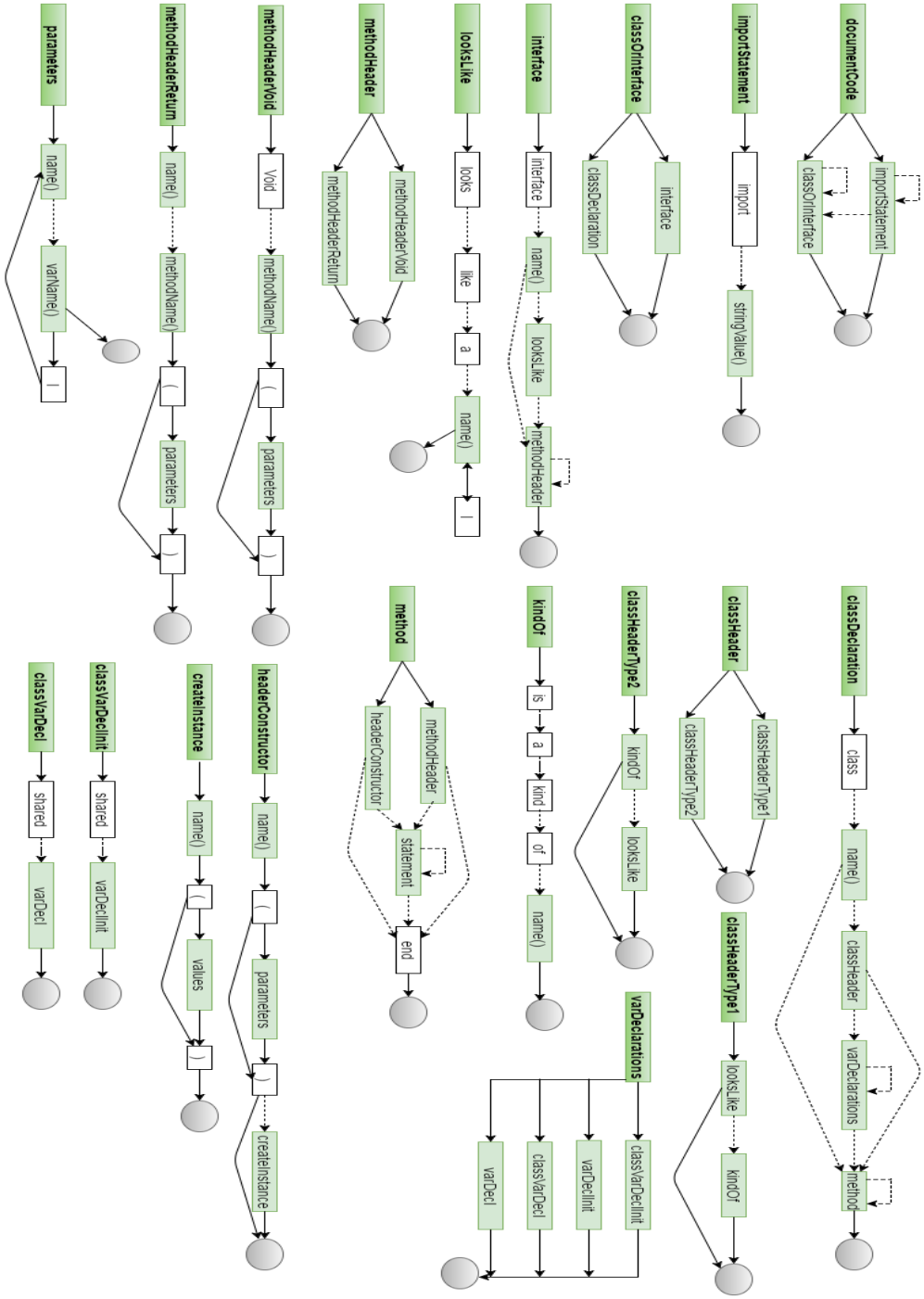

Figure 10: Syntax diagrams 1

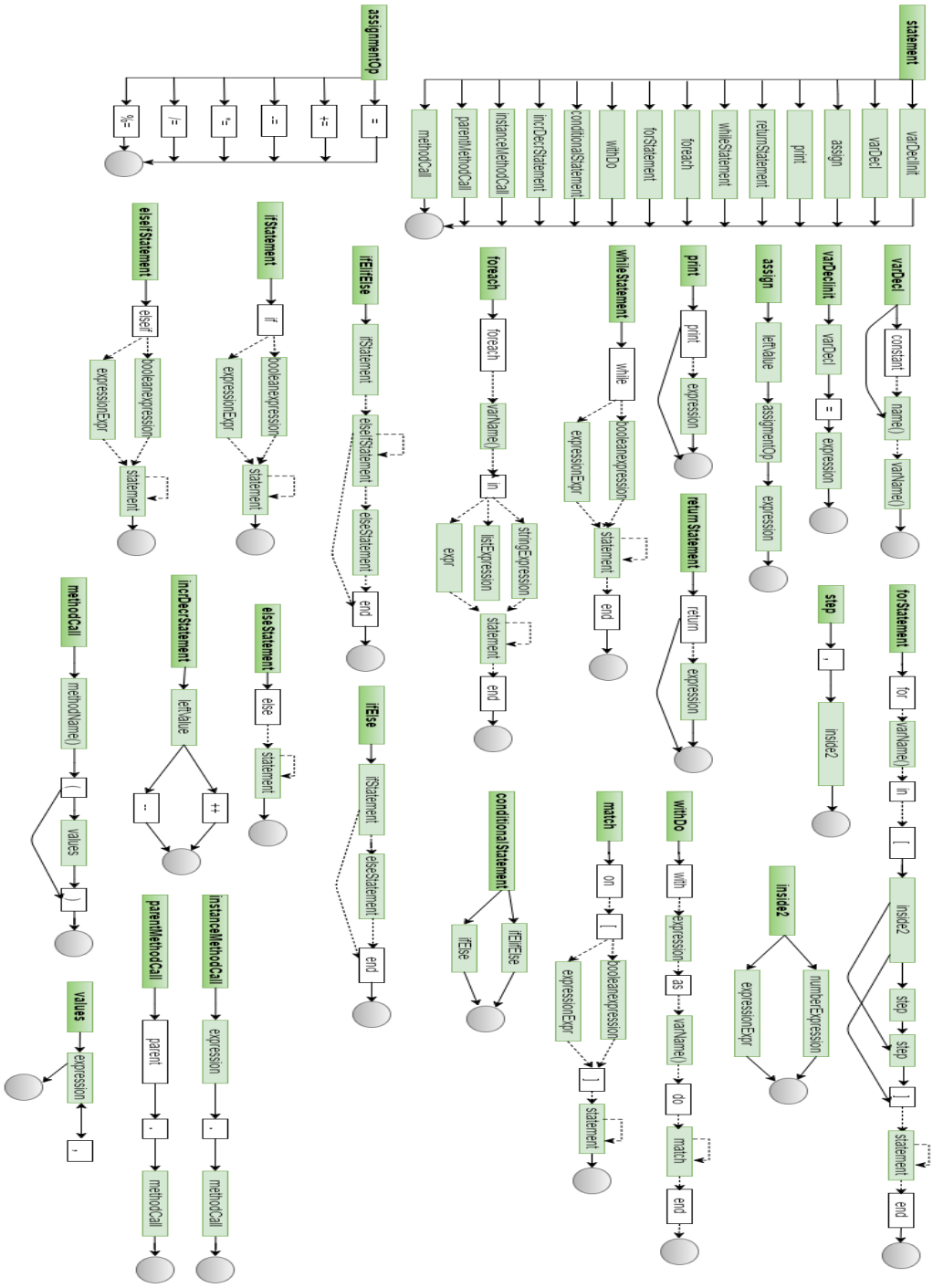

Figure 11: Syntax diagrams 2

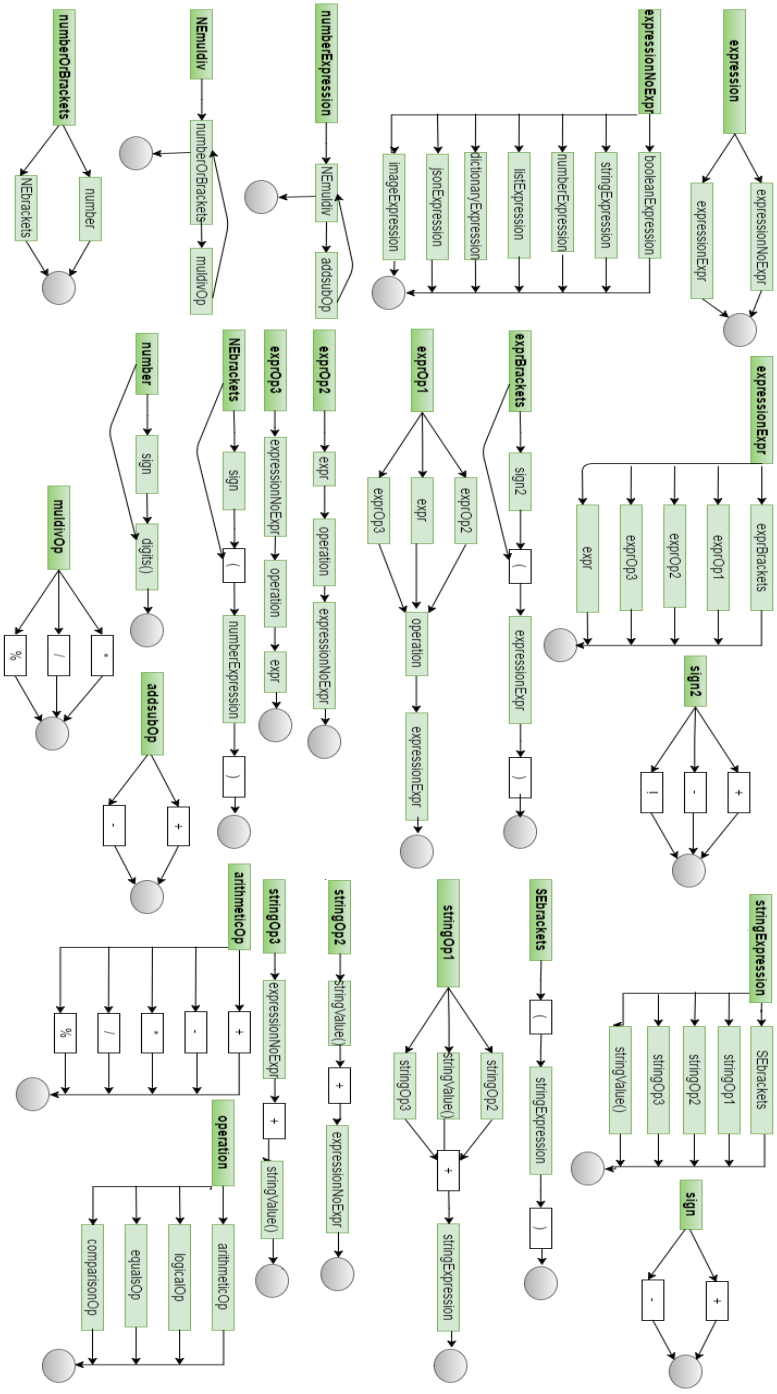

Figure 12: Syntax diagrams 3

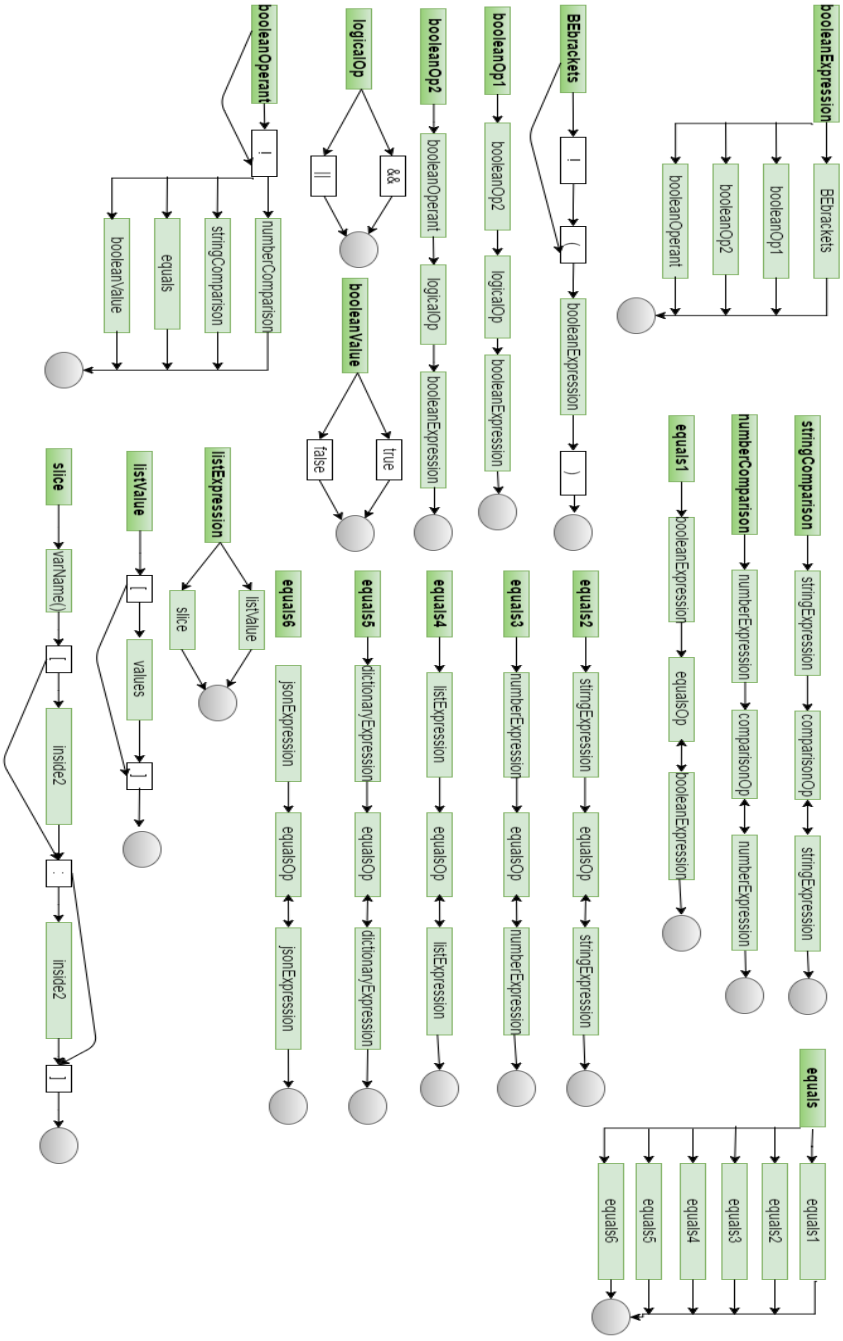

Figure 13: Syntax diagrams 4

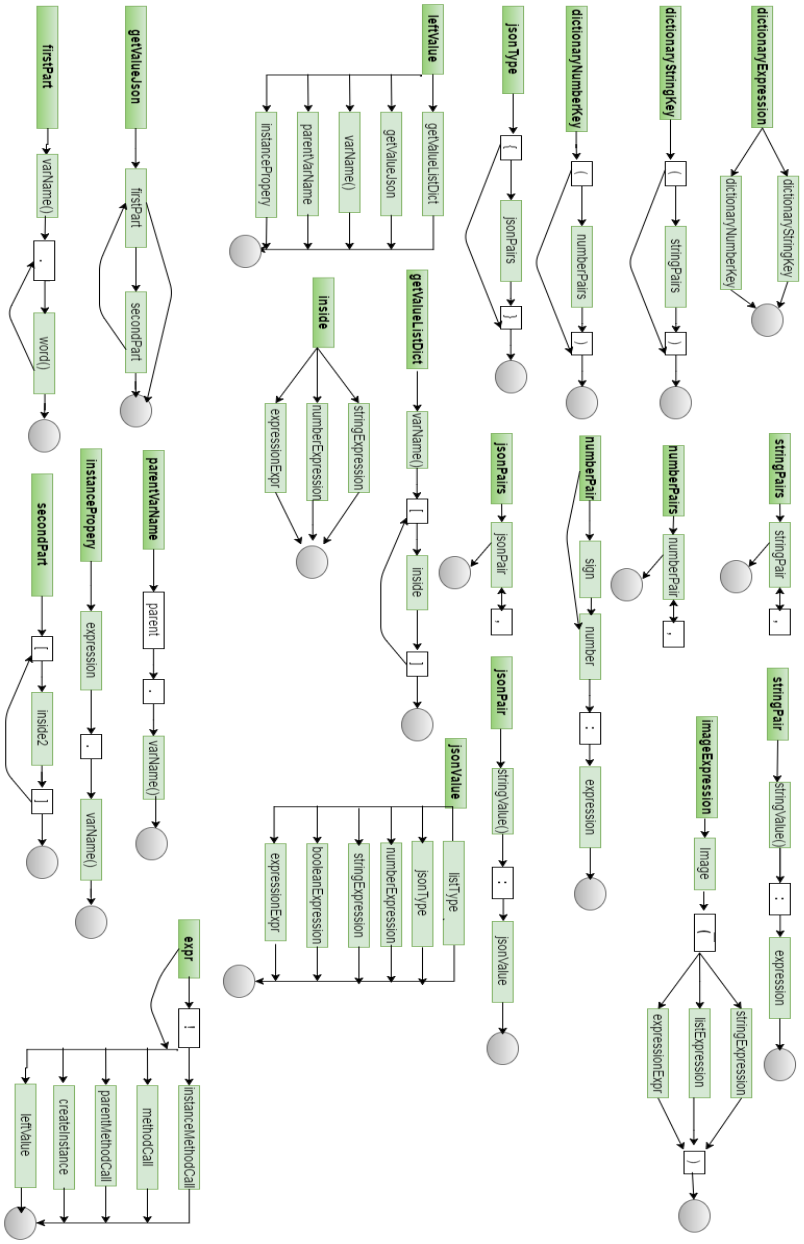

Figure 14: Syntax diagrams 5

Each diagram contains several nodes connected with directed edges shown as arrows. The first node labeled with the name of the diagram (dark green rectangles) is the starting point and the last node (gray circles) is the final point. The other nodes are labeled by terminals - elementary symbols of language (white rectangles) and nodes labeled by nonterminals (light green rectangles), that are names of sub diagrams or functions where the recognition of the program jump and then return back. From some nodes there is more than one possibility where to move and some nodes can also be skipped. The arrows are solid or dotted. The solid arrow specifies that two tokens can be separated with whitespace, while dotted arrow specifies that two tokens must be separated by whitespace. The arrows with both directions represent cycles.

Syntax diagrams are represented in a simple format that is both easily human-readable and computer-readable in a separate single file. This allows for easy modification of the language without making any other changes in the parser code. To make the processing more handy to implement, diagrams are written directly as JavaScript objects: each syntax diagram is represented by an object. It is assigned to a variable with the name of the first node of that diagram. Each object has a property *NAME* (string) and *RESULT* (empty array) that are used to generate webi parser output that is explained in section 3.3.2. Others properties are specified for every object based on nodes that the specific syntax diagram has. Possible properties (with examples) are the following:

