COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

PROGRAMMING LANGUAGE FOR CREATING INTERACTIVE WEB APPLICATIONS

Diploma thesis

Bratislava, 2018

Bc. Dominik Knechta

COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

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, 2018 **Bc. Dominik Knechta**

41943249

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Dominik Knechta
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: Programming Language for Creating Interactive Web Applications

Annotation: The aim of the thesis is to further develop Webi programming language as a proof of concept of a computer language with distributed computing environment on both client and server. The Webi language is a text-based educational programming language for developing multi-user web applications with support for graphics, and elementary data structures. Webi is easy to learn and its syntax is easy to understand. The main advances will include redesign of computational model based on asynchronous nature of Javascript execution, redesign of the internal representation and the compile/interpreter.

Supervisor:	Mgr. Pavel Petrovič, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	06.10.2017
Approved:	09.10.2017

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

....
Student

....
Supervisor

41943249

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Dominik Knechta
Študijný program:	aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	aplikovaná informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Programming Language for Creating Interactive Web Applications
Programovací jazyk na vytváranie interaktívnych webových aplikácií

Anotácia: Cieľom diplomovej práce je ďalší vývoj programovacieho jazyka Webi, ktorý je názorným príkladom programovacieho jazyka s distribuovaným výpočtovým prostredím na klientovi a serveri. Jazyk Webi je textový edukačný programovací jazyk na vývoj viacužívateľských webových aplikácií s podporou grafiky a jednoduchých dátových štruktúr. Webi sa dá ľahko naučiť a má zrozumiteľnú syntax. Hlavné prínosy budú zahŕňať prepracovanie výpočtového modelu založeného na asynchrónnej povahe výpočtov v Javascripte, prepracovanie internej reprezentácie a kompilátora/interpretra.

Vedúci:	Mgr. Pavel Petrovič, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.
Dátum zadania:	06.10.2017
Dátum schválenia:	09.10.2017

prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

....
študent

....
vedúci práce

Acknowledgement

I would like to thank to my supervisor Mgr. Pavel Petrovič, PhD. for valuable help and support during the whole process of writing this thesis.

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, 3.5.2018

Abstract

The main goal of this diploma thesis is the further development of Webi – programming language, which was created as a diploma thesis by another student. The Webi is language with distributed computing environment on both client and server. It is a text-based educational programming language for developing multi-user web applications with support for graphics, and elementary data structures. Webi is easy to learn and its syntax is easy to understand. While the first diploma thesis was basically proof of concept about this type of programming language, this work will analyze Webi in its current state and tries to find its benefits or drawbacks. After detailed analysis and understanding how Webi works now, we will continue in part two which is further development. The main part of the work will be redesigning of computational model based on asynchronous nature of Javascript execution. If the analysis shows that the current internal representation of Webi is insufficient or not optimal, we will continue with reworking this part. Otherwise we will proceed with developing more features and option for current representation. The same procedure will be applied for compiler/interpreter. The last part of the thesis will contain overall evaluation of my work and comparison of how Webi have worked then and how it is working now. The final thoughts will be about possible future development of Webi and what assets programming in this language can bring, compared to other existing solutions and languages.

Keywords: programming language, Web application, interactive application, children programming, multi-user, education application

Abstrakt

Hlavným cieľom tejto diplomovej práce bude ďalší vývoj programovacieho jazyka Webi, ktorého základ bol vytvorený ako diplomová práca iným študnetom. Webi je názorným príkladom programovacieho jazyka s distribuovaným výpočtovým prostredím na klientovi a serveri. Jazyk Webi je textový edukačný programovací jazyk na vývoj viacužívateľských webových aplikácií s podporou grafiky a jednoduchých dátových štruktúr. Webi sa dá ľahko naučiť a má zrozumiteľnú syntax. Zatiaľ čo v predošlej diplomovej práci išlo v podstate o prototyp takého typu programovacieho jazyka, táto práca zanalyzuje Webi v jeho súčasnej forme a pokúsi sa odhaliť súčasné výhody ako aj nedostatky. Po detailnej analýze a pochopení ako Webi pracuje teraz, budem pokračovať v druhej fáze – vývoj. Hlavná časť práce sa bude zameriavať na prepracovaní výpočtového modelu, ktorý je založený na asynchrónnej povahe výpočtov v Javascripte. Pokiaľ analýza ukáže, že súčasná interná reprezentácia jazyku Webi nie je dostatočná alebo optimálna, nasledujúci krok diplomovej práce bude návrh a implementácia novej reprezentácie. V opačnom prípade budeme pokračovať s ďalším vývojom a dorábaním novej funkcionality pre súčasnú reprezentáciu. Podobný postup bude použitý aj pre časť kompilátor/interpreter. Posledná časť diplomovej práce bude obsahovať celkové zhodnotenie mojej práce a porovnanie ako Webi pracovalo vtedy a ako Webi pracuje teraz. V úplnom závere sa budem venovať možnostiam o ďalšom vývoji jazyka Webi a taktiež jeho prínosom v oblasti programovania webových aplikácií v porovnaní s už existujúcimi riešeniami a jazykmi.

Kľúčové slová: programovací jazyk, webová aplikácia, interaktívna aplikácia, programovanie pre deti, multi-užívateľ, edukačná aplikácia

Table of Contents

1. Introduction

Nowadays, web applications are experiencing huge boom and success. Over a few years Web browsers started to serve not only for browsing webpages but almost as a standalone platform for applications – web applications. Although these apps cannot be considered as a standalone platform yet, they are still getting more and more popular. Web applications are usually designed to be interactive and should work with multiple-users at same time. Typical example of these apps can be some communication tool, online game or even social network. They are very popular because they are multiplatform and the only dependency you they need is a Web browser. However they have some drawbacks too.

The main drawback of web applications is its complexity. The number of technologies that are required to build and handle web application is too high. Although the majority of the work can be programmed in JavaScript, which is the most popular language for web applications due to its asynchronous nature, we still have to learn many technologies and principles. When programming web application technologies like node.js, socket.io are inevitable, so we need to learn them and understand how they work. The biggest difficulty comes in writing a different code for client and the server part.

To solve these problems, Webi – programming language was designed. The main purpose of Webi is the simplicity. It is designed for building a web application without necessity of knowing many technologies. Its main benefit is that you don't need to worry about setting up the entire environment for client and server, you just need to program the core of your web application. With this ideology, we can say that the Webi is mainly designed for children, who want to learn how to program web application but the required knowledge of all technologies is too high. However, if the Webi language will be fast enough, reliable and easy to learn it has potential to become popular programming language for everyone.

This diploma thesis will continue in further development of Webi, which was already designed for diploma thesis by another student. It will analyze current state of the language, tries to improve it and redesign the core parts to make it faster, more stable and bug free. The main goal is to improve this proof of concept into the real programming language for children, which helps them to learn how to build their first web application.

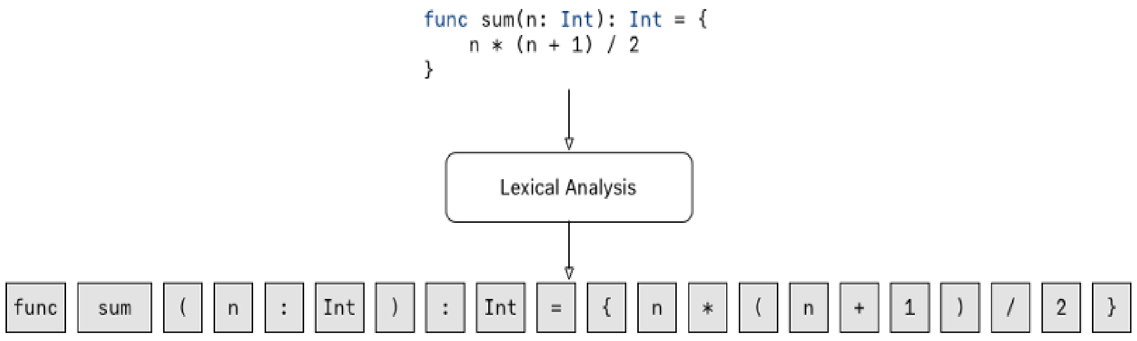
2. Theory and related technologies

In this chapter we will describe the whole process of building the language which will be very important in the next phase where we analyze how is Webi programmed now and what changes need to be done. We will also analyze some language recognition tools which might be helpful for our purpose.

2.1. Language building process

2.1.1. Lexing

In the first part lexer, also known as scanner or tokenizer scans the input (source code) and breaks it into meaningful elements called lexemes. From lexemes it then generate a sequence of tokens. The difference between lexeme and the tokens is that token is actually an object describing lexeme. Except the actual value of the lexeme, it also contains information like its type (keyword, identifier, operator…) and the position in the source code where it appears. The following figure 1 should clearly explain the process of lexer.



```
func sum(n: Int): Int = {
    n * (n + 1) / 2
}
```

```
func  sum  (  n  :  Int  )  :  Int  =  {  n  *  (  n  +  1  )  /  2  }
```

Figure 1 - process of lexer, (Compilers and interpreters) [1]

2.1.2. Parsing

During next step – syntax analysis (parsing), the program use the sequence of tokens generated by lexer and generate a tree-like data structure. This is can be a parser tree or an abstract syntax tree (AST). Both of them are very similar but they differ in the level of abstraction. While parse tree contains all the tokens which appeared in the program, the AST is reduced version of the parse tree where the information that could be derived or it's not important is removed (e.g. comments or grouping symbols). Syntax analysis is also the phase where eventual syntax error are detected and reported to the user. The following figure 2 now shows the whole process of generating AST.

