TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Institute of Computer Science
Chair of Theoretical Informatics

Implementing parser for CoCoViLa specification language using context-free grammar

Master thesis

Student: Ilja Nafigin
Student code: 121841IAPM
Advisor: Pavel Grigorenko

Tallinn
2014

I declare that this thesis is the result of my own research except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Signature	
Name	Ilja Nafigin
Date	June 2, 2014

ABSTRACT

The specification language plays the central role in model-based software development platform CoCoViLa. In CoCoViLa, domain-specific models of concepts and computational problems are defined visually, using diagrams, or textually, using the specification language. Visual specifications are translated into the textual form. That is why it is important to handle specifications correctly. CoCoViLa used to have a parser based on regular expressions which contained lots of mistakes and also did not allow to develop and extend the specification language without effort.

In this work the theory of formal languages is studied. The comparison of existing software tools for automatically generating parsers from context-free grammars is given.

A context-free grammar of CoCoViLa specification language is created. The realisation of a new parser generated by the chosen tool called ANTLR is described. The evaluation of the parser is presented with the description of new features as well as the comparison with the old parser.

CONTENTS

1. INTRODUCTION

CoCoViLa is a model-based software development platform implemented in Java language [1]. It enables to develop visual domain-specific languages and use these languages to specify various computational problems in a declarative manner (i.e. specifying *what* should be computed instead of *how*). Built-in *program synthesis engine* takes care of generating valid programs for solving specified problems automatically. Visual specifications in CoCoViLa are translated into the textual *specification language*. This is a Java-like language that allows declaring variables, specifying functional dependencies between variables, etc. The specification language's role is crucial to the whole platform and it is very important that specifications implemented in this languages are handled correctly. Unfortunately, this is not always the case.

CoCoViLa uses a manually implemented parser based on regular expressions in order to parse specification language. And this parser is a *bottleneck* that prevents future development and enhancement of specification language. Thus, implementation of more reliable and flexible parser, which will allow to make improvements effortlessly and conveniently is needed.

To do this, theory of formal languages is very useful and helpful, as it studies formal grammars, theory of automata, parsers and compilers. These branches give basic understanding of how formal languages are structured, what they contain and how the process of compilation works. Process of compilation includes parser and lexer, and the theory also studies automatic parser generator, which will help to achieve the goal of the given thesis.

1.1 Goal of the thesis

The goal of the given thesis is to come up with a solution to the problem of improving and extending the CoCoViLa specification language. To achieve this goal, the research in the topic of compiler construction and formal languages is required. The results of this research enable formalisation of the grammar of CoCoViLa specification language. The grammar allows to use existing parser generators in order to automatically construct the parser and implement improvements to language with much less effort, than it is with the language parser, based on regular expressions.

1.2 Outline of the thesis

The thesis is organised as follows. The second chapter presents an overview of theory of formal languages, i.e. description of grammars, tools for recognising context-free languages. Third chapter includes a comparison of parser generators conforming to certain requirements and justifies the choice made for implementing a parser. Fourth chapter describes the process of implementation of a new parser for CoCoViLa. Comparison of results with previous implementations is presented in fifth chapter. The final, sixth chapter, includes conclusion and discussion about the possible future developments in CoCoViLa.

2. THEORETICAL OVERVIEW

2.1 Grammars of programming languages

Grammar is a fundamental formalism used to depict programs' structure. Although the grammar describes only syntactic structure, the grammar can be also viewed as an instrument for defining semantics, as language's semantics can be described in the terms of syntax. Grammar is also fundamental when speaking of language specification, as it dictates a structure on the linear sequence of tokens that have to be used in order to create a parser, and it helps program developers in compiling syntactically legitimate programs and to give detailed syntax-related answers [2].

Grammar is the set of rules that explains how words are used in a language and it consists of a set of production rules, terminals, nonterminals and a start symbol. Production rules have left-hand side and right-hand side, separated by a rightwards arrow, e.g. $A \to \alpha$. Each side contains grammar symbols: terminals and/or nonterminals. Productions specify which grammar symbols may be substituted by other symbols. To generate, or parse a string that belongs to the language defined by the grammar, production rules are subsequently applied until no nonterminal symbols are left, this is called the derivation of a string. Terminal symbols cannot be changed during the application production rules. Nonterminals, on the contrary, are like variables, during the derivation of strings they are being substituted by sequences of nonterminal and terminal symbols.

In order to distinguish grammar symbol types from each other, they must have different alphabet and style. By convention,

1. terminal symbols are written in lowercase letters and are usually taken from the end of the English alphabet, e.g. x, y, z;
2. nonterminals are written using capital letter from the beginning of the English alphabet, e.g. A, B, C;
3. for start symbol, letter S is used;
4. symbol sequences are presented by Greek alphabet, e.g. α, β, γ;
5. for empty sequence, Greek letter ϵ is used.

2.2 Chomsky hierarchy

In the late 1950s [3] Noam Chomsky defined a containment hierarchy of classes of formal languages and grammars that enables for an information technology developer to achieve systematical linguistic goals, e.g. to create and develop a new programming language and/or to make an existing language more specific. As a language may be created with the use of different grammars, the grammars are divided into different types.

According to the Chomsky's classification there are four types of grammars: Type-0 or the unrestricted grammars (also called recursively enumerable), Type-1 or the context-sensitive grammars, Type-2 or the context-free grammars, and Type-3 or the regular grammars. Formal grammar, included in the hierarchy, is formed by a set of production rules, consisting of a start symbol, finite set of terminal symbols and nonterminal symbols (syntactic variables).

If it is possible to develop and write a language, with the use of different types of grammars, then the language is placed in a class of a simpler grammar. In Chomsky classification Type-0 is seen as the most difficult and Type-3 is considered to be the simplest. Considering the fact that according to the Chomsky's hierarchy there are four types of grammars, and also the fact, that if a system works on a precise level, then, generally, the system will also work on a lower rank, then it is possible to constatate, that if a system is created using Type-3 grammar, then it is possible to create the same system, using Type-2 grammar; if a system is created using Type-2 grammar, then it is possible to create the same system, using Type-1 grammar; if a system is created using Type-1 grammar, then it is possible to create the same system, using Type-0 grammar. If there is no lower level, then it means that the system is presented using only one type of grammar. The converse is not true. That is, it is not possible to promote a language, having a grammar of a particular type, to a type, higher in the hierarchy.

2.2.1 Unrestricted grammars

According to the Chomsky hierarchy, the first class of complexity is called Type-0 (also called the recursively enumerable) and to produce Turing-acceptable languages and initially it unifies other three types of formal grammar. This is the most complex type of grammar, and therefore at this moment this class is interesting only from the point of view of science.

As one can see from the name of the type, the complexity class has recursively enumerable sets. It means that left hand side of the grammar rule and the right hand side of the grammar rule may consist of any string. The only restriction is that the left side must contain nonterminals and cannot consist of only terminals. During the process the string may be shortened, it is not compulsory for a string to stay the same length or to get longer.