- *TRYCHECK:* try to check if the token is equal to a terminal that is specified in the property as string (node *constant* in diagram *varDecl*),
- *CHECK*: checks if the token is equal to one of the terminals that are specified in the property as an array of strings (node *looks* in diagram *looksLike*),
- *TRYJUMP:* try to jump to the diagram that is specified in the property (node *parameters* in diagram *methodHeaderVoid*),
- *TRYJUMPREPEAT:* try to jump to diagram that is specified in the property and while the jump is successful, repeat the process, (node *statement* in diagram *forEach*),
- *JUMPFUNCTION:* jumps to a function that is specified in the property as string (node *name()* in diagram *parameters*),
- *JUMP*: jumps into diagrams or functions that are specified in the property as an array of diagrams/functions (node *methodOrConstructor* in diagram *classDeclaration*),

- *CYCLE:* go through items of array that is specified in the property from first to last item and if it is successful then repeat the process, if item in array is a string do what is in property *CHECK*, if item is name of diagram/function do what is in *JUMP* (nodes *name(), ","* in diagram *looksLike*)

Each object also has the property *END* with an array of terminals or diagrams that direct to the last node in specific syntax diagram.

The property *JUMPFUNCTION* can specify one of the following functions that require the next token to be of the particular type:

- *name()*
- *methodName()*
- *varName()*
- *word()*
- *stringValue()*
- *digits()*

As an example, *classDeclaration* diagram would be represented as the following object:

var *getValueListDict* = {
NAME: "getValueListDict",
END: ["]"],
JUMPFUNCTION: "varName()",
CHECK: ["["],
JUMP: [inside],
CHECK: ["]"],