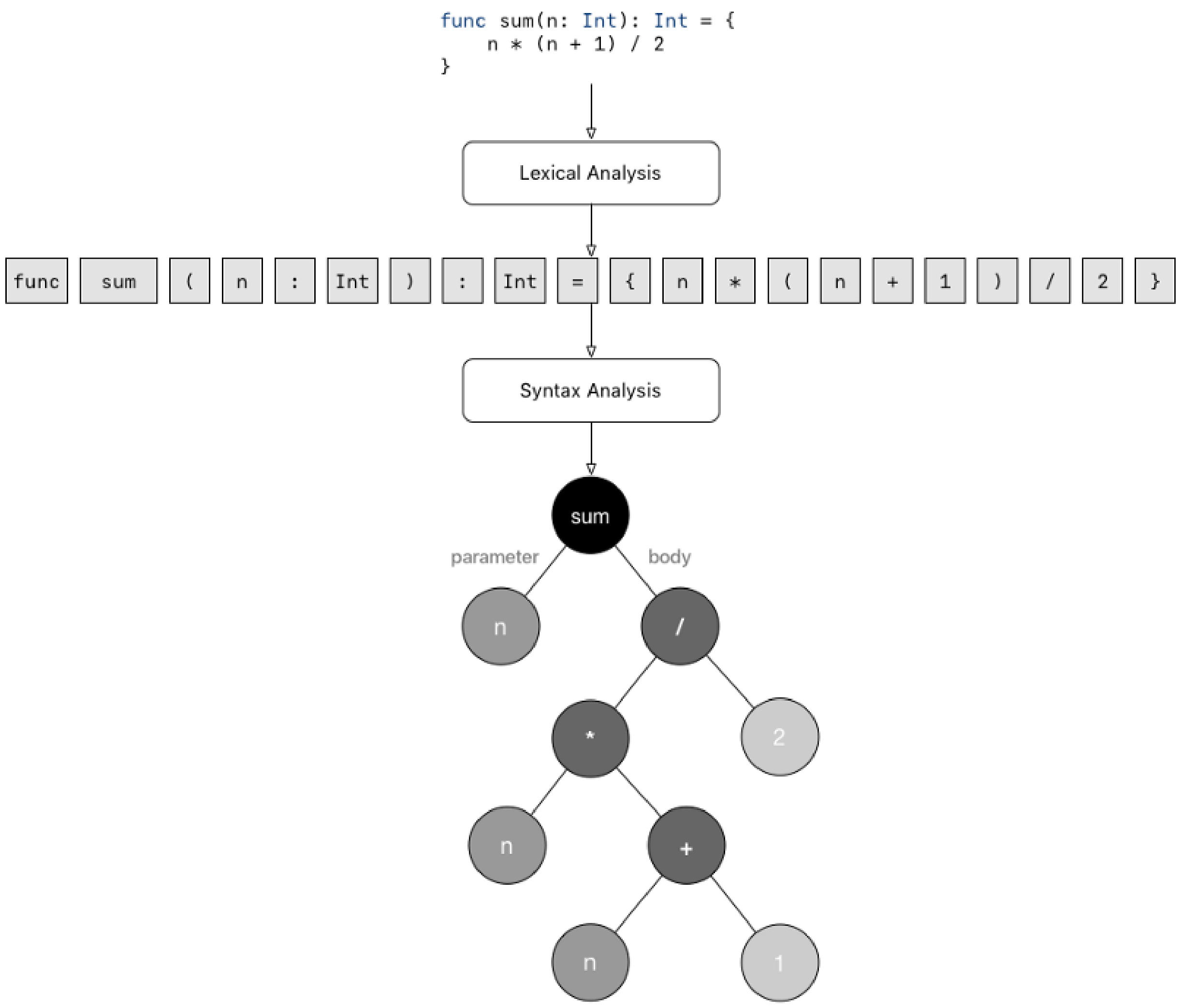


Figure 2 - abstract syntax tree generated after syntax analysis, (Compilers and interpreters)[1]

2.1.3. Semantic Analysis

During this phase, compiler or interpreter use generated tree to check if the source code is consistent with the rules defining our programming language. Semantic analysis may consist of these parts:

- **Type checking.** In this part compiler or interpreter checks all values assigned to variables and all arguments involved in the code have the correct type. For example it will check that no variable of type Number is assigned with a String value or that a value of type Boolean is not passed to a function accepting parameter String. It also check types in expressions, e.g. we cannot divide String with Number like “Hi” / 4.

- **Symbol management.** Alongside with performing type checking, the compiler or interpreter maintain a data structure called symbol table which contains information about all the symbols which appeared in the code. The process of symbol management is to determine answers to questions like: Is this variable declared before use? Are there two variables with same identifier in the same scope? Is this variable available in this scope? And many more.

The output of semantic analysis may be an annotated AST and the symbol table, which is shown on following figure 3.

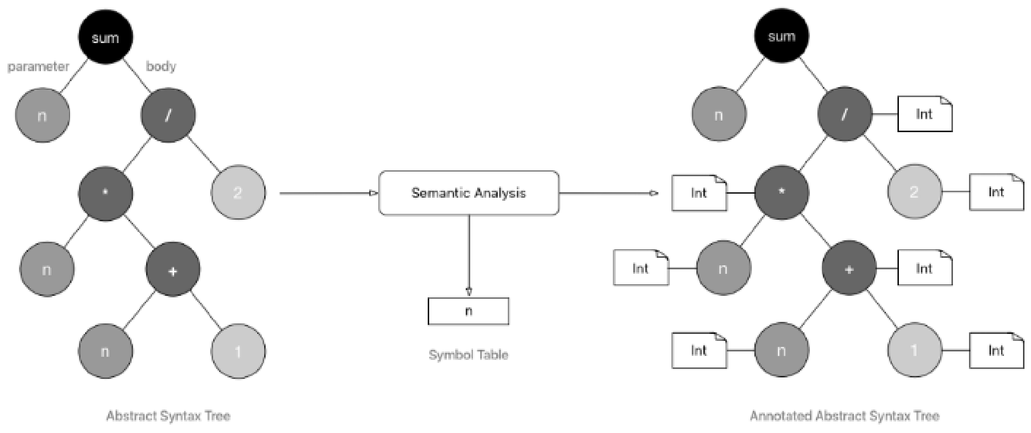


Figure 3 - Annotated AST, (Compilers and interpreters)[1]

2.1.4. Compiler vs interpreter

Interpreters and compilers are very similar in structure, but the main difference lies in the process of executing the instructions. While interpreter is directly executing the instructions in the source programming language, a compiler translate instructions into some form of efficient machine code (e.g. binary code). Interpreter typically generate an efficient representation, for example AST and immediately evaluate it. Consequentially, there is no need to generate annotated AST in interpreter.

2.1.5. Grammar

A grammar is a formal description of a language that can be used to recognize its structure. Grammars are usually major parts of language recognition tools, which are generating lexer and parser based on defined grammar. Simply grammar is set of rules that specify how each construct can be composed. For example an if rule must start with the keyword "if" then left parenthesis, expression, right parenthesis and then statement. A rule can reference other rules or token types. In parsing, the important feature is the support for *left-recursive rules*. This means that a rule could start with a reference to itself. The reference could also be indirect.

Taking the following example for arithmetic operations. We define addition as two expressions and symbol '+', but the expression can also contain other addition.

```
Addition ::= expression '+' expression
Expression ::= addition | number
```

This description will also match multiple additions like 1 + 2 + 3. That is because it can be interpreted as expression (1) ('+') expression (2+3). And then 2 + 3 can be divided in its two components.

The problem of these kind of rules is that they may not be supported by some parser tool generators.

2.2.Tools for building a parser and lexer

As mentioned before there are many tools which can help us in process of making a new language. In this part we analyze a few of them to find out if it can be used in our case. This solution could be more effective and potentially shorten development time. We had described the process how everything is done and now we briefly analyze three tools to find out which will be best for our needs.

2.2.1. ANTLR

ANTLR [2] (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It is written in Java but it can also generate parsers for other languages. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees. The most recent version of ANTLR is version 4.

- Fast and reliable
- It can generate parsers for many languages including JavaScript
- It has good testing tools for testing your grammar
- It is based on new algorithm: Adaptive LL parsing
- Support of left-recursive rules

2.2.2. JS/CC

JS/CC [3] is parser development system for JavaScript and ECMAScript-derivatives. It has been developed both with the intention of building a productive compiler development system. JS/CC uses bottom-up parsing. The main features:

- A regular expression-based lexical analyzer generator
- A LALR parser generator for a given context-free grammar
- It has been entirely written in JavaScript
- It can be run as platform-independent, browser-based JavaScript embedded on a website
- Or as a Node.js script.
- Comes with a graphical parse tree generator

2.2.3. nearley.js

Nearley [4] is the first JS parser to use the Earley algorithm. It also implements Joop Leo's optimizations to parse complex data structures easily. Nearly is really fast and powerful.

Main key points:

- powerful and expressive language
- use of macros, import from build-in library
- nearley processes left recursion without choking. In fact, nearley will parse anything you throw at it without complaining or going into infinite loop.
- It handles ambiguous grammars gracefully. Ambiguous grammars can be parsed in multiple ways: instead of getting confused, nearley gives you all the parsings (in a deterministic order!).

2.2.4. Conclusion

Although all of the mentioned generators have their pros and cons we decide to choose the **ANTLR**. The main reason is that it is the most popular tool, it has very good support and the main point is that the grammar can be parsed into many different languages. Therefore it is a huge advantage for future development of Webi, if someone would like to program it in different language he could reuse the already prepared grammar.

2.3. Other used technologies

- **JavaScript**: is high-level programing language in which we will be writing our language interpreter. We are using JavaScript standard ECMAScript5 [5] which is widely supported in all browsers.

- **Node.js** [6]**:** runtime environment for writing server side JavaScript. This will be used for our web-based Webi applications

- **Socket.io** [7]: most popular and powerful JavaScript framework that support bi-directional event-based and real time communication. It is supported in every browser or platform.

- **Browserify** [8]: it is a JavaScript bundler that lets you require modules in the browser by bundling up all the dependencies in one file.

3. Existing solution analysis

In this chapter we will analyze current version of Webi language and describe new terms and technologies which will be used during design and implementation phase.

3.1. Webi – existing solution analysis

As previously mentioned, the programming language Webi was designed and created as a subject of diploma thesis serving basically as a prototype intended for further development. Therefore some functions which were described and designed in the thesis are not implemented yet or contain some bugs. The goal of this thesis is to get the language into the state where user can build his own simple application or game in Webi.

3.2. Specification of the language

The main purpose of Webi is to have clear and understandable syntax which is easily memorable for the children. There are lot of complex programming languages like Java etc. which can help us program interactive content on web already. However these languages are not intended for kids. For the purpose of finding ideal conventions for Webi we took an inspiration based by master thesis from MIT [9] which was trying to design a language for kids for creating video games on Web. Although this source is relatively old it still follows similar principles and methodology we want to accomplish. We can say that the Webi syntax specification [10] accomplishes this goal well and only a few minor changes will be introduced which are essential for new implementation of the language. We will keep the almost same data types and arbitrary whitespaces, we will add some easy build-in functions and slightly redesign statements, condition and function declarations. The new design with complete changes will be described in chapter 4. Specification of Webi 2.0 syntax.

3.3. Lexical and syntax analysis

While the language specification of Webi was well designed, I see the main drawback in the compilation process. The process of compilation consist of lexical analysis (lexing), syntax analysis (parsing) and semantic analysis. These processes have been explained in the previous chapter. The previous thesis was trying to design its own lexer and parser however these parts, especially the parser turned out not to be effective. The parsing method with syntax diagrams is hard to follow up, it is prone to bugs and doesn’t exactly follow practices and principles for designing a language [11]. Although the first idea of this thesis was only to continue in development on existing Webi solution, we are going to completely rework this part from scratch. I will follow up more traditional and common ways for lexing and parsing, with the aim for easier understanding and better future development for new programmers evolving Webi. The aim is to divide the process in three parts. First we will define a new grammar based on language specification and after that we will move to lexer and parser. Defining a grammar will allow us simple modification of the language in the future without necessity of changing the other parts. Since our language is complex we will use a tool which will simplify our work. There are many tools on the market for designing and recognition of programming languages. In the previous chapter we have analyzed some of these tools and decided which will be best to fulfil our needs.

3.4.Execution model

There will be a significant difference between execution process of first and new Webi version. First Webi version is based on compiler design [12]. Webi version (Webi 1.0) was reading the whole syntax tree and compiling into compilation unit, new version (Webi 2.0) with the visitor parse tree run the commands immediately after reading a node in the tree. This will bring us some benefits like easier type control etc. Basically we can say that Webi 1 was a compiled language while Webi 2.0 is an interpreted language. The whole process of execution and visitor is described in next chapter.