2.2.2 Context-sensitive grammars

The second type of grammar in Chomsky hierarchy is named Type-1 and it describes and produces context-sensitive languages, wich is acceptable by a nondeterministic (restricted) Turing machine. Principle of the type is to use information about context before substitution is allowed, and working strings are not shortened in the process of production.

Rule of context sensitive grammar is depicted as $\alpha A\beta \to \alpha\gamma\beta$, where:

1. A contains a set of terminal and/or (not less than one) nonterminal characters and cannot not empty;
2. α and β contains a set of terminal and/or nonterminal characters and may be left empty .
3. γ contains terminal and nonterminal characters and cannot be empty.

Context of nonterminal is very important when applying context-sensitive grammar's production rules.

The context-sensitive grammar is to some degree a structural model for human languages. Although, according to Cohen,"there is no mathematical proof of this fact, there are mathematical proofs of the fact that context sensitive languages are recursive" [3]. As it had been said before ("if a system is created using Type-1 grammar, then it is possible to create the same system, using Type-0 grammar"), not all of the context-sensitive languages are recursive. They might be, but it is not a rule.

2.2.3 Context-free grammars

The third type in Chomsky hierarchy is classified as Type-2, and it concerns production of context-free languages, using context-free grammar, and usually it is used to describe and analyse programming language's syntax and to produce languages, recognisable by pushdown automaton. Context of nonterminal is not important when applying context free grammar's production rules. Most programming languages are based on the Type-2 grammar [2].

A context-free grammar G is defined by the 4-tuple: $G = (V, \Sigma, R, S)$ where:

1. V – a finite set, consisting of variables (nonterminal symbols), which are sometimes called syntactic categories;
2. Σ – a finite set of terminals, disjoint from V;

3. R – a finite set of production rules of the grammar;

4. $S \in V$ – a start symbol.

All production rules in context-free grammars should be in the form of $A \to \beta$, where:

1. $A \in V$;

2. $\beta \in (V \cup \Sigma)^*$.

Strings are derived [4], that is, placed in sequence of grammar rule applications that transform the start symbol into the string. In other words, it is the replacement of an occurrence of the left hand side by the right hand side of the production. If leftmost nonterminal is replaced, then it is called canonical derivation. The derivation proves that the string belongs to the grammar's language. During construction of a context-free language, it is suggested to break construction into several simple pieces, as it is easier to solve several little problems instead of one arduous problem. If there are too many pieces, then the structure may become very complex and therefore indirect recursion may appear. It is difficult to notice this kind of recursion, but it may lead to infinity loop.

Context-free languages are recognised by pushdown automata. A special case of context-free languages are deterministic context-free languages, accepted by deterministic pushdown automata [5].

Grammar notation techniques

There are two main types of notation technique [6] for context-free grammar:

1. van Wijngaarden form;

2. Backus–Naur Form.

The first type, van Wijngaarden form, is a technique used to define in a finite amount of rules context-free grammars, which are potentially infinite, and context-free grammars restrict functions of meta-rules.

The second type of context-free grammars notation techniques is Backus-Naur form. It is a formal syntax describing system, where syntax categories are sequentially determined via other categories. This form is used in order to describe formal context-free grammars, and to describe programming languages, document formats, instruction sets and communication protocols.

The latter notation has a subtype – extended Backus-Naur form. The extended form has the advantage, as it is possible to describe simple repetitional constructions of indefinite length (e.g. lists, lines, sequences) without the use of recursive rules. Another advantage of the extended version is that terminals are strictly enclosed within quotation marks and it is possible to omit chevrons for nonterminals (it uses only three types of brackets, vertical line, equals sign, quotation marks, comma), whilst simple form does not use quotation marks around terminals, and it uses chevrons and other symbols for itself and therefore the symbols are not used in the input language and a special symbol is required in order to represent an empty string.

Next, one can see two examples of definition the production rule with BNF notation:

```
<varDecl> ::= <var> <varList>
<varList> ::= <varName> | <varName> <coma> <varList>
```

EBNF notation:

```
varList = var varName {coma varList}
```

2.2.4 Regular grammars

The last type in Chomsky hierarchy - Type-3 - contains one of the most simplest types of formal grammar, which is called regular grammar. Outcoming regular language may be created with context-free grammar, if the left hand side of the rule contains nonterminal and the right hand side of the rule contains exactly one nonterminal and terminals (terminals are not required on the right hand side of the rule), and the given theorem was proven by Chomsky and Miller in the late 1950's [3].

Regular grammar may be divided into leftstring and rightstring grammar, and they are equivalent. Regular languages are recognised by finite automata [7].

2.3 Tools for recognising, analysing and translating formal languages

In order to execute actions, described in source code of a program, the source code should be recognised, analysed and translated. All these processes are in general performed with a tool called compiler.

Process of recognition is the process of combining letters from the program source code into words (tokens). This process stands for lexical analysis, and it is performed by lexer. Process of analysing is process of combining tokens into a tree data structure. This process is performed by syntactic analyser (parser). In the translation phase (performed by semantic analyser) the tree structure is translated to another formal language. This could be a high-level programming language or a machine code.

In next subsections the author will introduce lexer and parser.

2.3.1 Lexer

In order to recognise the languages, first, a lexical analysis needs to take place. Lexer is a program that performs such analysis by accepting sequences of symbols as input. It relies on existing grammar of a language in order to produce a sequence of tokens (meaningful character strings) that put chunks of input data into certain significant groups. The given process is called tokenising.

Lexer works in two phases:

1. scanning – lexer is usually implemented as a finite-state automaton, defined by regular expressions, and in this phase data is coded as a sequence of tokens;

2. evaluating – lexer evaluates the input string in order to produce meaningful tokens.

Every token can be presented as a structure, containing token's identification. Tokens might be numbers, identifiers, parenthesis, brackets etc.

Process of tokenising is needed to prepare input sequence and verify, if each and every word that is being processed exists in the language. Lexical analysis, performed by the lexer simplifies the implementation of a parser, as the latter does not need to determine lexical details of the input sentence.

2.3.2 Parser

After the phase of lexical analysis is finished, the next step is started – *parsing*. The process of parsing is process of syntactic analysis.

Parser is a program that builds a representative data structure of the received input. It uses tokens, obtained during the process of tokenising, as input stream.

Parser's responsibility is to check, if the neighbouring tokens are compatible and if the tokens form allowable expressions. Parser is required to:

1. inform clearly and precisely about presence of errors;

2. provide fast recovery after discovering an error to continue the search for other errors;

3. not to slow down processing of correct incoming string in case errors are found.

During the process of finding errors and discarding them, the parser arranges processed tokens into constituents, resulting in a parse tree, which shows information containing syntactic relations of tokens to each other.

Depending on parser's type and implementation, it may have components as follows:

1. an input buffer (contains input string);
2. a stack (storage of terminals and nonterminals);
3. a parsing table (informs on the production rule that has to be used).