CYCLE: ["[", inside, "]"],
RESULT: []
}

The algorithm that recognizes and checks the whole source program begins with the first token and starts with the *documentCode* diagram. It then iterates through its properties (nodes) and performs instructions which current property specify i.e. checks if the token is equal to one of the specified terminals, tries to jump to another diagram, etc. If the property is *JUMP* to another object (diagram) then the algorithm starts to iterate through its properties and when all properties of the diagram are successfully matched, or any terminal specified in the array *END* is reached, the algorithm returns back and continues with parsing the previous object. If the current property is *JUMP* with an array that has more than one item (representing nondeterministic branching) then the algorithm starts at the

first item and tries to match it. If matching is unsuccessful, it then follows with the second item etc. When the algorithm iterates through object that has property *TRYCHECK*, *TRYJUMP* or *TRYJUMPREPEAT* and they do not match, then the algorithm skips to the next property. The algorithm ends and returns a value true when the object *documentCode* was successfully matched. If the whole process of trying to recognize the source code does not finish in end of object *documentCode* and the algorithm tried all possibilities then the algorithm ends, returns a value false and prints an error message. This algorithm is represented with the recursive method *syntaxDiagram(diagram)* that has a parameter *diagram* on which the process of iterating starts. This method uses the method *scan()* which sets a next token to the variable *token* when the previous token is already recognized by the algorithm.

4.3.2 Specification of Webi Parser Output - WebiTree