3.5.Interface

The first version on Webi contained only simple graphic interface where you could write and execute the code and see the result in the browser. After other parts will be successfully implemented we would like to make more advanced interface with code syntax highlighting or some graphic components (e.g. buttons).

4. Webi 2.0 specification

In this section we are going to design new Webi syntax and describe whole process of building the language, starting by configuration of our chosen tool ANTLR, continued by setting grammar, building lexer and parser and finally doing semantic analysis so we will be able to execute Webi code. Although we want to keep Webi syntax same as in previous version, some minor changes will be inevitable. In this chapter we will briefly introduce whole Webi specification which is mainly focused on the changes compared to previous version.

4.1. Coding standards

Webi 2.0 is trying to keep standards defined in version 1.0. Whitespace (tabs, new lines, space) is still arbitrary. We still keep mind on removing semicolon terminator, which we consider unsuitable because it is often forgotten in other languages. We are not using semicolon as a terminator. The code still consist of statements or expressions which evaluated to a value. We still mind naming standards from Webi 1.0 – variable is all UPPERCASE, function is starting with lowercase letter and class starting with uppercase letter. However these standards are now only arbitrary and the code will run even when variable is declared as lowercase.

4.2. Comments

Webi 1.0 declared comments with '#' symbol for both start and end point. In Webi 2.0 we changed commenting for more traditional way, distinguished single line comments starting with two slashes "//", or multiline comments starting with symbol "/*" and ending with "*/".

4.3. Data types

Basic data types are remaining unchanged, consisting of **Number, String and Boolean**. Number can be both Integer and Float, user doesn't have to specify exact type. String is defined as a sequence of UTF-8 characters which is defined between double-quotes "Hello" or single quotes 'Hello '. If you want to use single quotes or double quotes inside the string value you can escape them with backslash " \"something\" ". Boolean has traditionally two values true or false.

There are bigger changes in build-in structured data types. While first version was contacting List, Json, Dictionary and Image we are merging types Json and Dictionary into one single type called Object. Reason for that was that these two types were very similar and there was no need to use both. Instead of having two types we will introduce some build-in methods which will allow us work with Object in different ways. Therefore we have **List**, **Image** and **Object** types. List is defined as an ordered sequence of values separated with comma. List values are initialized between square brackets []. List can be also empty. An object is a pair of key and value. Key has to be a string identifier. Value can be any datatype including List or other object. Therefore we can create Json like structure with just type Object. To work with Json there will be build-in method which will convert an Object to string Json like format. Objects can be empty as well. Image is a graphical type which contains information about image. Creating image is only possible with initialized Canvas and it will require three parameters: source, x coordinate and y coordinate. Additionally you will be able to set image with and height which is by default obtained from specified image. Compared to Webi 1.0, new version can also contain null value and every variable must be declared with a correct value when initialized, meaning something like "Number A" without value will not work.

4.3.1. Examples

List:

```
>> List ZOZNAM  = [2, 4, 6, 8]
>> println(ZOZNAM)
2,4,6,8

>> println(ZOZNAM[0])
2

>>println(size(ZOZNAM)-1)
8

>> ZOZNAM[2] = "string"
>>println(ZOZNAM)
2,4,"string",8
```

Object:

```
>> Object PERSONS = {}
>> Object STUDENT = {‘name’ : ‘Dominik’, ‘surname’ :
‘Knechta’ , ‘age’ : 25}
>>PERSONS[‘student’] = STUDENT
>> print(PERSONS.student.name)
“Dominik”

>>STUDENT[‘subjects’] = [ ‘math’, ‘biology’ , ‘chemistry’ ]
>>STUDENT[‘studying’] = true
>>print(PERSONS.student.subjects)
“math”,”biology”,”chemistry”
```

Image:

```
>>Image img = (“image.png”,0,0)
>>Image img2 = (“image2.png”,0,0,200,400)
>>createImage(img2)
```

4.4. Variables

Variable is a symbolic name associated with value which can be changed. Variable name consist of UPPERCASE or lowercase English characters and digits 0-9 and underscore ‘_’. Variable has to start with a letter or underscore and then followed by any sequence of letters or numbers. As mentioned before we are naming variables with UPPERCASE letters to distinguish it from functions or classes but this is just an arbitrary convention for now. Before usage every variable must be declared in the form “type variableName = value”. Variables declared inside some scope like function or if statement are local and only available in its scope. There are no global variables.

4.4.1. Examples

```
>>Number a = 25;
>>Number pi = 3.14;
>>String s= "string";
>>String s2 = 'dom';
>>Boolean b1= true;
```

```
>>Boolean b2 = 5==5;
>>Boolean b3 = '5' == 5;
>>List zoz = [2,4,5,6];
>>Object person = {"meno" : "Dominik", "priezvisko":
"Knechta"};
```

4.5.Operators

Webi 2.0 contains operators for assignment, comparison or logical and arithmetical expressions. It is possible to count different types with arithmetical operator ‘+’. The type of the new evaluated expression depends on types of two operands. Comparison operators are evaluating with “strict” types meaning expression “5” == 5 will be evaluated as false. Precedence of operators is remaining same. Complete precedence of operator is shown in the table below. To prioritize addition before multiplication we can use parentheses. Operators with same priority are evaluated from left. Whole process of how we dealing with operator priority will be described in next chapter.

OPERATOR	**OPERATOR TYPE**
( )	Parentheses
.	Member access
[ ]	Indexing operator
^	Exponentiation operator
*** / %**	Multiplication, division, modulo
+ -	Addition, subtraction
> , >=, <, <, ==, !=	Comparison operators
&&, ||, !	Logical operators
=	Assignment operator

4.5.1. Examples

```
>> 5 + “string”
>> “5string”

>> (5 == 5) + 1
>> 2
>> (5 == 6) + 1
>> 1

>> (5 == 5) + "test"
>> "truetest"

>> (5 ==6) + "test"
>> "falsetest"

>> 1 + 2 * 3
>> 7

>> (1+2) *3
>> 9

>> 6 / 3 * 2
>> 4

>> 6 / (3 * 2)
>> 1

>> 5 == '5'
>> false

>> 10 >= 5+5
>> true

>> (false || true) && true
>> true
```

4.6.Statements

A statement is an instruction which will perform some action. Webi 2.0 program statements consists of simple statements: variable declaration, assignment, function call, return call and structured statement which can contain other sequence of statements. Structured statements are: if statement, for statement, and while statement. Function call can be either a build-in function call like print() or assert() or user defined function. There are minor changes in syntax in structured statements which can be seen in example.

4.6.1. Examples

Return statement

```
>> List list = [1,2,3]
>> return size(list)
>> 3
```

For statement

```
>> for I = 0 to 10 do
     print(I) print(‘ ‘)
   end
>> 0 1 2 3 4 5 6 7 8 9
```

While statement

```
>> Number A = 10
>> while A > 0 do
     A = A - 2
     print(A) print(‘ ‘)
   end
>> 8 6 4 2 0
```

If else if else statement

```
>> if 10 > 5 do
      print(‘A’)
    else if 10 == 11 do
      print (‘B’)
    else do
      print(‘C’)
   end
>> ‘A’
```

4.7. Classes and objects

While we want to keep Webi easy for kids, we will simplify the whole process of creating classes and object. We are heavily inspired by JavaScript classes introduced in ECMAScript 2015. This system is based on prototype-based inheritance not objectoriented inheritance like Java. Classes in Webi 2.0 are in fact “special function” and they have two components: class declaration and class body. Classes can have only one constructor which is initialized at class declaration. Parameters passed to constructor are automatically created as class variables and have their values assigned when creating an object from class. To create a class method you simply define a new function in a class scope with common Webi 2.0 function declaration syntax. To create new class object you simply call assignment with keyword “new”. To see how classes are working see example below.

4.7.1. Examples

```
>> class Person(String Name, Number Age)
     def getAge()
       return ‘your age is ‘ + this.age
      end
     Number attendance = 0;
>> Person me = new Person(“Peter”,20)
>> Person Jake = new Person(“Jake”,34)
>> print(me.Name)
>> “Peter”
>> print(me.getAge())
>> “your age is 20”

>>me.attendance = me.attendance + 1;
>>print(me.attendance)
>> 1
>> print(Jake.attendance)
>> 0
```

4.8.Execution and error handling

As already mentioned bellow Webi 2.0 is interpreted language. This mean that instructions are executed one by one from the first instruction at first line of the source until last instruction. The only exception is Class and Function declaration, which can be declared in code also after function call, because we are visiting tree nodes with these declarations first. Error handling is happening at the same time while executing the instructions. This means that if some error is occurred on the 6th statement of our code, the previous 5 statements will run if they are correct and the execution only stops on the first error occurred. The reason for choosing this model is that error handling is easier to implement. Also our language is very similar to JavaScript which is using this model as well. Webi 2.0 also support event-based callback. This means that something is executed only after some event was triggered, for example clicking on button or pressing specific key.

4.8.1. Examples

Calling function before its delcaration

```
>> println(sum(10,5))
>> def sum(Number A, Number B)
     return A+B
   end
>> 15
```

Error occurance

```
>> Number A = 25
>> String S = "Hello"
>> println(S + " " + A)
>> "Hello 25"
>> print( S + " " B)
>> Error: Undefined variable B
```

4.9.Built-in methods

Webi 2.0 offer some basic build-in methods that are defined in our grammar for now, but they can me moved to separate library in next version. These methods can be called in as a classic function call like "println()" or there are methods assigned for specific value type like List. For example list.pop() or list.push(9). Build-in method for graphical interface are described in the next chapter 4.10.

4.9.1. Examples

```
>> List list = [2,4,6,8]
>> list.push(9)
>> size(list)
>> 5
>> list.pop()
>> 9
>> assert(size(list)==5)
>> false

>> print('Hello') print('World')
   'HelloWorld'
>> println("Hello") println('World')
   'Hello'
   'World'
```

4.10. Graphical user interface

Except the simple text output interface, Webi 2.0 also provide a basic graphical user interface = GUI. Because Webi is a web based language which is written and executed in browser we will use HTM5 canvas which is very popular and supported in all modern browsers for implementation of graphic interfaces. Webi 2.0 support to have only one canvas called **Playground** at same time therefore there is no need to initialize it with some id as a variable. Our IDE is by default working only in a text-mode, if we want to create some graphic elements, we just need to call the canvas with simple command shown in example. Webi Playground offer some build-in methods for drawing rectangles, inserting image, creating buttons or etc. All methods which will be implemented are shown in example, more can be added later. It is using standard Cartesian system for coordinates where x, y is starting at top left corner with 0, 0.