Parser also needs to determine, which production rule it has to use. In order to decide that, parser has to look a particular quantity of tokens ahead. This procedure is called *lookahead*. The number of tokens that parser can use for lookahead is presented in brackets after the name of the parser. However, sometimes there are no numbers in the brackets, and instead of them there is an asterisk. The asterisk means that quantity of lookahead tokens is not limited.

There are two types of parsers: *top-down parser* and *bottom-up parser*.

Top-down parser

Parser checks for possibilities for leftmost derivations by taking a target word, which means that it works from front to the end. It means that process first visits current node and then visits its children. Next the parser searches for sequence of productions that generate the target word, and it is organised by building a tree of all of the possibilities. When and if it becomes clear that the building is not possible any further, as the target word does not appear on the branch of the tree, then the process of building stops. A branch of a tree is suspended [3], if at least one of these conditions is found:

1. bad substring;
2. good substrings, but too many of them;
3. good substrings, but wrong order;
4. improper outer-terminal substring;
5. excess projected length;
6. wrong target word.

In top-down parser there is a procedure, called backtracking, which is the method of tree search. With this method it is not needed to grow all branches instantly, as one branch may be pursued downwards until the moment, when parser either finds the needed word, or terminates the branch.

Top-down parsers may be divided into:

1. recursive descent parser;
2. LL parser;
3. ALL parser;
4. Earley parser.

First type, the recursive descent parser, is usually implemented manually. It is built from a set of mutually recursive procedures, every procedure implements one of the grammar's production rules and therefore the result is that program's structure closely mirrors grammar it recognises. A type of recursive descent parser is predictive parser, and it does not need backtracking.

The second type, LL(k) parser, is a table-based parser. The acronym means that the parser parses the input from left to right and builds leftmost derivation of a sentence. This parser is constructed for LL(k) grammars and in order for it to decide, which type of production rule should be used, it reads the next-available symbol from the input stream and the top-most symbol from the stack.

The third type is ALL parser, and the acronym stands for adaptive LL parser. It is constructed for non-left-recursive context-free grammars and it does not need static grammar analysis [8]. It runs subparsers for each production at a decision point; the subparser that predicts a production at the minimum lookaheads is identified by prediction mechanism, other subparsers are suspended. In order for the subparsers not to grow exponentially, a so-called graph-structured stack is used [9].

The fourth type, Earley parser, is a chart-based parser (chart is a special structure introduced by Martin Kay [10]). Advantage of this parser is that it can parse all context-free languages and it does not have limits of a particular class of language, and it is especially useful in case left-recursively written production rules. Earley parser has three stages it executes: prediction, scanning, completion.

Bottom-up parser

This type of parser works from the end to the front, that is from the target word itself. It means that process first visits the children of a node and then visits the node itself. All substrings of the working string of terminals and nonterminals that are right halves of productions are found, and then it substitues back to the nonterminal that might have produced working string of terminals and nonterminals. Intermediate strings of this parser never exceed the length of the target word, and as there are no new terminals introduced, then no bad terminals ever appear. The parse tree, created by bottom-up parser, is wide, but not very deep.

Bottom-up parsers may be divided into:

1. precedent parser;
2. BC parser;
3. LR parser;
4. CYK parser;
5. recursive ascent parser.

First type, precedent parser, may be divided into operator-precedence parser and simple precedence parser. Operator-precedence parser is a table-driven productive parser mostly for grammars, where two chronological nonterminals never occur in the right-hand side of any production rule. It is not in common use, despite the fact that it is easy to write it by hand, and that it might be written to consult an operator table at execution time of a program. Simple precedence parser is efficient, if there are no erasing production rules, no useless production rules, if there is only one Wirth–Weber precedence relationship, and if no two distinct productions give the same result [11, 12].

Second type, BC parser, tries the production rules in system, starting from the top, and the first production rule to match is applied, and the process is repeated until no production rule matches. Process of the parser can be described by productions of Robert W. Floyd, that is by production rules for rewriting a marker-containing string, on which the production rules focus. The production has a form $\alpha \triangle \beta \Rightarrow \gamma \triangle \delta$ and the production means that if the marker $\triangle$ is surrounded with α on the left hand side and β on the right hand side, then the construction is replaced by $\gamma \triangle \delta$. Despite the fact that BC parser was once almost completely superseded by LALR parser, BC parser is now again in use due to great properties of error recovery [13].

Third type, LR parser, is a table-based parser. The acronym means that the parser parses the input from left to right and builds rightmost reversed derivation of a sentence. This type of parser produces one precise parse tree without assumptions or backtracking. LR parser can be constructed for a wide spectrum of grammars. It may be divided into types as follows:

1. SLR parser – it calculates lookahead sets of input symbols that appear next through a method of approximation, which is established by the grammar, and ignores particular context of a particular parser state. During the use of *Follow sets* it does not have shift/reduce or reduce/reduce conflicts.

2. LALR parser – it calculates lookahead sets of input symbols that appear next through a method of exploration, considers particular context of a particular parser state. During the use of LALR follow sets it does not have any conflicts.

3. Canonical LR parser – it is constructed for all deterministic context-free languages, as it is based on a deterministic automaton, and is based on static state transition tables. This parser use one lookahead token.

4. GLR parser – it is a table-based parser, and it is constructed for handling nondeterministic languages, which contain more than one leftmost derivation. This parser uses breadth-first search for processing all of the potential interpretations of a given input. Although tables of this parser allow forming multiple transitions, the parser allows shift/reduce and reduce/reduce conflicts.

Fourth type, CYK parser, is constructed for context-free languages in Chomsky normal form. Its worst-case asymptotic complexity is $\Theta(|G|n^3)$ [14]. This parser tests for opportunities to split a current sequence of words in half. It works in two phases: recognition and construction, where in the first phase it builds a table, where it determines which nonterminals derive which substring of a sentence, and the second phase constructs all of the possible derivations of the sentence, using the grammar and the table, constructed in the first phase. It is notable that CYK parser constructs parse trees of elements on the array.

Fifth type, recursive ascent parser, is constructed for the literal implementation of the concept of LR parsing, where every parser's function presents one LR automaton state, and inside of every function multi-branch statement is implemented in order to select correct action relying on a current token that had popped off the input stack. When the popped token is identified, then, relying on the encodable state, action (shift or reduce) is taken. If shift counter decrements to zero, then goto action takes place, but only after multi-branch statement.

2.3.3 Syntax tree and abstract syntax tree

As it was mentioned in the previous section, parser builds a representative data structure, which is represented by a syntax tree. The syntax tree is a scheme that depicts syntactic structure of a written program code (see Figure 2.1). The syntax tree is needed from the point of view of grammar for it to show different sections and the structure of the program. Synonymous wordphrase for syntax tree is parse tree, as syntax tree is retrieved through to the process of parsing.