The algorithm that recognizes and checks the whole source program generates webi parser output called *WebiTree*. During the process when the algorithm iterates through the object's properties, the visited tokens and diagrams are pushed into its *RESULT* (initialized to an empty array at the beginning), but only if instructions specified in the current property matched successfully. Recursive visits to the same diagram are handled by a stack on a separate level in the RESULT field. This applies for each object that is iterated during the recognition of specific source code. If the iteration of specific object does not finish in its end then its *RESULT* that was continuously generated must be removed. Finally, when the algorithm ends and returns true, then the *WebiTree* includes object that represents diagram *documentCode* on which the algorithm started and its *RESULT* includes tokens and diagrams. These diagrams also have their *RESULT* that includes tokens and diagrams etc. Consider the following source code snippet:

interface Princess looks like a Figure

Void move()

Boolean collect(String THING, Number COUNT)

Printing the *WebiTree* output after the algorithm ends would get the following (notice that the diagram names are printed instead of diagram objects):

documentCode

classOrInterface

interfaceDeclaration

Interface Princess looks like a Figure

methodHeaders

methodHeaderVoid

Void move()

methodHeaders

methodHeaderReturn

Boolean collect(

parameters

String THING, Number COUNT

)

or when the source program looks like:

Void count()

print 2(-4)*

end

we would get:

method

methodHeaderVoid

Void count()

statement

printStatement

print

numberExpression

NEmuldiv

numberOrBrackets

number

2

muldivOp

****

numberOrBrackets

NEbrackets

(

numberExpression

NEmuldiv

numberOrBrackets

number

addsubOp

```
                    -
                  4
               )
          end
```