4.10.1. Examples

Drawing in canvas

```
>> initPlayground() // default width and height is 600x600px
>> strokeStyle(“red”)
>> fillStyle(“#6699ff”)
>> fillRect(10,10,50,50)
>> strokeRect(100,100,200,200)
>> strokeStyle(“green”)
>> linewidth(15)
>> drawLine(300,0,300,600)
>> drawText(“Hello World”,200,200)
>> Number version = 2
>> drawText(“Webi ” + version, 100, 300, “50px Comic Sans
MS”)
>> createImage(‘img.png’,400,400)
```

Assigning callbacks to button

```
>> def clear()
    clearPlayground()
   end
```

```
// ButtonID, ButtonLabel, onclickFunction (params
function-arbitrary)
>> createButton(‘btn1’,”Clear canvas”,”clear”)
```

Result

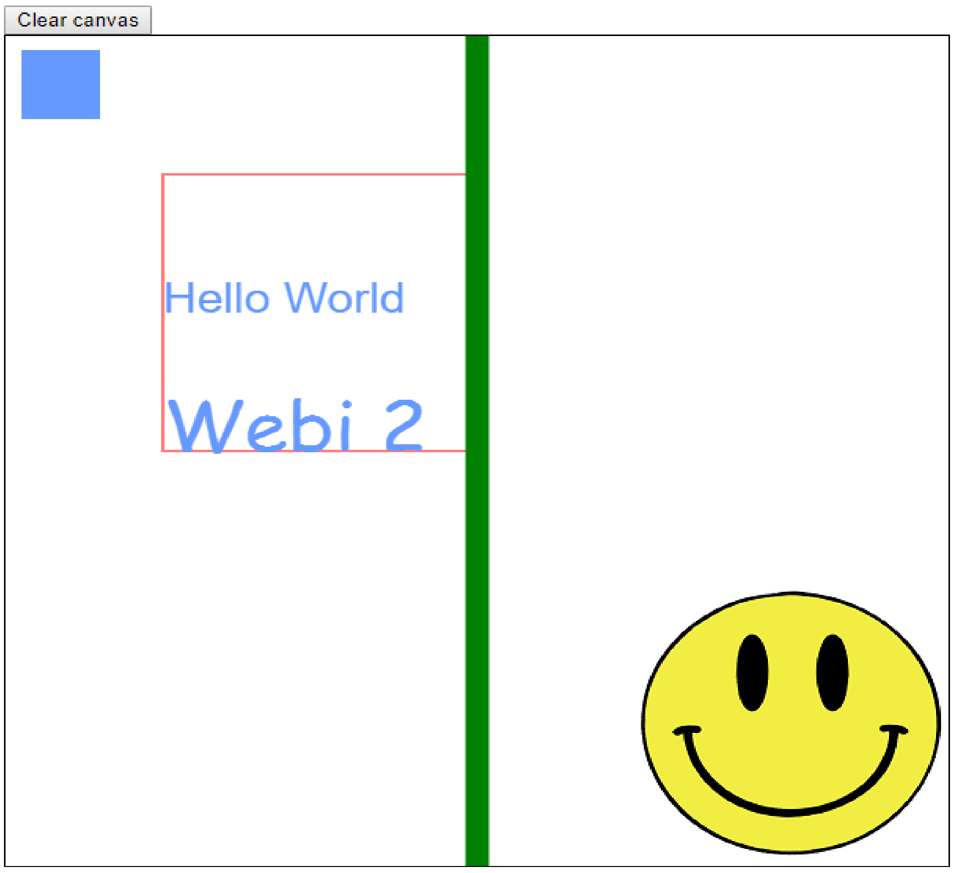


Figure 4 - result of above Playground commands

5. Designing language Webi 2.0

This part is describing the whole process of building our new language Webi 2.0. Starting from configuration of our selected tool ANTLR4, continued by defining grammar, interpreting generated parsing tree, dealing error handling and building our simple IDE. It will also describe problems which we encountered during development and how we deal with it. This is the first part, where our work is only to design fully working interpreted language. After the first part is done, we will continue on designing how the distributed environment will work in the Webi 2.0 (communication between client-server, remote references handling etc…).

5.1.Setting up the project

ANTLR tool actually consists of two essential parts:

- The tool which is generate the lexer and parser
- Runtime part, needed to run parser and lexer

While the first parts is only need by the programmer who is designing the language, the runtime part has to be included in the final product using our language Webi 2.0. The tool is always same and it is written in Java, but the runtime part is different for every language and must be available for both the developer and the user. As we previously mentioned we are going to compile the runtime tools to JavaScript, while our language is designed for web applications. To include the runtime part into the application we will use JavaScript library, which create a one file bundle.js which will contain all necessary tools and modules required to run Webi 2.0 in browser. For this purpose we chosen a bundler called Browserify. The process how to use browserify to bundle everything will be mentioned below.

For setting up the project we just need to download the .jar library of ANTLR and follow the official getting-started instructions to set paths for command line interface. The instructions are described on official ANTLR4 getting-started documentation [2].

5.2.Defining the grammar

The starting point of using ANTLR is defining your grammar. Grammar is a file with .g4 extension which contain the rules of the language that you analyze. After defining the grammar we just run simple command from command prompt.

```
$ antlr4 <options> <grammar-file-g4>
```

You can see list of all options in console or documentation, only important option for us is target language. The file containing the grammar needs to have same name as the grammar name declared at the top of the file. Our name of language is Webi, therefore we are creating the file Webi.g4 and running the command:

```
$ antlr4 –Dlanguage=JavaScript Webi.g4
```

Now we need to write lexer and parser rules for our grammar. We know that lexer takes individual characters and transform them in tokes, the parser uses this token to create a logical structure. Take a look at following example.

```
/* * Parser Rules */

expression op=( '+' | '-' ) expression     #addExpression


/* * Lexer Rules */

DIGIT     : [0-9];

Identifier : [a-zA-Z_] [a-zA-Z_0-9]*
```

This is only a fraction of the grammar but we can deduct a simple rules for making grammar here.

- Lexer rules are all UPPERCASE
- Parser rules are lowercase
- Syntax of the rule: rule_name : rule_definition ;
- Rules of lexer are analyzed in the order that they appear, the can be ambiguous

The table below summarize what ANTLR4 grammar notation we will use and its meaning

Syntax	Description
x	Match token, rule reference, or subrule x
x y …z	Match a sequence of rule elements
(.. | .. | ..)	Subrule with multiple alternatives
x?	Match x or skip it
x*	Match x zero or more times
x+	Match x one or more times
R: ….. ;	Define rule r
R : … | … | … ;	Define rule r with multiple alternatives

Table 2- ANTLR core notation

The other thing we have to consider is the approach how we are going to build a grammar. There are two alternatives here: bottom-up approach or top-down approach. **Top-down approach** is starting from most general which is representing the whole file. Then you continue with smaller parts, typically followed by some code blocks, imports, package declaration etc. Then you move to more low level rules and so on. This approach works best when you already know the whole specification of the language. **Bottom-up approach** is focusing the smallest possible elements first. Define how tokens are captured like strings, number then moving to basic expressions and so on until we define the rule which represents whole file. I have chosen the letter approach, because its advantage is that I can test and debug small bits of code while working on the next part.

5.3.Lexer rules

Before moving on defining our lexer rules, first we need to know how to distinguish parser rules from the lexer ones. Because lexer rules can use recursion too, lexer rule can be actually same powerful as parser rule. Defining this actually depends on the language but there are few rules which we are going to stick:

- Match and discard everything that parser doesn't need to see like comments, whitespaces

- Match common tokens such as keywords, strings, numbers in the lexer
- Match together into a single token type all lexical structures that parser doesn't need to distinguish. For example Integer and Float number can be match as Number

We are now going to define rules based on our Webi specification. We will need a rule for Identifier, Number, comments and whitespaces. We will also use **fragments** that are reusable building blocks for lexer rules.

Identifier: We have defined identifier as a nonempty sequence starting with uppercase or lowercase letter followed by sequence of letters digits or underscore. This is matching rule below. Note that dash (-) is symbolizing a range of English letters from a-z and A-Z.

```
Identifier : [a-zA-Z_] [a-zA-Z_0-9]*  ;
```

Number: In our language number can be either integer number or float. We define float number with integer part followed by '.' and floating part. In definition we first define fragments that define what is Digit and Int and from these rules we built a rule for Number.

```
NumberVar : Int ( '.' Digit* )?   ;

fragment Int : [1-9] Digit* | '0'  ;

fragment Digit  : [0-9] ;
```

Comments: We defined two types of comments. Single line starting with // and multiline starting with /* and ending */ . In our rule we use some regex expression symbols to define the rule. This sequence ".*?" is a notation to match everything while notation "~[ ]" mean to match anything except content in square brackets. We are also introducing " -> skip " command which tell the tool that this information is redundant and can be tossed away, therefore it will not be processed in any way.

```
Comment : ( '//' ~[\r\n]* | '/*' .*? '*/' ) -> skip ;
```

Whitespaces: In Webi 2.0 every matched whitespace is ignored and skipped same way as comments. With ignoring whitespaces we can write our code in single line or use unlimited numbers of new lines or spaces it doesn't matter.

```
Space : [ \t\r\n\u000C] -> skip ;
```

Strings: String in Webi are defined as any sequence of UTF-8 character between single or double quotes. This configuration additionally say that if we want to use double or single quotes inside the string we can escape them with backslash.

```
StringVar :
   ["] ( ~["\r\n\\] | '\\' ~[\r\n] )* ["]
| ['] ( ~['\r\n\\] | '\\' ~[\r\n] )* [']  ;
```

Keywords: Keywords are special reserved words in language that are specified for some specific functionality. We have to define all keyword that our language recognize before specifying Identifier rule to avoid incorrect recognition. The example how to define keywords is below. You can view all defined keywords in the defined grammar file.

```
Playground: 'Playground';
Button:     'Button';
Or:          '||';
```

5.4.Parser rules

We are now defining the parser rules which will finally produce us a parser tree. Parser rules follow same syntax as lexer rules, meaning rule name followed by colon and rule terminating with semicolon. The lowest parser rule in our grammar is an expression. The most upper rule in our grammar is called "parse" containing the whole source code. Parse rule is contacting a block rule followed by end of file terminator. Then the rule block is divided and can contain statement, function declaration, class declaration or return expression. Both parser and lexer rules are written in the same file Webi.g4. Below we are including an example of block rule and function declaration.

```
block : ( statement | functionDecl | classDecl )* ( Return
expression )? ;

functionDecl : Def Identifier '(' idList? ')' block End ;
```

5.5.Testing the grammar

In previous section we showed an example how to write lexer and parser grammar for Webi. The whole grammar is stored in the file Webi.g4 which can be found in the source of this thesis. Now it is time to test the grammar before continue on the next step. ANTLR is providing a testing tool in runtime library which can tell us lot of information about how recognizer matches the input. Although we are compiling our tools for JavaScript, the testing tool is only available in Java. We will run following commands to compile our grammar to Java for testing purpose.

```
$ antlr4 Webi.g4
$ javac Webi*.java
```

After this we are able to run test tools to check if our grammar is correct. Let's first test a simple variable assignment. We need to specify a rule name as a parameter which in this case is "assignment". We run the following command. Note that symbol ^Z signalize the EOF.

```
$ grun Webi assigment -tokens
  Number A = 56;
  ^Z
```