Simple form of syntax tree may be unsuitable, as it contains a lot of excessive information. In order for the tree to be more precise, parser optimizes the compiled syntax tree, omitting excessive punctuation and delimiters, e.g. brackets, pluses and minuses, semicolons etc., as the structure of a tree is self-explanatory. As a result of this process, abstract syntax tree is formed (see Figure 2.2). Abstract syntax tree, in comparison with simple syntax tree, can modify every element

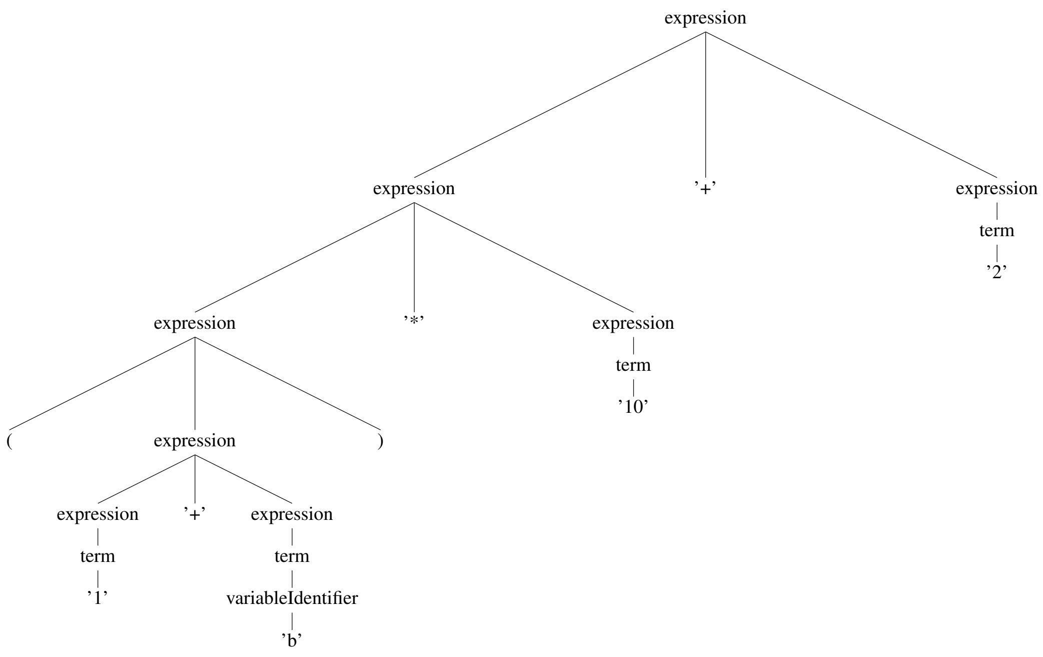
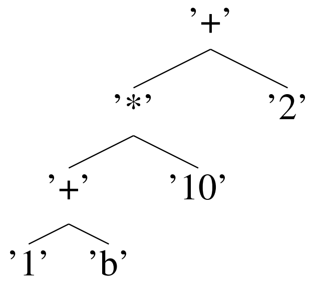


Fig. 2.1: Syntax tree for $(1 + b) * 10 + 2$



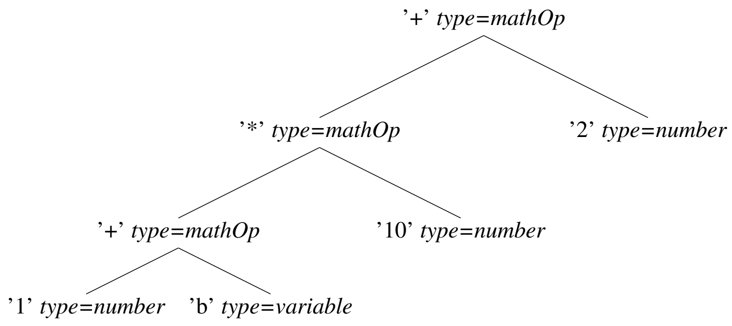


Fig. 2.2: AST for $(1 + b) * 10 + 2$

Fig. 2.3: Annotated AST for $(1 + b) * 10 + 2$

through annotations and properties, which makes compiling much more easier. Abstract syntax tree allows to use annotations in order to add new data, e.g. type data, optimisation data or data on source code, and context handling module finds and inserts the annotations.

As it had been said before, parser constructs abstract syntax tree in the way that there can be added annotations (see Figure 2.3), containing important data [15], and thus there exists a name ”annotated abstract syntax tree”. In order for the annotations to be flexible, that is, there must be a possibility to create, modify and erase them, they have to fit the length of restriction. Ignoring the restrictions may lower the speed of evaluation, and it might not be possible for the parser to erase the unnecessary annotations and whole nodes. Flexibility of annotations is important, as static annotations are rarely used, usually dynamic annotations are needed.

Nodes of annotations consist of tags and values, and every node has an equivalent function in the signature, and subnodes are compatible with the requirements of the node's type.

2.4 Automata theory

Automata theory is a field in theoretical computer science that studies abstract machines and automata. It deals with the study of self-operating virtual machines in order to help to have analytical knowledge of processes of input and output, and it is very important in artificial intelligence, compiler design, formal verification, parsing and theory of computation. Automaton is a mechanism that is relatively self-operating, and it is designed to follow automatically a predetermined sequence of operations or respond to encoded instructions [7, 16].

Automaton is a mathematical abstraction that has only one input, only one output and only one particular state at one particular moment. Formally automaton can be described as a 5-tuple $A = (S, X, Y, \delta, \lambda)$, where:

1. S – a finite set of automaton's states;
2. X – an input alphabet;
3. Y – an output alphabet;
4. $\delta : S \times X \to S$ – a function of transitions;
5. $\lambda : S \times X \to Y$ – a function of outputs.

Automaton has subtypes as finite-state machine and pushdown automata. If functions of input and output are clearly determined for each pair $(s, x) \in S \times X$

of terminals and nonterminals, then the automaton is called deterministic. If this condition is not determined clearly, then the automaton is called nondeterministic. This categorisation is applied to all of the subtypes of automata.

Finite-state machine is an automaton that has a finite amount of possible states, and it has only one state at a time. The transition from one state to another is triggered by a particular event or condition. It can be presented either as a state diagram or as a state transition table. This automaton recognises regular languages.

Pushdown automaton is a subtype of finite automaton. The difference is that first's memory works as a stack to store states, and the current state of the automaton depends on any previous state [7]. If pushdown automaton is deterministic, then the automaton finishes working when it reaches the finite state, and if it is nondeterministic, then the automaton finishes working when either stack becomes empty or when automaton reaches the finite state. Pushdown automaton recognises any context-free language.

Turing machine is a subtype of finite automaton. Visually it is represented as a strip of tape, divided to cells, left hand side of the strip is limited and the right hand side of the tape is infinite. Automaton proceeds symbols according to a table of production rules, and each production rule gives instructions, whether it is needed to write a new symbol to a particular cell, to perform a transition, or to move one cell left or right. Turing machine is considered to be one of the most powerful automaton, as the infinite quantity of tape gives unlimited amount of storage space. Every part and action of the automaton is finite, discrete and distinguishable. Turing machine recognises unrestricted (recursively enumerable) languages.