4.3.3 Operator Priority and Left-to-Right Evaluation Order

In our language all expressions are evaluated from left to right, if the operators have the same precedence. Otherwise, an operator with the higher precedence is evaluated first. When we have an expression *5+4*3* and the *WebiTree* output is generated based on diagrams, the resulting structure would suggest that the expression would be evaluated correctly: *4*3* first and then *5+12*. But the expression *6/3*4* after generating the *WebiTree* output would be evaluated incorrect: first *3*4* and then *6/12*. To solve this problem we first flip over the tokens in the expression *6/3*4* to yield *4*6/3*. The program then evaluates the flipped expression while getting the arguments in the opposite order. This applies to all expressions in our language. It means that during the processing of expressions that are in the source code, expressions will be flipped over first to circumvent the right-to-left evaluation order implied by the right-to-left nonterminal nesting in the syntax diagrams. The right-to-left nesting in diagrams has to be used to avoid infinite loops. This is an elegant solution that leads to simplification of the syntax diagrams. However, flipping has to be done carefully as there are exceptions to it. Operands in expressions that specify calling methods, accessing object variables, listing items in argument list or elements in structured type will be better handled in the original order. We expand the language of syntax diagrams with labeling of starting and ending point where will be process of flip order applies and also about labeling of starting and ending point where will be protection against this process. The labels are added on the edges of the diagrams. A label *FLIP ON* is a starting point of flip order, and a label *FLIP OFF* is an ending point of flip order. A label *PROTECT ON* is a starting point of protection against flip order and a label *PROTECT OFF* is an ending point of protection against flip order. They can occur recursively, and only the last FLIP OFF results in the whole sequence of tokens from FLIP ON till FLIP OFF to be flipped and returned back to the input for further processing. Figure 15 shows an example of labeling.

Figure 15: Flip order process

During the syntax analysis can happen repeated/recursive activation/deactivation of flip order or protection against it. The process of flip order is following:

When the algorithm that recognizes and checks the whole source program, meets the label *FLIP ON* then the generation of the *WebiTree* output is paused and then all the recognized tokens are added into the stack and also all labels that are around them. Otherwise, when the algorithm meet the last label *FLIP OFF* all tokens from the stack are added in flipped order to the beginning of actual input. The order of removing tokens and labels from stack is the following: At the moment when on the top of stack is label *PROTECT OFF* continue the processing of stack in contrary order. It means that the processing of stack continues from the corresponding label *PROTECT ON* and follows in the reverse direction towards the top of stack. The whole section between the equivalent labels *PROTECT OFF* - *PROTECT ON* is removed only after PROTECT OFF has been reached. Similarly, when during the processing of stack in contrary order the label *FLIP ON* is encountered, the processing of stack continues in standard order of processing from corresponding label *FLIP OFF*. This process can be better understood in the following example. Consider the expression:

3 + f(4 + 2, 7 + g(2 + 3, 1)) - 2

after it has been processed by a part of the syntax diagram between *FLIP ON* and *FLIP OFF*, the stack where all the tokens were stored looks like this:

2
-
)front
)
)stack
)front
)
1
,
)stack

3
+
2
stack(
(
g
front(
+
7
stack(
,
)stack
2
+
4
stack(
f(
front(
+
3

where "**stack(**" represents *FLIP ON*, "**)stack**" represents *FLIP OFF* and "**front(**" represents *PROTECT ON*, "**)front**" represents *PROTECT OFF*. Notice the number 2 appears at the top of the stack as it was read as the last token from the input. Finally, the expression in flip order is: *2 - f(2 + 4, g(3 + 2, 1) + 7) +3.* If it will be processed with right-to left priority and swapped arguments, the intended behavior is achieved.

4.4 Internal Executable Representation

Internal executable representation is constructed from *WebiTree* output before the execution of the program. To construct this representation we use classes for internal representation and their *construct(WebiTree input)* factory method. Each class is explained separately in following sections. Each of the classes contains a method *run()* or *eval()* - depending if it is an executable artefact (such as program, method, statement) or a representation of an artefact that is to be evaluated to some value.

4.4.1 Class Program

The class *Program* represents a complete program and it contains the following fields:

- **classStartup:** an object of class *ClassDecl*

- **classUser:** an object of class *ClassDecl*
- **classes**: a dictionary with all classes, where a key is name of class and a value is object of class *ClassDecl*
- **interfaces:** a dictionary with all interfaces: a key is name of interface and a value is object of class *InterfaceDecl*

4.4.2 Class ClassDecl

The class *ClassDecl* represents a complete definition of a single class and it contains the following fields:

- **name:** a string that store name of class
- **superClass:** an object of class *ClassDecl*, or none, if the superclass is *Object*
- **implementedInterfaces:** an array of objects of class *InterfaceDecl*
- **classVarDecl:** a dictionary of class variable names and their types
- **classVarDeclInit:** a dictionary mapping class variable names to their initial values
- **varDecl:** a dictionary of object variable names and their types
- **varDeclInit:** a dictionary mapping object variable names to their initial values
- **constructors:** an array of objects of class *ConstructorDecl*
- **methods:** a dictionary with all methods, where a key is name of method and a value is object of class *MethodDecl*

4.4.3 Class InterfaceDecl

The class *InterfaceDecl* represents a complete definition of an interface and it contains the following fields:

- **name:** a string that store name of interface
- **implementedInterfaces:** an array of objects of class *InterfaceDecl*
- **methodHeaders:** an array of objects of class *MethodDecl* with empty property statements.

4.4.4 Class ConstructorDecl

The class *ConstructorDecl* represents a complete definition of an constructor and it contains the following fields:

- **superClassConstructor:** an object of class *ConstructorDecl*
- **parameters:** a dictionary with all parameters, where a key is name of parameter and a value is type of parameter

- **statements:** an array of objects of class *StatementCall*

4.4.5 Class MethodDecl

The class *MethodDecl* represents a method declaration or definition, it contains the following fields:

- **name:** a string that store name of method
- **parameters:** a dictionary with all parameters, where a key is name of parameter and a value is type of parameter
- **statements:** an array of objects of class *StatementCall*
- **returnType:** a string that store type of method, if it is a method that returns a value

4.4.6 Class StatementCall

The class *StatementCall* is an abstract class whose subclasses represent all types of statements. Its construct() method delegates construction of the respective statement type to one of its subclasses.