We will see the following output from the console.

```
[@0,0:5='Number',<'Number'>,1:0]
[@1,7:7='A',<Identifier>,1:7]
[@2,9:9='=',<'='>,1:9]
[@3,11:12='56',<NumberVar>,1:11]
[@4,15:14='<EOF>',<EOF>,2:0]
```

We can see that tokens are identified properly and no error has occurred. In the next example we forgot to put equals symbol to see if the tester can identify wrong assignment.

```
$ grun Webi assigment -tokens
  Number A  56;
  ^Z
```

This will result in following ouput:

```
[@0,0:5='Number',<'Number'>,1:0]
[@1,7:7='A',<Identifier>,1:7]
[@2,9:10='56',<NumberVar>,1:9]
[@3,13:12='<EOF>',<EOF>,2:0]
line 1:9 mismatched input '56' expecting {'[', '='}
```

The testing tool successfully recognized the missing symbol which has to be equal symbol or left square bracket to set the index of a value. We can also get the graphical result in the form of tree, which is even more practical. Let's test if our grammar can handle advanced expressions and operator priority.

```
$ grun Webi expression -gui
6 + 5 - 8 / 2
^Z
```

Output by the command:

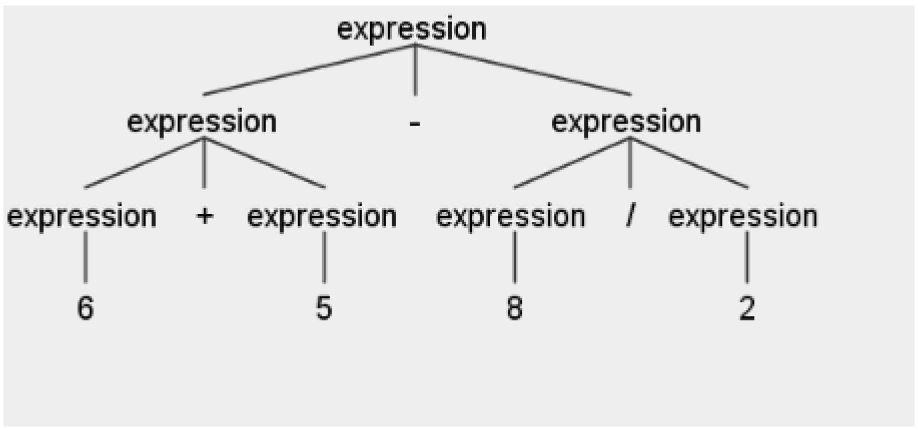


Figure 5 –example 1. parse tree from ANTLR

```
$ grun Webi expression -gui
  6 + (5 - 8) / 2
  ^Z
```

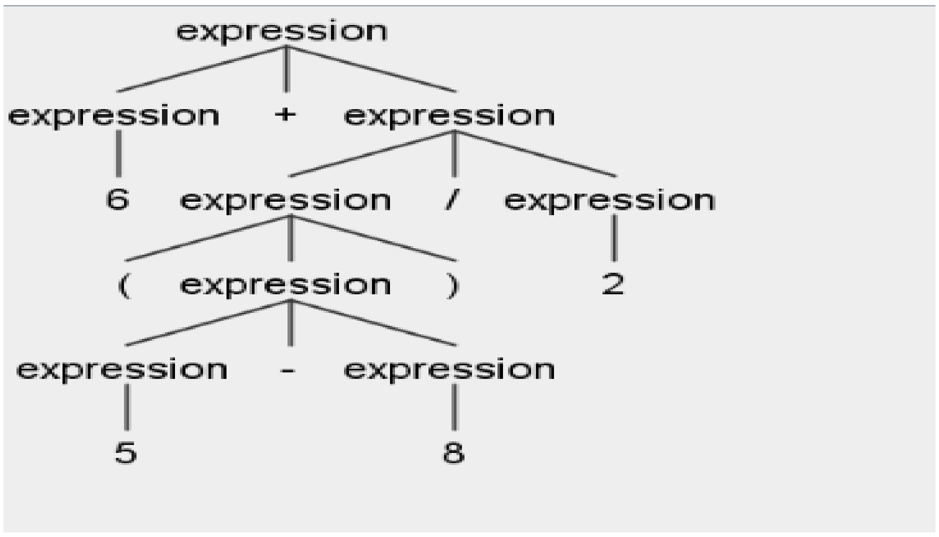


Figure 6 – example 2 from parser tree

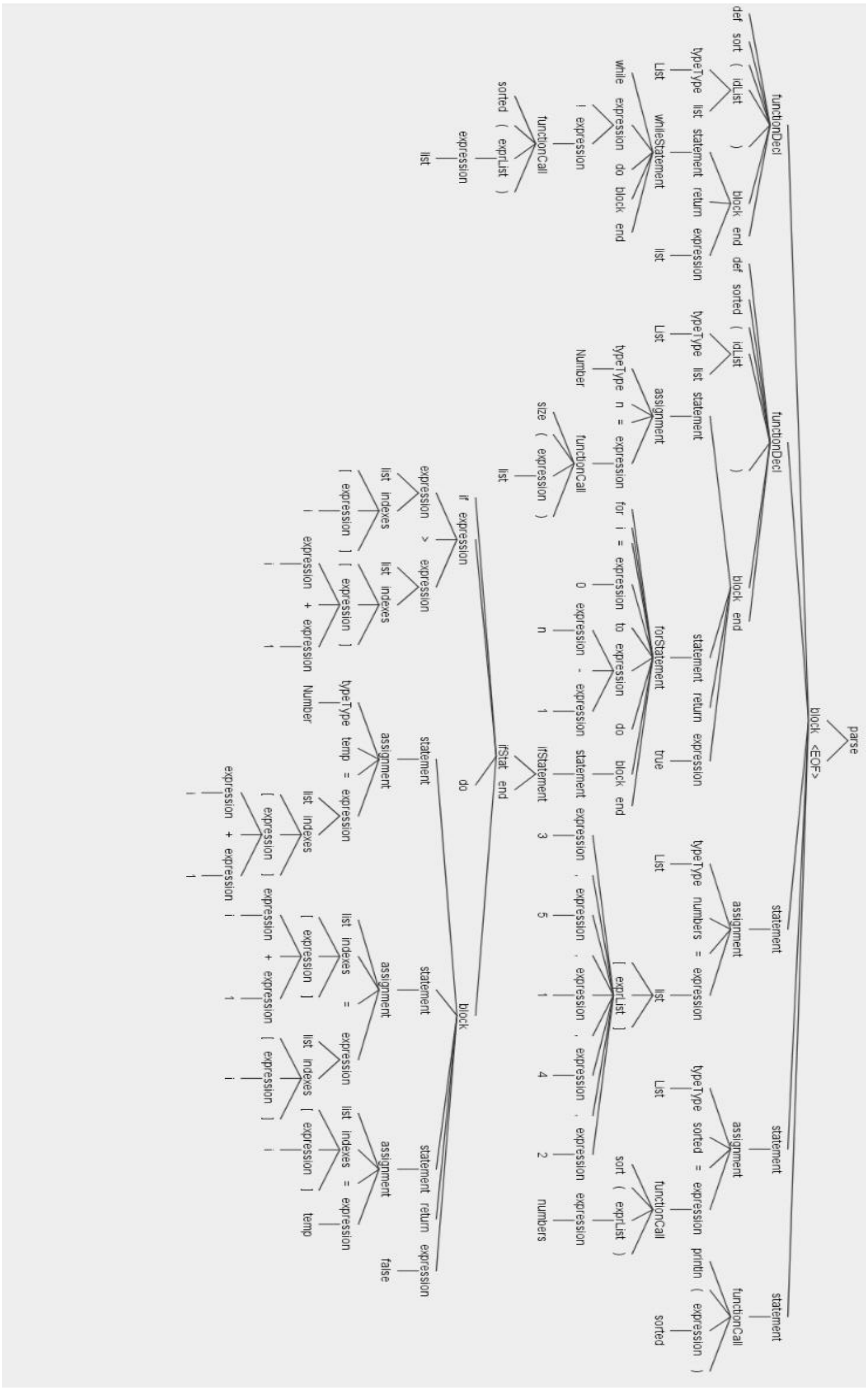
We can see that our grammar is well defined and prioritize operators correctly. The process how we deal with operator precedence and ambiguity is described in the below section: Dealing with ambiguity and operators precedence.

Before moving to last phase which is building language application using parsed tree, let’s test how a parse tree of a simple application in Webi 2.0 language would look like. Let’s test a simple bubble sort written in Webi.

```
def sort(List list)
while !sorted(list) do
    end
return list
end

def sorted(List list)
Number n = size(list)
for i=0 to n-1 do
    if list[i] > list[i+1] do
        Number temp =
list[i+1]
list[i+1] = list[i]
list[i] = temp
return false
end
end
return true
end

List numbers = [3,5,1,4,2]
List sorted = sort(numbers)
println(sorted)
```



parse
block <EOF>
functionDecl
def sort (idList) block end
typeType list statement return expression
List whileStatement list
while expression do block end
! expression
functionCall
sorted (exprList)
expression
list
functionDecl
def sorted (idList) block end
typeType list statement
List assignment
typeType n = expression
Number functionCall
size (expression)
list
forStatement
for i = expression to expression do block end
0 expression - expression
n 1
statement return expression
true
ifStatement
if expression do block
ifStat end
expression > expression
list indexes list indexes
[expression] [expression]
i expression + expression
i 1
statement statement statement return expression
false
assignment
typeType temp = expression
Number list indexes
[expression]
expression + expression
i 1
assignment
list indexes = expression
[expression] expression + expression [expression]
i 1 i
list indexes [expression]
i
assignment
list indexes = expression
[expression] temp
i
statement
assignment
typeType numbers = expression
List list
[exprList]
expression , expression , expression , expression , expression
3 5 1 4 2
statement
assignment
typeType sorted = expression
List functionCall
sort (exprList)
expression
numbers
statement
functionCall
println (expression)
sorted

Figure 7 - example of parse tree for bubble sort

5.6.Dealing with precedence and ambiguity

The ambiguity problem in building language process occurs when a rule can match single input stream in more than one way. This will usually occurs in expressions rules where some expression with multiple operators can be interpreted in more ways. To illustrate a problem lets imagine a simple arithmetic expression language that has multiply and addition operators and integer atoms. Expressions are self-similar, we can say that a multiplicative expression will be two subexpressions joined with the ‘*’ operator. We can say similarly addition is two subexpressions joined by ‘+’. Last option is to have simple integer as expression.

```
expr : expr '*' expr // match subexpressions with '*'
     | expr '+' expr // match subexpressions with '+'
     | INT // matches simple integer atom
     ;
```

The problem is that this rule is ambiguous just for some inputs phrases. For example rule is fine for single operator expressions like 5+5 or 5*5 because there is only one way how to match them. The problem will occur with input like “ 1 + 2 * 3” . This can be interpreted as multiplying the result of 1 + 2 or addition the result of 2*3. This is the question of operator precedence and from math we know that multiplication has bigger priority over addition. However this is not working in parsers by default and we have to specify it.