3. RELATED WORK

Compiler construction is a very well researched topic. A lot of tools exist for automating this process by enabling generating lexical analysers and parsers from language specifications in the form of grammars.

In this chapter the author presents a comparative analysis of some parser generators. As a result of the analysis, one of the existing parser generators is chosen for further usage.

3.1 Comparison of parser generations

There are a lot of solutions for generating parsers. Taking into account the goals of the present thesis, the following requirements are formulated for including tools in comparison:

1. include lexical parser generator;
2. generate parsers in Java language;
3. use for grammar EBNF notation or improved version of EBNF notation;
4. are still maintained.

Three syntax parser generators correspond to stated criteria: JavaCC, SableCC, ANTLR.

The analysis is constructed as follows. Each selected parser generator is used to implement a language for simple arithmetic calculations. The language allows to use symbols “ $+$ ”, “ $-$ ”, “$/$”, “ $*$ ” for arithmetic operations, symbol “$;$” for separating expressions from each other, and symbol “.” as separator in floating point numbers. Program, written in the language, has to execute arithmetic calculations. Also the program has to consider a correct order of execution of arithmetic operations.

Also there must be an opportunity in source text of a program to set several arithmetic calculations, which will be divided by a semicolon. Spaces, tabulation symbols, newline and carriage return must be ignored in a source text of a program. Results of all of the arithmetic expressions, and also of original expressions, are printed into system standart output, e.g. $12 = 6 + 2 * 3$, for each expression on every line.

3.1.1 JavaCC

JavaCC[1] (acronym for Java Compiler Compiler) is a LL(*) parser generator. It was created by Sriram Sankar and Sreenivasa Viswanadha in 1996, and released by Sun Microsystems. This tool specialises in Java, and therefore the output of the generator is easily converted to Java program, and it has "full support for automatic tree building and visitor pattern". The documentation and the program are freeware.

The first step is creating a grammar of a language, and this grammar has to consist of several blocks. The first block describes tokens that parser should ignore:

```
SKIP : {
    " "
|   "\r"
|   "\n"
|   "\t"
}
```

Second block describes allowed tokens of the language:

```
TOKEN : {
    < NUMBER: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
|   < DIGIT: ["0"-"9"] >
}
```

Here are two tokens defined: $DIGIT$ and $NUMBER$.

And, finally, four production rules:

```
void calculator() : {
    (expression() ";")* <EOF>
}
void expression() : {
    mult() [
        "+" expression()
    |   "-" expression()
    ]
}
void mult() : {
    term()[
```

[1] Home page of JavaCC https://javacc.java.net

```
        "*" mult()
    |   "/" mult()
    ]
}
void term() : {
    "(" expression() ")"
|   token = < NUMBER >
}
```

First production rule *calculator* can be compared to *main* method in Java, it is an entry point to the language's grammar. It defines that program may consist of zero or more *expression*s, followed by a semicolon, and ends with the end of a file. As JavaCC does not allow to use left recursion, it is required that three different production rules are implemented: *expression* for addition or subtraction, *mult* for multiplication or division, and *term* for number or parenthesised expression.

JavaCC provides two opportunities for adding semantics to the grammar. The first opportunity is to add Java code directly to the grammar, but it is very unreliable, as in this case the grammar is bound with implementation, and process of application of different behaviours requires modification of the grammar. And therefore it is not covered in current thesis.

The second opportunity for adding semantics is to use syntax tree and visitor pattern. In order to do that one has to apply an external tool called "JTB"[2]. This tool generates grammar with Java code that builds syntax tree from source grammar, and then it is needed to apply JavaCC on grammar generated by JTB to generate the parser. As a result, a lot of additional classes are created. Most interesting of them are:

- DepthFirstRetArguVisitor
- DepthFirstRetVisitor
- DepthFirstVoidArguVisitor
- DepthFirstVoidVisitor

All classes implement tree visitor pattern and it is possible to extend any of them, and override any method from parent class to add desired behaviour to the program, for example, author of the given thesis extended the *DepthFirstRetArguVisitor* class and overrode four methods:

- *public Double visit(final calculator n, final Object arg)* – is invoked when node defined by *calculator* is visited

[2] Home page of JTB `http://compilers.cs.ucla.edu/jtb/`

- *public Double visit(final expression n, final Object arg)* – is invoked when node defined by $expression$ is visited
- *public Double visit(final mult n, final Object arg)* – is invoked when node defined by $mult$ is visited
- *public Double visit(final term n, final Object arg)* – is invoked when node defined by $term$ is visited

```
public Double visit(final term n, final Object arg) {
   Double nRes = null;
   INode node = n.f0.choice;
   if(node instanceof NodeToken){
      nRes = Double.parseDouble(((NodeToken)
         node).tokenImage);
   }else if(node instanceof NodeSequence){
      //[(, expression, )]
      node = ((NodeSequence) node).elementAt(1);
      if(node instanceof expression){
         nRes = node.accept(this, arg);
      }
   }
   return nRes;
}
```

In the method, presented above, the check is performed to insure, which production rule alternative is selected, and if it is a number, then received number should be converted from string representation to double value, and returned. Deeper tree traversing is not possible. Otherwise, if it is a parenthesised expression, it is required to move deeper and visit $expression$ tree node.

```
public Double visit(final mult n, final Object arg) {
   //[term, * or /, mult]
   Double nRes = ((term)n.f0).accept(this, arg);
   NodeChoice nodeChoice = (NodeChoice)n.f1.node;
   if(nodeChoice != null){
      NodeSequence nodeSequence = (NodeSequence)
         nodeChoice.choice;
      if(nodeSequence.elementAt(0).toString().equals("*")){
         nRes = nRes *
            nodeSequence.elementAt(1).accept(this, arg);
      }else{
         nRes = nRes /
            nodeSequence.elementAt(1).accept(this, arg);
```

```
        }
    }

    return nRes;
}
```

In method, presented above, the check is performed to insure, whether node, that is being visited at the given moment, has only one child node, and in this case the node is $term$, or if it has three children nodes: $term$, operator (* or /) and $mult$. In both cases the $term$ node will be visited to get the number from there. But only in second case the $mult$ node will be visited in order to calculate the result of arithmetic expression, and after the $mult$ node is visited, the value of current node is calculated. Also in both cases the value of current node is returned. Same logic is applied for $expression$ production rule.

And code for last production rule $calculator$:

```
public Double visit(calculator n, Object arg) {
    for (INode node : ((NodeListOptional)n.f0).nodes) {
        //[expression, ;]
        System.out.println(
            ((NodeSequence)node)
                .elementAt(0).accept(this, arg));
    }
    return n.f1.accept(this, arg);
}
```

In the code above there is a loop that visits all child $expression$ nodes and prints their values to standart output. It is important to mention that with JavaCC there is no convenient way to receive textual representation of a node, and therefore it was not possible to print original expression.