4.4.7 Classes PrintCall, ReturnCall

The classes *PrintCall*, *ReturnCall* represent a type of statement, one to print the evaluated value of the specified expression to text console, the other one to return it from the currently executed method. It contains a single field:

- **expr:** an instance of class *expressionFormula*

For all other types of statements there exists a specific class that is a subclass a statementCall. For brevity we only show the example above.

4.4.8 Class ExpressionFormula

The class *ExpressionFormula* is an abstract class whose subclasses represent all types of expressions. Its construct() method delegates construction of the respective expression type to one of its subclasses.

4.4.9 Class NumberExpressionFormula

The class Number*ExpressionFormula* represents a binary operations and their arguments. It contains the following fields:

- **sign:** unary "-" sign in front of formula (represented as "+1" or "-1")
- **arg1:** an instance of class *NumberConstant*
- **arg2:** an instance of class *NumberConstant*

- **operator:** one of the following signs: “+”,”-”,”*”,”/”,”%”

For all other types of statements there exists a specific class that is a subclass a statementCall. For brevity we only show the example above.

4.5 On Mutability of Strings

In C language the issue of immutable strings may be confusing to a novice programmer. For instance:

*int *a = “abc”;*

a[0] = ‘x’; // leads to runtime error, because the string “abc” is stored in read-only memory

On the other hand

int a[] = “abc”;

A[0] = ‘x’; // correctly modifies the first character of string

The way Java language responds to this problem is making strings immutable. That however leads to extremely inefficient code when using the ‘+’ concatenation operator, or other ways of modifying strings, forcing the user to use StringBuffer or StringBuilder, which must be then converted to String.

Our language brings an elegant and efficient solution:

Internal representation has an attribute *constant* that has value *true* for every string that is in code before execution and *false* for every string that is created during the execution. At the moment when the constant string is modified the new non-constant copy is created automatically. This is achieved thanks to the fact that during the evaluation of statement or expression, the reference to an the L-value is keep as reference to L-value as long as it is possible. Thus when a method receives a reference, it may modify it - in this case to a new non-constant copy of the constant string. It is the same for types as list, dictionary and json.

4.6 Remote References Handling

One of the most important features of the Webi language is that the programmer writes program for a distributed execution environment, but he or she does not have to think about where the code runs and where the data is stored. Even though, he or she is allowed to in order to write more efficient code. In practical terms this means that objects can be instantiated at any computational node and methods of their classes can call methods on objects that were instantiated at different computational nodes. These remote

calls must be handled by Webi automatically. Webi virtual machine knows about each object reference whether it refers to a local or remote object. It does so by using a separate instance of its own Reference class that contains a label “local” or “remote” and the reference itself. Local references are implemented as usual JavaScript references to the object of class ObjectInstance holding the object variables of that object. Remote references contain an integer identifier of the remote reference stored in the local computational node’s *Remote references list table*, see Figure 16.

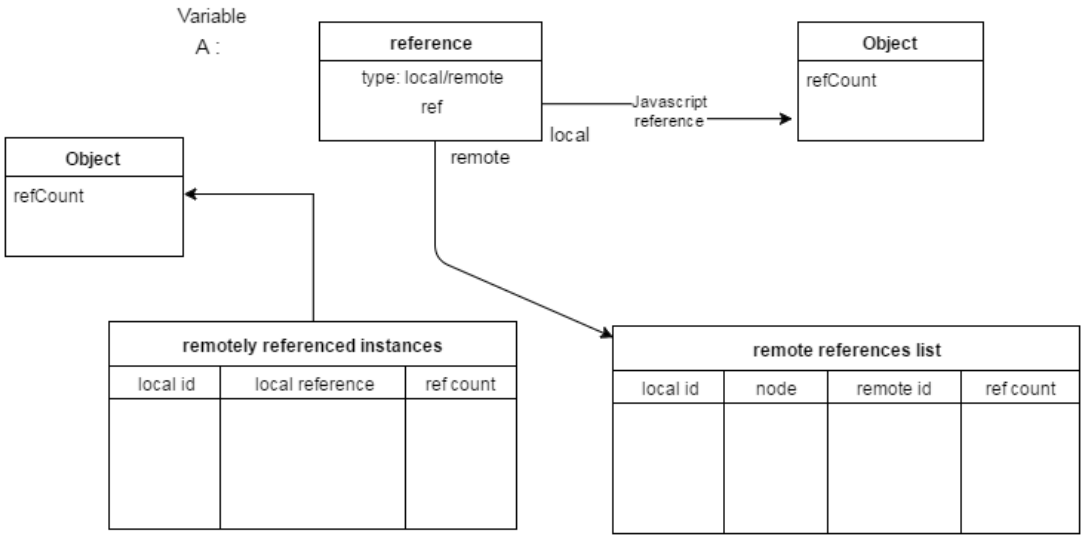

Figure 16: Remote reference management

The Remote references list table contains the computational node number (server: 0, clients: 1..N), and the reference id in the remote node’s *Remotely referenced instances table*. It also contains the local reference count of this remote reference, see the following section for details. Each time a remote instance is referenced, the Webi virtual machine retrieves the record from the *Remote references list table*, and asks the corresponding computational node (using the underlying communication protocol based on socket.io) to access that instance: either call its method or access its object variable. The computational node that receives the instance access request looks into its Remotely referenced instances table to retrieve the real local reference in order to call its requested method or read/write the requested field. The reference count stored in this table contains the number of computational nodes that use a remote reference to this instance.