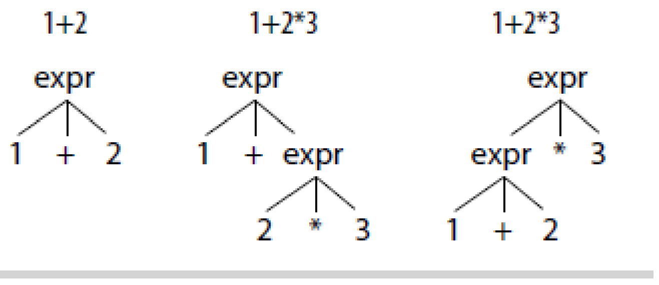


Figure 8 – parse tree interpretation

ANTL4 resolves ambiguities in favor of the rule given first. This means that if we specified multiplication subrule for expression before addition, ANTLR will resolve the input 1+2* 3 in favor of multiplication. By default operators are associated from left to right as in most programming languages. However there are some cases where input should be evaluated from right to left, for example exponentiation. The following example show difference between left and right association in exponentiation input. The right tree is our desired output.

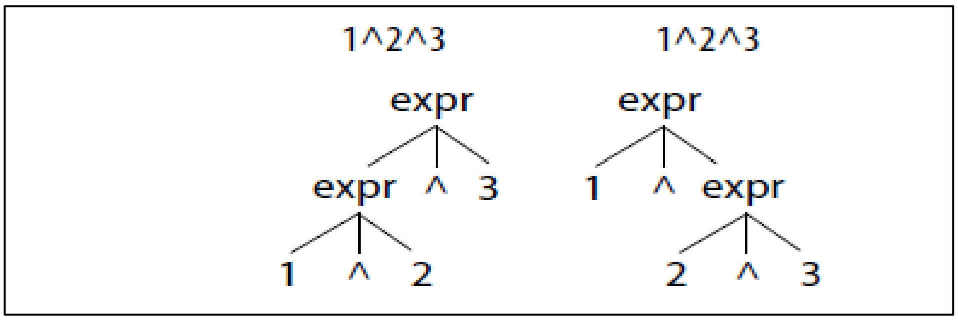


Figure 9 – parse tree interpretation for exponentiation

To achieve right to left association we use option <assoc=right>

```
expr : expr '^'<assoc=right> expr // ^ operator is right
associative
      | INT
      ;
```

To combine all three operators: "+", "*" and "^", we have to place exponentiation subrule before other because it has the highest precedence. We also know that operators multiplication '*' division '/' and modulo '%' have same precedence as well as addition '+' and 'subtraction '-'. We can group and match these operators into one rule like in example below. The final expression rule, which can handle all basic arithmetic operations will then look like this:

```
expr : expr '^' <assoc=right> expr
      | expr op = ( '*' | '%' | '/' ) expr
      | expr op = ( '-' | '+' ) expr
      | INT
```

Now when our grammar is unambiguous and successfully tested we can move on final step which is building applications using generated parse tree.

5.7.Building application using parse tree

To create an application, we have to execute some appropriate code for each of tree node (phase). For this we will operate on parse tree which was generated automatically by

ANTLR parser. The operations on the tree are now just matter of some programming language, in our case JavaScript, so there is no more ANTLR syntax used.

First we are going to analyze data structures and class names that are used for recognition and parse trees. We already know that when processing an input, lexer process characters and pass them as tokens to the parser, which will checks syntax and creates a parse tree. Corresponding classes for this are CharStream, Lexer, Token, Parser and ParseTree. The pipe connecting lexer and parser is called TokenStream. For clarity we will always output parse tree of Webi application into console. For now let's look at example bellow to better understand how objects of these types connect to each other in the memory.

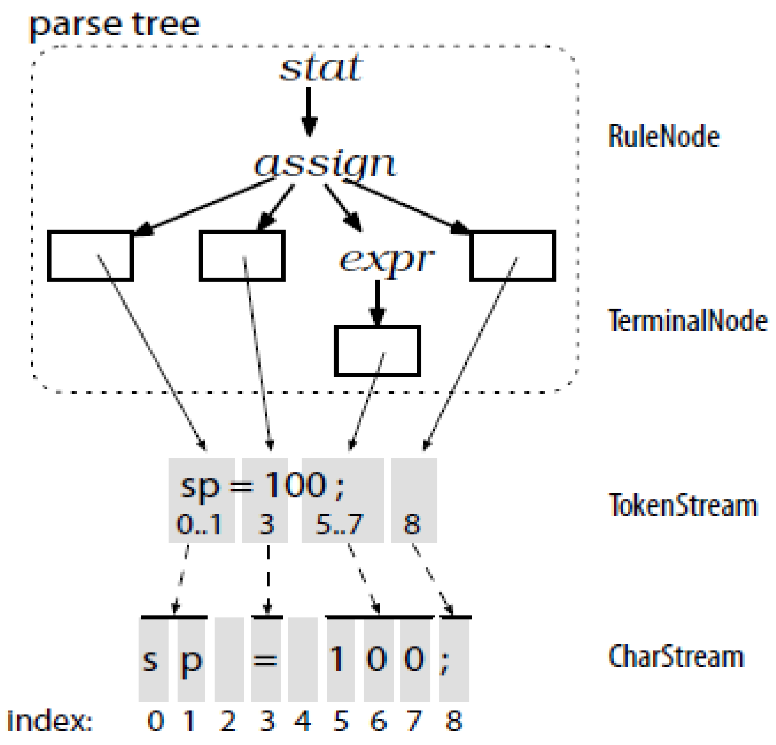


Figure 10 – diagram representing generated parse tree, (The Definitive ANTLR4 Reference) [13]

In the diagram above we see that leafs node in the tree are containers that point at token in the *TokenStream*. There are no tokens associated with whitespace character (on index 2 and 4) because previously we declared our lexer to throw away whitespaces. We also have subclasses *RuleNode* and *TerminalNode* that are responding to subtree root. Rule node is a subclass genreted for every rule. It has method like *getChild()* or *getParent()* for easy access to the elements of specific node. Based on this description we can write a code to perform a depth-first walk of the tree. We can perform any action on visiting each node of the tree. Typical action can be computing results, updating or creating data structures or generating output. ANTLR offers two mechanisms for walking trees. They are visitor and listener. In Webi implementation, I tried to implement both mechanisms, one of them however appeared as unsuitable for our language. Comparison of visitor and listener and why one of this is not suitable is explained in following section.

5.8.Listeners and Visitors

There are two mechanisms for tree-walking provided in ANTLR. The default one is listener and the other alternative is called visitor. Webi 2.0 is using visitor pattern for tree-walking because listener has appeared as not suitable for our language specification. The reason why will be clear when I explain the process how both these mechanisms work.

5.8.1. Parse-tree listener

For walking a tree and trigger calls, ANTLR runtime provides class *ParseTreeWalker*. To create some application in Webi language using listener, we have to build methods in a *ParseTreeLister* file. This is the file specific for each grammar that contains methods for every rule. As the walker encounter any node, the method *onEnter()* is called. For example on encountering an assignment rule the method enterAssign() is triggered. The AssignContext node is passed to this method. Simillary, after walker visit all children of assign node, it will trigger *onExit()* method. In this case exitAssign(). Figure below is showing ParseTreeWalker performing depth-first walk indicated by dash line.

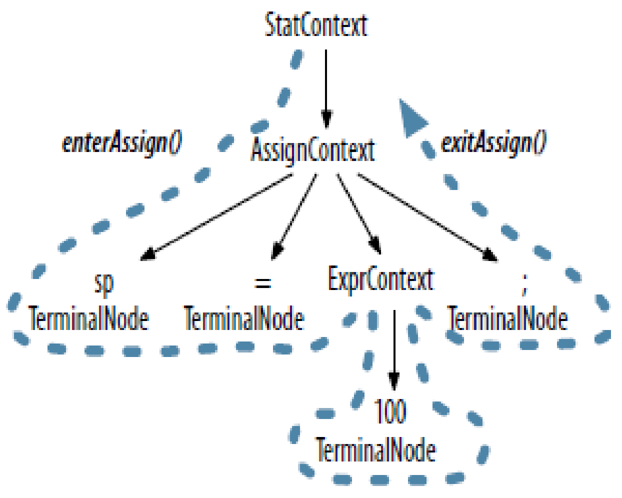


Figure 11 – ParseTreeWalker performing a depth-first walk, (The Definitive ANTLR4 Reference) [13]

The biggest advantage in using listener mechanism is its automation. We don’t need to explicitly visit children of every node. The next figure bellow is showing the whole sequence of calls for our statement.

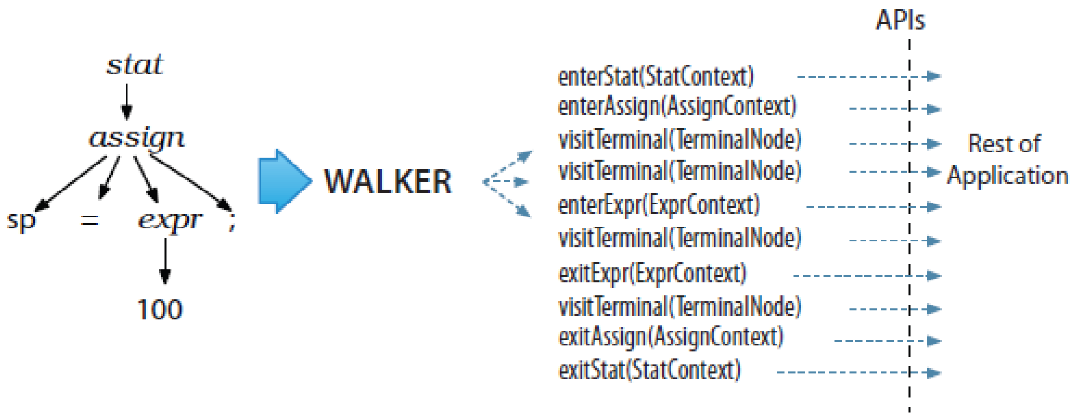

stat
assign
sp
=
expr
;
100
WALKER
APIs
enterStat(StatContext)
enterAssign(AssignContext)
visitTerminal(TerminalNode)
visitTerminal(TerminalNode)
enterExpr(ExprContext)
visitTerminal(TerminalNode)
exitExpr(ExprContext)
visitTerminal(TerminalNode)
exitAssign(AssignContext)
exitStat(StatContext)
Rest of Application

Figure 12 - call sequence of ParseTreeWalker, , (The Definitive ANTLR4 Reference) [13]

5.8.2. Parse-tree visitor

In opposite to listener which walks the tree automatically from root to leaves, in visitor we have walk the tree manually. This is suitable if we want to control how and when we will evaluate children nodes. Following figure show the process how visitor is walking the parse tree.