Only the last step remains, to launch the parser on a prepared input file:

```
InputStream stream = new FileInputStream("simple.ss");
Calculator s1 = new Calculator(stream);
calculator input = s1.calculator();
input.accept(new DepthFirstRetArguVisitorImpl(), null);
```

3.1.2 SableCC

SableCC[3] is a LALR(1) parser generator for Java [17]. SableCC is a bottom-up parser. Documentation and program are freeware.

[3] Home page of SableCC `http://sablecc.org`

Unlike in JavaCC, in SableCC it is required to define all tokens in lexer's part of a grammar (in JavaCC it was possible to inline tokens in production rules). Also it is not possible to refer to one token from definition of another, but there are tokens of special type (called "Helpers") to support analogous functionality. Rule $Ignored\ Tokens$ defines tokens that should be ignored. Below is a listing of lexer's part of SableCC grammar file:

```
Helpers
   digit = ['0' .. '9'];

Tokens
   t_number = (digit)+ ('.' digit+)?;
   t_plus = '+';
   t_minus = '-';
   t_mult = '*';
   t_div = '/';
   t_l_par = '(';
   t_r_par = ')';
   t_blank = (' ' | 10 | 13 | 9)+;
   t_semic = ';' ;

Ignored Tokens
   t_blank;
```

The definition of production rules is similar to JavaCC:

```
Productions
   calculator =
      statement*;

   statement =
      expression t_semic;

   expression =
      {factor} multiplication |
      {minus}  expression t_minus multiplication |
      {plus}   expression t_plus multiplication;

   multiplication =
      {term} term |
      {div} multiplication t_div term |
      {mult} multiplication t_mult term;

   term =
      {number} t_number |
      {expr}   t_l_par expression t_r_par;
```

But there are two important differences compared to JavaCC:

1. SableCC supports left recursion, but does not allow right recursion;
2. every alternative of a production rule should be labeled (identifier in curly brackets).

SableCC provides only one possibility to add semantics to the grammar. By using tree traversal listeners, in contrary to visitor pattern, this does not allow nor require manually visiting child nodes. All classes required to implement a listener are generated by SableCC, and there is no need to use any external tools.

SableCC generates a lot of classes, but there are two important classes:

- DepthFirstAdapter
- ReversedDepthFirstAdapter

These classes implement parse tree traversal in top-down and bottom-up ways respectively. The developer may extend any of them, and override the listener methods. The naming convention for listener methods is as follows: every method is prefixed by in or out, after that comes letter "A", then comes a label of an alternative, and at the end comes the name of a production rule. Methods prefixed with in are invoked, when tree traveller enters the node, and methods with out prefix are invoked before tree traveller leaves current node (all child nodes are visited). For example, method $outAMultMultiplication$ will be invoked, when all child nodes of $mult$ alternative of $multiplication$ production rule are visited.

As with approach of using tree traversal listener pattern, provided by SableCC, there is no convenient way to control the travelling over the parse tree, and therefore return statements, as in JavaCC, cannot be used. Another algorithm should be used to implement the behaviour of arithmetic calculations' language. One of the possibilities to do it is by using a stack and $DepthFirstAdapter$. The algorithm is as follows: when a tree node, containing a number, is reached, then it is pushed to the stack.

```java
@Override
public void inANumberTerm(ANumberTerm node) {
   numbers.push(
      Double.parseDouble(node.getTNumber().toString()));
}
```

Before leaving the node that represents a particular arithmetic operation, program should pop two numbers from the stack, and apply required operation on them, then the result of the process is pushed back to the stack. Code listing below shows only adding operation, but the similar code is used for other operations, the only difference is that another arithmetic operation is used:

```
@Override
public void outAPlusExpression(APlusExpression node) {
   double left = numbers.pop();
   double right = numbers.pop();
   numbers.push(left+right);
}
```

As traveller reaches the *statement* node, the stack will contain only one value, and it is the result of evaluation of the current expression. And the result, along with original expression, could be printed.

```
@Override
public void outAStatement(AStatement node) {
   System.out.println(
      numbers.pop() + " = "
      + node.getExpression().toString());
   numbers = new Stack<>();
}
```

Only the last step remains, to launch the parser on prepared input file:

```
File inputFile = new File("file.in");
InputStreamReader inputStreamReader =
   new InputStreamReader(new FileInputStream(inputFile));

PushbackReader pushBackReader
   = new PushbackReader(inputStreamReader, 1024);
Lexer lexer = new Lexer(pushBackReader);
Parser p = new Parser(lexer);
Start tree = p.parse();
tree.apply(new CalculatorTreeWalker());
```

3.1.3 ANTLR

ANTLR[4] (acronym for ANother Tool for Language Recognition) is an ALL(*) parser generator [18], and it is used to design automatic lexers, parsers, tree parsers and combined lexer-parsers. Predecessor of ANTLR (PCCTS, acronym for Purdue Compiler Construction Tool Set) was created by Terence John Parr in

[4] Home page of ANTLR `http://www.antlr.org`

1989. At the moment support is targeted more on Java and C#, but older versions also support Ada95, ActionScript, C, Java, JavaScript, Objective-C, Perl, Python, Ruby, and Standard ML. The documentation and program are freeware, and in addition the creator of the program has released paper-print documentation, i.e. The Definitive ANTLR 4 Reference, which concernes the latest version of ANTLR.

Similar to JavaCC, ANTLR allows to inline tokens in production rules, so the only tokens that should be defined for the arithmetic calculations language are tokens that represent the number:

```
NUMBER
    : INTEGER ('.' INTEGER)?
    ;

INTEGER
    : '0'..'9'+
    ;
```

For ignored tokens:

```
WS : [ \t\r\n]+ -> skip ;
```

Unlike SableCC and JavaCC, ANTLR allows using both left and right recursion in production rules. This leads to more clear, compact and convenient way in defining those production rules:

```
calculator
    : (expression ';')+
    ;

expression
    :  NUMBER                            #number
    |  '(' expr = expression ')'         #expr
    |  left = expression '*' right = expression #mult
    |  left = expression '/' right = expression #div
    |  left = expression '-' right = expression #minus
    |  left = expression '+' right = expression #plus
    ;
```

Similar to SableCC, alternatives also could be labeled, but with different syntax, the identifier of a label goes after hash symbol in the same line, where alternative is located, also it is allowed to give names inside of the production rule for subrules, the usage of subrules' name is covered next in current section.

ANTLR provides three possibilities to add semantics to the grammar. First and second is similar to JavaCC: by adding Java code to the grammar and by implementing the tree visitor. Third is similar to in SableCC method, by using tree traversal listeners. As using Java code in the grammar is a bad approach (the reason of that is described in section 3.1.1), and implementation of tree traveler listener is very similar to SableCC, the usage of the tree visitor is demonstrated below.

ANTLR generates the $CalculatorBaseVisitor$ class. By extending this class, and by overriding its methods, it is possible to add semantics to the language.

When visiting a tree node, represented by $number$ alternative of production rule $expression$, the following code will be invoked:

```
@Override
public Double visitNumber(NumberContext ctx) {
   return Double.parseDouble(ctx.NUMBER().getText());
}
```

In the code above the textual value of a node is converted to a number, and returned to the parent node.