To achieve the functionality specified above, each client keeps an open websocket connection to the server and sends and receives packets that describe the access request or a related response. In addition to the following basic request types, the nodes need to exchange a bookkeeping information about their entries in the remote reference tables:

- calling a method of a remote instance
- returning a value (either Number, Boolean - these are always passed as values, since they cannot be modified, or an Object reference)
- writing to object variable
- reading object variable
- broadcasting a write to a class variable

4.6.1 Remote Eccess communication protocol

Each packet sent between the nodes has the following format: Command, Data length and Data

Here we list the possible commands:

- Release reference, data: reference id (int)

Sent by a node that used to have a reference to a remote instance and its local reference count fell to 0. It informs the owner that he does not need this remote reference anymore.

- Method call, data: call id (int), reference id (int), method name (string), method arguments (references or Number/Boolean values)

If the node already has a remote reference to a remote instance, it can call its methods

- Returning a value, data: call id (int), returned Number/Boolean or Object reference

The call id is the same as specified in the method call packet - it should be unique to the calling computational node.

- Writing object variable, data: reference id (int), variable name (string), Number/Boolean/Object reference value

- Reading object variable, data: read id (int), reference id (int), variable name (string),

- Returning object variable value, data: read id (int), Number/Boolean/Object reference

The read id is the same as specified in the reading object variable packet.

- Broadcasting class variable name, data: class name (string), variable name (string), Boolean/Number/Object reference.

Each time a class variable is written, this event is broadcasted to all computational nodes in order to avoid inconsistences - all nodes share the same classes, which must have only a single instance of each class variable.
In all cases - when sending an Object reference - this is a pair (node, reference id) - it is possible that a node receives a remote reference to its own instance, in that case it converts it locally to its local reference.

4.7 Dynamic memory management

In our language the user does not have to deallocate instances explicitly, automatic deallocation takes place. Instead of using background garbage collection process, each instance contains reference count when this reference count is false to zero. Instance is release from memory and all its member variables will have the reference count degrees and recursively released from memory if no more references exist. Since we are using underlying javascript garbage collector this reference counting would not be necessary if we had no remote references. However we have no way of knowing in advance wherever some variable is local or remote reference. Without reference counting remove references would never had been deallocated. The possible solution would be use finalizer which is feature for instance Java programing language, but Javascript has no finalizer and therefore we are forced to use reference counting in all reference types both local and remove.

5 Design of the Webi Integrated Development Environment

Figure 17 shows the design of the Webi integrated development environment. The main window consists of:

- menu
- one or two webi editor windows
- output messages

The menu consists of submenu and four buttons - run project, stop project, split mode and logout. When the user runs the project with some graphical user interface then the new window is opened and the project starts running. Otherwise only some output is printed into the output messages window. The button split mode toggles between a single editor window and two column mode with two webi editor windows. Then the user can see two view modes at once. For instance, the left window will show one part of the program and the right window will show another part of the same program, or the left window will show the list of project resources, and the right window will show the source code. The submenu consist of buttons - new project, open existing project, save, save as and exit. The webi editor window consists of view modes: Program, Help, API, Tutorial and Explorer.

The mode Program is the window for editing the source code, where the user can also see the line numbers, for easier navigation and finding lines with errors. The mode Help includes syntax rules and examples of code written in webi language - it is a sort of programmer's guide. The mode API includes list of built-in classes, classes for creating a graphical user interfaces and their useful methods - a soft of programmer's reference. The mode Tutorial includes set of lessons how to create a simple game, it can help the user to start programming in Webi language. The mode explorer includes a project description, list of classes, interfaces, sublists their methods and list of resources - images, jsons, sounds and list of exportable sources. The output messages is window where the user can see the output of program or error messages. To develop Webi IDE we will use framework Bootstrap that was explained in section 2.3.7 and code editor Ace (13) that offers for example syntax highlighting, an optional command line, line wrapping, Search and replace with regular expressions, and more.