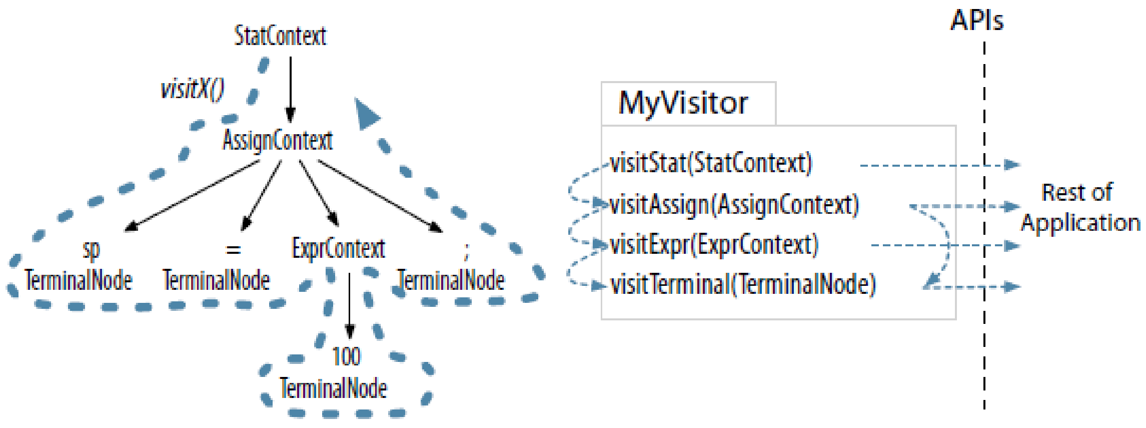


Figure 13 – call sequence of Visitor parse tree, (The Definitive ANTLR4 Reference) [13]

Thick dash line represent a depth-first walk of parse tree. The thin dash line is indicating call method sequence among visitor method. To walk-of the tree by visitor we need to create visitor implementation and call method *visit()* passing parse tree as a parameter. Upon seeing the root node, visitor call its visit method. In the example above visitStat() . From there visitStat() would call visit() with children as arguments for continuing the walk and so on. With visitor mechanism we can therefore only implement visit methods for rules of interest and skip the others.

During the implementation I have chosen a Listener parse tree mechanism, however it has proven as insufficient. We mentioned in chapter 4.8 that it is possible to call a function call which in code is above function declaration. The listener nature is visiting all node automatically from root to leafs, meaning that if function call will be called before function declaration it will lead to undefined function error. Same applied for classes and object creation. Therefore whole listener had to be rewritten to visitor method.

5.9.Building the visitor and its methods

Finally it is time to write a visitor and implements its methods that will execute the code for our language. The visitor will walk the whole generated tree and execute the commands or evaluate the expressions. Visitor is not generated by default therefore we need to pass -visitor option when we compiling our grammar.

```
antlr4   -Dlanguage=JavaScript Webi.g4 –visitor
```

After this step, the structure should contain the file *WebiVisitor.js*. It contains methods and functions that we are going to override with our own visitor. We are not going to modify it, because changes would be overwritten every time the grammar is regenerated. Now let’s generate structure for our application.

Index.html

This is an entry file for our application, which is loaded first. It contains textarea where we will write our Webi 2.0 code and Compile button with onClick event which triggers the execution process. This file also contain an empty div where text output of the program will be displayed. Last element is graphical canvas for graphical outputs and elements.

Index.js

This is the file which serves as a listener for onClick compile button. When button is clicked we fetch the whole input text from textarea and run buildAst() function with parameter inputText.

build-ast.js

This is the file where the construction logic of our tree and visitor is saved. First we need to require antlr4 runtime library and classes generated by ANTLR. Note that parts of code in this section are only snippets for explanation purpose and redundant lines may be skipped.

```
var antlr4 = require('antlr4');
var WebiLexer = require('./WebiLexer'). WebiLexer;
var WebiParser = require('./WebiParser').WebiyParser;
```

Then we define the buildAst() function which will look like this.

```
function buildAst(inputText){
    var chars = new antlr4.InputStream(inputText);
    var lexer = new WebiLexer(chars);
    var tokens = new antlr4.CommonTokenStream(lexer);
    var parser = new WebiParser(tokens);
    var tree = parser.parse();
    var visitor = new Visitor();
    visitor.visit(tree);
}
```

Lines 1-5 shows the foundation of every ANTLR program: you create the stream of chars from the input, you give it to the lexer and it transforms them in tokens that are then interpreted by the parser. Then, on line 6, we create a tree by calling method parse(). Finally we create our custom Visitor which is overwriting default visitor and call method visit() passing tree as parameter.

Visitor.js

```
var WebiVisitor = require('./WebiVisitor).WebiVisitor;
var WebiParser = require('./WebiParser').WebiParser;

Visitor.prototype = Object.create(WebiVisitor.prototype);
Visitor.prototype.constructor = Visitor;

function Visitor(functions, scope, classes) {
    this.scope = scope;
    this.functions = functions;
    this.classes = classes;
    WebiVisitor.call(this);
    return this;
}

Visitor.prototype.visitBlock = function (ctx) {
     this.scope = new Scope(this.scope);

     if (ctx.statement() !== null){
         this.visit(ctx.statement());
     }

     if (ctx.expression() !== null) {
       var returnValue = this.visit(ctx.expression());
       this.scope = this.scope.getParent();
       throw returnValue;

     }

      this.scope = this.scope.getParent();
      return null;

     };
```

This is the core file which is going to evaluate the expressions, assign variables or execute commands. First lines are declaring required files WebiParser and WebiVisitor which are we going to override. Second part of code is declaring our Visitor and its constructor. In the last part of snippet we are visiting the basest rule of our grammar which is block. We don’t have to override method visitParse() which is root node of our grammar because we don’t need to change its functionality. The **ctx** argument is an instance of a specific class context for the node that we are visiting. So for visitBlock it is block and so on. This specific context will have the proper elements for the rule that would make possible to easily access the respective tokens and subrules.

The method is then checking if it contains children nodes defined in our grammar. This can be statement, functionDeclaration, classDeclaration or return statement. We don’t checking declarations in this Visitor because we want to have all functions and classes already declared when we encounter function call or object creation. For that purpose we created SymbolVisitor which is described below.

SymbolVisitor.js

Symbol visitor work on same principle as our first visitor except the fact that in this one we only visit function and classes declarations. All functions and classes are then stored in object functions{} or respectively classes{} which are passed to Visitior.js constructor and therefore all functions and classes are available where walking the main code block. Snippet below show how function is parsed and saved in functions object. Same process is applied for classes.

```
SymbolVisitor.prototype.visitFunctionDecl = function (ctx) {
    var params = [];
    if(ctx.idList() !== null){
        for(var i =0;i<ctx.idList().Identifier().length;i++){


          params.push([ctx.idList().Identifier(i),ctx.idList(
          ).typeType(i).getText()]);

          }

     }
    var block = ctx.block();
    var id = ctx.Identifier().getText() +params.length;
    functions[id] = new Function(params,block);
    };
```

Scope.js

As we declared in Webi specification, we are not using global variables. Therefore variables declared inside a function, for or while loops will not be available and recognized outside its scope. For this we created file Scope.js which is saving its variables into one object. Every scope is also keeping track of its parent scope. When the parsing start, global scope with null parent is created. In this way we can access variables in opposite way, meaning variables declared in global scope will be still available in function declarations or loops. The Scope.js is always looking for a variable in its current scope and if nothing is found it will move to parent scope. Therefore we can have variable with same name declared in global scope and for example some function scope. Bellow snippet is showing how the resolving of variable works.

```
Scope.prototype.resolve = function (id) {
    var value  = this.variables[id];
    if(value != null){
        return value;
    }
    else if(!this.isGlobalScope()) return
this.parent.resolve(id);
    else return null;
};
```

Function.js, Class.js

These files are JavaScript prototype classes which are instanced with new function or class declaration. They contain method invoke which will resolve its parameters and then create a new instance of visitor which will walk and execute the function or respectively class body.

bundle.js

The last step is to pack all the dependencies with some bundler into one file. We are using Browserify plugin which will generate a file bundle.js and then we include this script in an index.html file which serves as the code editor.

```
browserify index.js -o bundle.js
```

5.10. Designing a multiuser web application in Webi

By now we only implemented Webi 2.0 as regular programming language. The aim of the language is also creating web application without the need of programing server side. For now, we design a simple model for creating web applications in Webi which allow us communication between 2 or possibly more users. This model is based on callback therefore it aimed for graphical application, which contains some callback elements for example buttons.

With this model there are many possibilities to create a simple game for 2 players. Example of a game using this model can be a simple quiz game where a user will define some questions and create buttons which will symbolize an answer for the question. When players click on the button, the message will be emitted to the host who will evaluate the answer and send back a socket with the result.

Another examples can be some reflex challenge game where user has to click on some element first to win, or very simple Rock – Paper – Scissors game. Although this model is simple there are many options in its usage and it is a good start for children who want to learn how to create web application. Figure below is showing how the communication between users could look like. For communication between users we will use technologies NodeJS and socket.io.

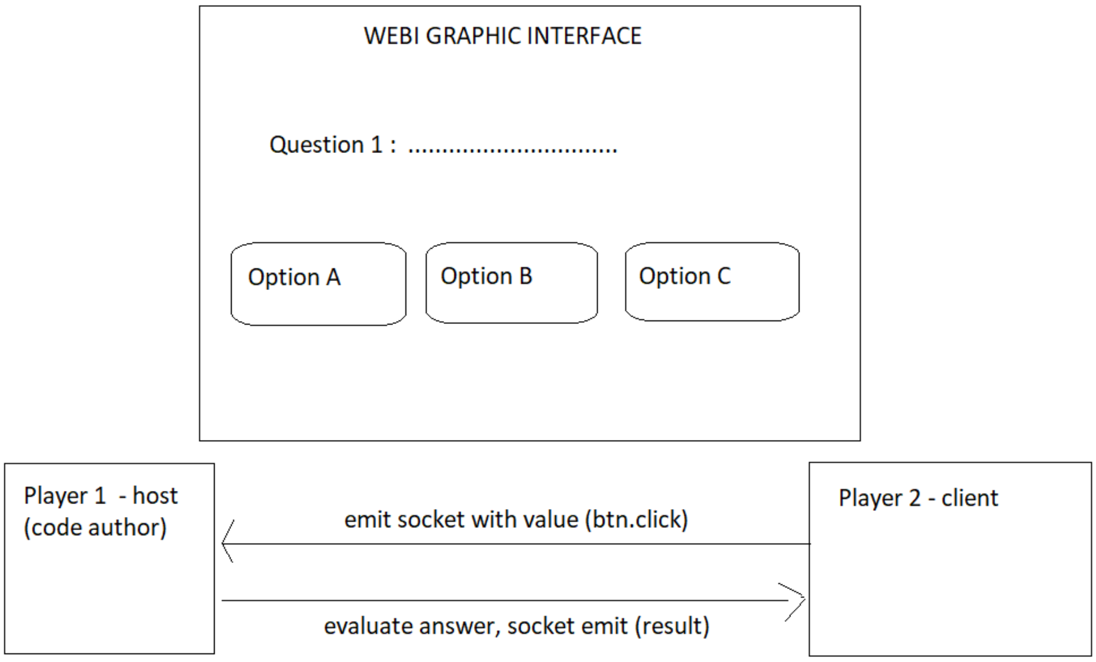

WEBI GRAPHIC INTERFACE
Question 1 : ....
Option A
Option B
Option C
Player 1 - host (code author)
emit socket with value (btn.click)
evaluate answer, socket emit (result)
Player 2 - client

Figure 14 – example model of communication between two users

Compared to Webi 1.0 which described more advanced remote references handling model this is just a simple prototype, but it can be easily implemented so it will be possible to create real working simple games already in Webi 2.0. The model can be evolved and contain more advanced features in possible further versions of Webi language.