In $expr$ node, the program will get an expression from parentheses, and visit its tree node:

```
@Override
public Double visitExpr(ExprContext ctx) {
   return visit(ctx.expr);
}
```

Also the visit method for each alternative, where arithmetic happens, alternatives $mult$, div, $minus$ and $plus$ should be implemented. For brevity, only one method will be shown below (others are very similar):

```
@Override
public Double visitPlus(PlusContext ctx) {
   double left = visit(ctx.left);
   double right = visit(ctx.right);
   return left + right;
}
```

It is notable how the value of left and right hand expressions are received, it is done by using their labels from production rule definition $left$ and $right$.

The code of method that is invoked by visiting the top tree node is as follows:

```
@Override
public Double visitCalculator(CalculatorContext ctx) {
   for (ExpressionContext expressionContext :
      ctx.expression()) {
      System.out.println(
            visit(expressionContext) + " = " +
                expressionContext.getText());
    }
    return 0.0;
}
```

In the code above, the behaviour is similar to the one that was in JavaCC. The program iterates over all child nodes of *calculator* node and visits them.

Only the last step remains, to launch the parser on prepared input file:

```
CharStream input = new ANTLRFileStream("simple.ss");
CalculatorLexer lexer = new CalculatorLexer(input );
TokenStream token = new CommonTokenStream(lexer);
CalculatorParser parser = new CalculatorParser(token);
CalculatorContext calculatorContext = parser.calculator();

CalculatorBaseVisitor<Double> visitor = new
    CalculatorBaseVisitorImpl();

visitor.visit(calculatorContext);

ParseTreeWalker walker = new ParseTreeWalker();
walker.walk(new CalculatorBaseListener(),
    calculatorContext);
```

3.2 Selection of parser generation

To conclude the comparison of parser generators, the justification of the selection of the tool for implementing stated goals of the thesis is presented in this section.

JavaCC is not the best solution, as it has very poor documentation, and therefore it is very difficult to develop a language parser, using this tool. Also, in order to create a parse tree, it is needed to use a third party tool. It is noticeable that this method of creating parse trees is a workaround, as parse trees are built with the help of Java code, which is not written by hand, but still it is added to grammar. Also, labeling of alternatives is missing, and this circumstance complicates the development, as one has to find the needed alternative by hand, and also one has to guess, in which position in the array it is located. It means that even any minor change in the grammar will require for the review of whole written code in order to check, whether the selection of alternative in changed production rule is still valid. These factors are the main reasons for the author to exclude this program.

In comparison with JavaCC, SableCC has better documentation, but it still causes problems for a developer during the process of development. Despite the fact that SableCC provides labeling of alternatives and internal tool for creating syntax trees, it leaves an impression that the product is still in a raw stage of development. Especially it is expressed by the lack of integration with IDE. Alike JavaCC, SableCC, unfortunately, does not meet the requirements, and therefore it is not chosen.

ANTLR, in comparison with other above-described programs, has very good documentation. It is documented in a book "The Definitive ANTLR 4 Reference" [18], which is written by the creator of the program. The book is not only documentation to ANTLR, but it also contains a lot of useful information about and for the development of languages. Also, a great distinction from JavaCC and SableCC is that ANTLR provides possibility of using simultaneously left recursion and right recursion in production rules. This program has very good integration with IDE, and it also has an integrated tool for visual representation of a parse tree, which makes the process of development of a grammar much simpler, and it also makes the process of searching for the errors in grammar graphical, fast and simple. ANTLR met all of the requirements, needed for developing a language, and for this reason the author of the given thesis chose ANTLR as a tool for developing a parser for CoCoViLa specification language.

4. IMPLEMENTATION

CoCoViLa is a model-based framework for developing and using visual domain-specific languages (VDSLs). Textual models underlying diagrams and visual objects are implemented in a declarative language called *the specification language*.

Until the process of present thesis' work had started, CoCoViLa had a hand-written specification language parser that used regular expression in order to process statements of the specification language. Regular expressions are not flexible and it is not easy to understand and change them, it was difficult to maintain the parser and introduce new syntactic constructions to the language. The main goal of this work was to create a possibility for the specification to be parsed in a much more flexible way, using automatically generated parser from a given formal language grammar specification. ANTLR was chosen as a suitable parser generator for this task.

CoCoViLa enables to specify computational problems, either visually or textually, using the specification language. Specifications are added to Java classes. Classes, annotated with such specifications, in CoCoViLa context are called *meta-classes* and their corresponding specifications are *metainterfaces*. The following example of specification will be used to demonstrate various statements of the specification language:

```
specification ExampleSpec {
   int a,b,c,d,x,y,z,aliasElem;      //variable declaration
   alias (int) ali = (a,b);          //alias declaration
   a = 1;                            //equation
   b = 5+a;                          //equation
   c = d/z + y;                      //equation
   [x->d],z,ali->c{someJavaMethod}; //axiom
   x = 2;                            //equation
   z = 11;                           //equation
   d = x+10;                         //equation
   aliasElem = ali.0;                //equation
}
```

Every declaration of specification should start with a keyword $specification$, it should be followed by the name of the specification, in this case $ExampleSpec$, and the code of the specification is enclosed in curly brackets. In code example above first line of the code of specification contains declaration of variables. In this case there are several variables of type integer with names declared: a, b, c, d, x, y, z and $aliasElem$. A regular expression used to parse such statement in CoCoViLa was as follows:

```
^ *(static)? *([a-zA-Z_][0-9a-zA-Z_]*
(([\\.][a-zA-Z_][0-9a-zA-Z_]*)*)[0-9a-zA-Z_$]*
(\\[\\])*) (([a-zA-Z_$][0-9a-zA-Z_$]* ?, ?)*
?[a-zA-Z_$][0-9a-zA-Z_$]* ?$)
```

Also the code of specification includes a declaration of an alias (second line of specification). Alias in CoCoViLa specification language is a special data structure that may be compared with a tuple. Syntax of declaration of an alias is as follows: first is the keyword $alias$, then in the parentheses comes type of variables, which will be contained in alias (a type with parentheses may not be present). In our case type of alias is integer. Next is the name of alias, after that equals sign, and then in parentheses, separated with a comma, come names of variables, which are contained in an alias. Access to elements inside alias, according to the syntax, is as follows: on the left hand side of the dot, contains the name of alias, and on the right hand side of the dot contains index of the element inside an alias (starting from zero). Example of this is on the last line of example specification. Next is presented a regular expression, used in CoCoViLa, which is needed for handling an alias declaration:

```
alias *(\\(( *[^\\(\\) ]+ *)\\))*
*([^= ]+) *= *\\((.*)\\) *
```