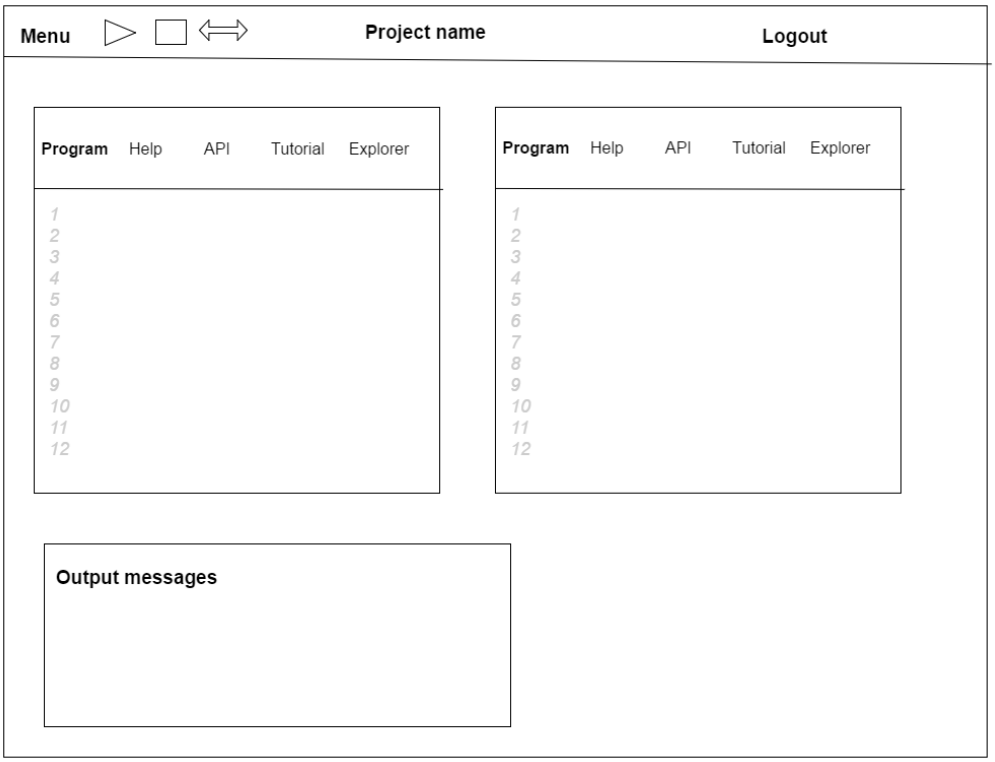

Figure 17: Webi IDE

6 Implementation

This section describes some details about implementation of all realized parts of Webi language and IDE. Lexical analyzer is located in the program scanner.js. The work is performed by the method *scan()*, which uses methods *next(), isLetter(), isNumber().* These methods were explained in section 4.2. The file diagram.js contains the specification of the language syntax diagrams in form that were explained in section 4.3.1. Modifying this file is sufficient to make changes in the Webi syntax. Our syntax analyzer uses these diagrams for implementation of algorithm that recognizes and checks the whole source program (4.3.2) that is represented with method *Syntaxdiagram(diagram)* in parser.js. This file also includes methods that represent various types of tokens used in the syntax diagram through the *JUMPFUNCTION* keyword and the method *scan()*, which starts the whole process of syntax analysis. An array *diagrams* includes *WebiTree* output that can be printed into the console using method *printWebiTree().* The file syntaxTrees.js includes classes for internal representation and method *startBuild()* that constructs internal executable representation that is stored in variable *prg* and *prg.run("client")* starts process of execution. The file interpreter.html includes code of the basic IDE and method *start()* that is invoked when the user enters his or her source code of the program and pressed the button run. This method reads the source code and calls the methods *next(), scan(), parse()* and *startBuild().*

7 Conclusions and Further Directions

The main goal of our thesis was to design and develop a new programming language for creating a multi-user web applications with graphical user interfaces that run simultaneously on both a server and user clients. The programming language that would be text-based, clear, simple, easily-understandable and universal for all server and client components. The next goal was to design and create integrated development environment and also create examples of applications written in our language.

At first we analysed a suitable technologies and similar systems for programming applications. Then we designed a new programming language Webi and wrote a complete specification with syntax, rules and examples of code written in this language. The language is pure object-oriented (everything is an Object), statically typed, and compiled. It does not require semicolons as statement separators or terminators and allows an arbitrary use of whitespace. The specification also includes API of the built-in classes and API for creating graphical user interfaces. We designed the architecture of the whole system. Then we started develop Webi programming language. We programed a lexical parser, a syntax parser (based on syntax diagrams that we designed and created) and a builder of an internal executable representation that allow to run simple programs and simplified IDE. We consider our goal that was the proof of concept to be fulfilled. The rest would exceed the scope of a single diploma thesis and remains for further development. The process of compilation is inspired by the compilers theory, but we have introduced our own methods, formalisms, and algoritms that might be of some interest to the computers languages community.

Instead of producing an extension to one of the existing languages, or a compiler that would translate the Webi code into a source code of one of the available languages, we chose a more difficult, but in our belief a more correct way: to design a completely independent language with its own syntax, its own semantics, compiler, and interpreter / virtual machine. The code does not compile to a binary machine code, not even to byte codes of some existing virtual machine. Instead, it compiles into our own internal representation that is constructed from the output of syntax parser when the program is started. This internal representation is formed of instances of the internal representation classes, which are the same for all user programs and form a part of the Webi virtual machine. Thus, the loaded program can then be run without the need of being interpreted.

It runs more efficiently than if it were interpreted either in form of a text source code or in form of byte codes.

The most unique contribution of this work is the unified single distributed execution environment. A single program runs on multiple computational nodes, objects are distributed based on where they are instantiated. They may contain references to objects that reside in the memory of another computational node. We designed a remote references manager for handling the list of remote references on both sides of the relation. This influenced also the way we perform the memory management: objects are freed based on reference counting and using Javascript garbage collection.

One of the possible extension of our IDE would be to add a debugger and editor for creating parts of applications using mouse, drag & drop and also add short tutorial to help create simple application in our language. The another extension of language would be to allow assignment to this keyword which would result in replacing the current instance with another instance. In effect of modifying the reference which was used to access the executed method.

Our software is open-source, the source code can be found on CD that is appendix of our thesis or on Github on url address: https://github.com/pohorencova2/webi. The appendix also includes an API of the built-in classes and for creating a graphical user interfaces.

8 Literature

1. **Aho, Alfred V., et al.** *Compilers Principles, Techniques, & Tools.* Boston : Greg Tobin, 2007. ISBN 0-321-48681-1.

2. **Mogensen, Torben Ægidius.** *Basics of Compiler Design.* Copenhagen : lulu.com, 2000. ISBN 978-7-993154-0-6.

3. **Blaho, A., et al.** *Imagining with Logo.* s.l. : Logotron, 2001. ISBN: 0582 439876.

4. **The Hybrid Group.** Home. *kids RUBY.* [Online] 2011. [Cited: May 3, 2017.] http://kidsruby.com/.

5. **Mozilla Developer Network and individual contributors.** About JavaScript. *MDN Mozilla Developer Network.* [Online] 2005. [Cited: May 3, 2017.] https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.

6. **Tutorials Point.** HTML5 - Web Workers. *Tutorials Point.* [Online] 2017. [Cited: May 3, 2017.] https://www.tutorialspoint.com/html5/html5_web_workers.htm.

7. **Microsoft.** *TypeScript.* [Online] 2012. [Cited: May 3, 2017.] https://www.typescriptlang.org/.

8. **Dart.** *Dart.* [Online] [Cited: May 3, 2017.] https://www.dartlang.org/.

9. **Young, Alex and Harter, Marc.** *Node.js in Practice.* Shelter Island : Manning Publications Co, 2015. ISBN 9781617290930.

10. **socket.io.** *socket.io.* [Online] [Cited: May 3, 2017.] https://socket.io/.

11. **Bootstrap.** *Bootstrap.* [Online] [Cited: May 3, 2017.] http://getbootstrap.com/.

12. **Carbon Five.** The JavaScript Event Loop: Explained. [Online] October 27, 2013. [Cited: May 3, 2017.] http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/.

13. **Ace**. *Ace.* [Online] [Cited: March 4, 2017.] https://ace.c9.io/.

9 Appendix

An appendix includes CD with the source code and API of the built-in classes and for creating a graphical user interfaces.