6. Design of Integrated Development Environment

Our new integrated development environment (IDE) is very similar to version in Webi 1.0. The main difference is that everything is located in same window, therefore you can edit code without going back. The main window consist of 3 parts:

- Textarea for writing Webi 2.0 code
- Console output
- Canvas output

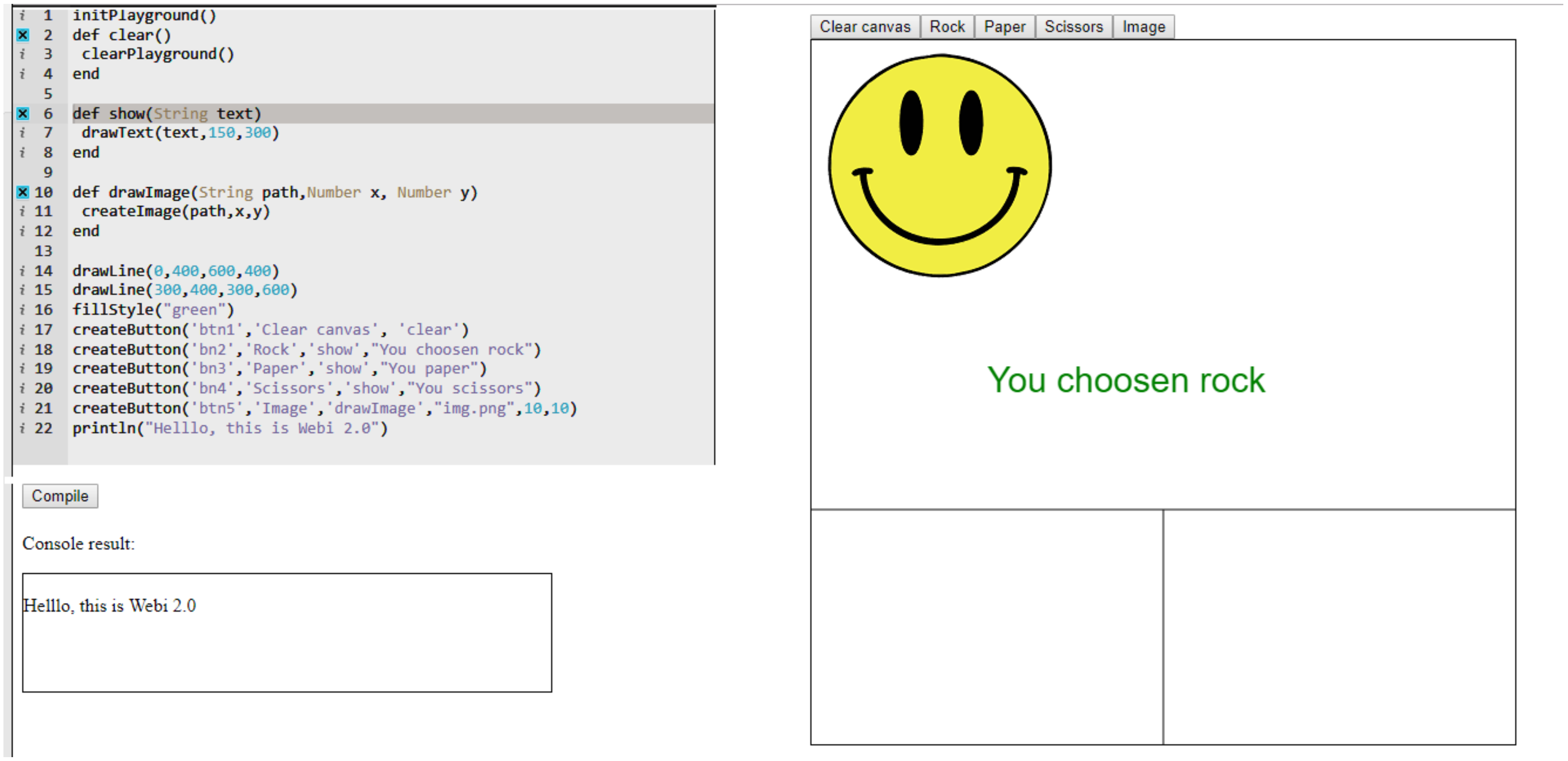
For the textarea we used a code editor called ACE [14] which offers syntax highlighting, line wrapping, search and replace with regular expression and many more. For now syntax highlighting is only set for JavaScript syntax which is mostly similar to Webi. The full support of syntax highlighting is the matter of next Webi version.

The console output will display outputs from print() or println() calls or any semantics error occurred during program execution.

The canvas output is by default hidden and has to be initialized first. The default canvas dimensions are 600 x 600 px.

Webi 2.0 interface will also contain few examples of code to better understand Webi 2.0 syntax. They will be loaded by clicking on their links.

To run the Webi 2.0 application, user need to click on button “Compile” which will invoke the whole execution process. When running application in a graphical mode, user should refresh window if he want to re-compile their code again. Otherwise a warning may be shown saying that “you are trying to create button which already exists”. Anyway this is only a warning and program will continue in its execution. Bellow figure is a real screenshot of graphical application built in Webi.



```
initPlayground()
def clear()
 clearPlayground()
end

def show(String text)
 drawText(text,150,300)
end

def drawImage(String path,Number x, Number y)
 createImage(path,x,y)
end

drawLine(0,400,600,400)
drawLine(300,400,300,600)
fillStyle("green")
createButton('btn1','Clear canvas', 'clear')
createButton('bn2','Rock','show',"You choosen rock")
createButton('bn3','Paper','show',"You paper")
createButton('bn4','Scissors','show',"You scissors")
createButton('btn5','Image','drawImage',"img.png",10,10)
println("Helllo, this is Webi 2.0")
```

```
Helllo, this is Webi 2.0
```

You choosen rock

Figure 15 – IDE for Webi 2.0

7. Execution

Our application is working in two modes. The first mode is an “offline mode” for single user application which doesn’t require communication between users. This mode can be run offline just open index.html file from source code. This version will be also available on webpage. Actual link to webpage is included in readme file on github, where most up-to-date version of Webi will be uploaded.

Other mode is an “online mode” which will allow communication between users and require Node.js. Requirement for this is running node server, therefore this version will only be available online on specific page. The link for this version will be included on github as well. Alternatively you can install whole project locally by installing node.js and then by running command `npm install` to install all dependencies followed by `npm run app.js` to start server locally.

Link to github Webi repository: https://github.com/dominik01/webi

8. Conclusion and further directions

The aim of this thesis was further development of Webi programming language. Webi should be a prototype language with distributed environment on server and client. It is a text-based, clear and simple language with syntax that is easy to learn for children. Language should be universal for both server and client side components. Our goal was redesigning computation model based on asynchronous nature of JavaScript, redesign of internal representation.

At first I have analyzed the theory of whole process how to build a new programming language, including building own parser and lexer, comparison between compiler and interpreter but also analyzing language recognition tools which could have been useful in process of making language. This step was essential before moving to next part which was actual further development of Webi.

Before the real development process, I had analyzed current state of Webi language. Although the first Webi language propose some new interesting techniques for building a language, for example internal representation in form of JavaScript objects, they actually appeared as not suitable for further development. My first attempt was to continue with the techniques defined in first Webi version, however during implementation process I had encountered issues which made me decide to look for a better alternative. It has appeared that the technique was really hard to understand and debug when trying to build a more complex application, moreover it was really prone to bugs.

For this reason, I decided to completely rework the Webi language and starting from the beginning, marking the first version of language as Webi 1.0 and new version as Webi 2.0. The new version of Webi is very different but it still tries to keep most of the benefits and easy syntax which was designed in first version. The main difference between these versions is that Webi 1.0 is compiled language whereas Webi 2.0 is interpreted.

For building Webi 2.0, I have chosen a language recognition tool called ANTLR4 which can help us in generating parse tree that is necessary step in building a language. First I had to define a grammar, which was inspired by JavaScript and Webi 1.0. From defined grammar a parse tree is generated. Next step was walking the tree and run the whole logic for every tree node visited. There are two method of walking the tree, listener and visitor.

During the implementation phase I have started with listener method but it has appeared as insufficient for Webi 2.0 specification and I had to choose visitor method instead.

Last part of the thesis was design and implement a model for web applications written in Webi 2.0. I have proposed a simple model which allow us simple communication between two or more users. This model is suitable for programming an easy game like some quiz or any similar games.

Summarizing all, we can say that not all of the original goals were accomplished. However I have designed and implemented new Webi 2.0 language which is working well and user can program real simple application or games in it. Example of some working applications in Webi 2.0 language are included in source code and web page. Unfortunately designing and building a new programming language which is 100% working is a very long process which is out of a scope of diploma thesis, but I believe that I have created a working prototype which is easy to understand and therefore suitable for further development.

There are still many possibilities for further development of Webi language like enhancing executional model, improving stability of the language, auto-completion and syntax highlighting or advanced IDE with more functions.

9. Bibliography

1. **Faiçal Tchirou.** Compilers and interpreters**:** Let's build a programming language series. [Online] [Cited: January 6, 2018] hackernoon.com/compilers-and-interpreters-3e354a2e41cf
2. **Antlr4**. Antlr4. [Online] [Cited: January 4, 2018] antlr.org/papers/allstar-techreport.pdf
3. **JS/CC**. Welcome to JS/CC. [Online] [Cited: January 4, 2018] jscc.brobston.com/documentation/introducing/welcome.html
4. **Nearly.js**. Nearly.js. [Online] [Cited: January 4, 2018] nearley.js.org/
5. **ECMAScript5.1**. ECMAScript® Language Specification. [Online] [Cited: April 10, 2018] ecma-international.org/ecma-262/5.1/
6. **Node.js**. Node.js documentation. [Online] [Cited: April 10, 2018] nodejs.org/en/docs/
7. **Socket.io.** Socket.io. [Online] [Cited: April 10, 2018] socket.io/
8. **Browserify.** Browserify. [Online] [Cited: April 10, 2018] browserify.org/
9. **Andrew B. Begel**, Bongo: A Kids' Programming Enviroment for Creating Video Games on the Web, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 23. 1997
10. **V. Pohorencova**. Educational Programming Language for Creating Interactive Web Application. Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Bratislava, May 2017
11. **Aho, Alfred V., et al**. Compilers Principles, Techniques, & Tools. Boston : Greg Tobin,2007. ISBN 0-321-48681-1.
12. **Mogensen, Torben Ægidius**. Basics of Compiler Design. Copenhagen : lulu.com, 2000.ISBN 978-7-993154-0-6.
13. **Terence Parr**. The Definitive ANTLR4 Reference. Dallas, Texas: The Pragmatics Programers,2013.ISBN-13:978-1-93435-699-9
14. **Ace**. Ace editor. [Online] [Cited: April 30, 2018] ace.c9.io/

10. Apendix

An appendix contain a CD with all files for Webi 2.0 application.