In CoCoViLa specification language there does not exist such thing as assigning the value to a variable. Instead of that there exist equations (in the above-presented example of the code all lines that set equations have corresponding comment $equation$). Equations set to the CoCoViLa language methods that can be used by the program in order to calculate value of one or another variable from other variables, or, in other words, set arithmetic dependency between variables. In the example above CoCoViLa finds a way on how to compute a variable y. It is known that variable d can be calculated from variable x, and value of variable x is known, and it equals 2, and value of variable z is also known, and it equals 11. Also program knows value of a variable c, which can be calculated with the help of an axiom (meaning of an axiom will be described later). The following is a regular expression used to parse equations:

```
 *([^=]+) *= *([-_0-9a-zA-Z.()\\+\\*/^ ]+) *$
```

Function of axiom is similar to function of equation, but unlike equation, axiom sets not arithmetic dependency, but functional dependency between variables. Syntax of axiom is as follows: a possibly empty list of input variables followed by $- >$ symbol, separating the list of axiom's output. After output, in the curly brackets, a name of a Java method is specified, and it implements this functional dependency. Inputs of axiom can be either variables or subtasks. Subtasks (in the list of inputs set in square brackets) stand for function arguments, and CoCoViLa synthesizes it's implementation automatically. This axiom, in the example above, can be read as follows: variable c can be calculated by calling a Java method $someJavaMethod$ from the underlying Java class passing a function that calculates d from x and variables z and ali as arguments. Next, a regular expression for parsing axioms is presented:

```
(.*) *-> *(.+) *\\{(.+)\\}
```

All of the presented examples of regular expressions, which were used in CoCoViLa parser, are difficult to understand and to maintain, if process of maintenance is needed.

Further, the author presents a detailed architectural overview of the realisation of a new parser, and explains, how the new parser is integrated into the CoCoViLa system, and what is improved in comparison with the old hand-written parser.

4.1 Architectural overview

In the upcoming section of the thesis, the author presents a high-level architectural description of implementation of a new parser, which was generated using ANTLR. The purpose of created classes and their interaction with existing CoCoViLa code are described.

4.1.1 Interaction with ANTLR

The Class diagram (see Figure 4.1) represents interaction between the classes, created by the author, with the classes, generated and provided by ANTLR. Three colours are used in the scheme:

1. green – classes, created by the author;
2. blue – classes, which already existed in the library of ANTLR;
3. yellow – classes, generated by ANTLR.

Next, the author will describe the functions of every class depicted in the scheme.

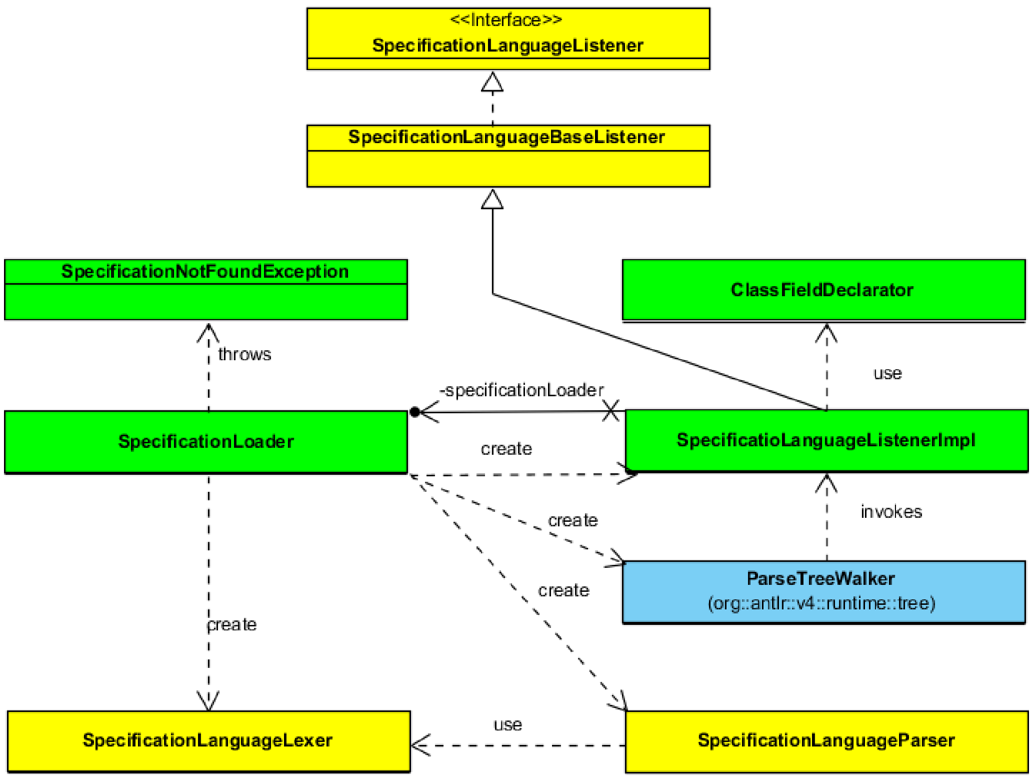

<<Interface>>
SpecificationLanguageListener
SpecificationLanguageBaseListener
SpecificationNotFoundException
ClassFieldDeclarator
throws
use
-specificationLoader
SpecificationLoader
create
SpecificatioLanguageListenerImpl
invokes
create
ParseTreeWalker
(org::antlr::v4::runtime::tree)
create
create
use
SpecificationLanguageLexer
SpecificationLanguageParser

Fig. 4.1: ANTLR classes relation

SpecificationLanguageListener is an interface, which is needed in order to implement tree traversal pattern. All of the methods, needed for the implementation of the pattern, are located in this interface. For every production rule there are two methods, and every method has a prefix - either it is $enter$, or it is $exit$. The methods are invoked respectively, when tree traversal visits a tree node, which corresponds to a production rule, or when tree traversal finishes the process of visiting a tree node (after all children tree nodes of current tree node are visited).

As during the process of implementation of an interface in Java language it is needed to override all of the methods of the interface, and in the generated class that is meant to create a tree traversal listener are too many methods, which is very uncomfortable for a developer, then, in order to ease developer's work, ANTLR generates one additional (adapter) class, which is called SpecificationLanguage-BaseListener. This class implements an interface, called SpecificationLanguage-Listener, and overrides all of the methods, which are defined in the interface, by giving them empty implementation. Due to this advantage, which is presented by ANTLR, developer can extend the class of SpecificationLanguageBaseListener instead of implementing SpecificationLanguageListener interface, in order to create one's own listener, and therefore developer can just simply override only needed methods.

As a result of the process, described above, a class called SpecificationLanguageListenerImpl is created. This class contains whole logic, which is needed for processing the specification language, and the result of the author's work is the implementation of a pattern, called tree traversal listener.

The next class – $ClassFieldDeclarator$ – is a helper-class, which is used in $SpecificationLanguageListenerImpl$, and one needs it for creating and managing context of variables.

The class, which can be though of as an entering point into a parser, is called SpecificationLoader. This class may be compared with a class from Java API, called *ClassLoader*. It can either accept a source code of a specification, presented as a string, as a whole, and also the name of the specification, or as a path to the file, where the specification is kept. In this case, if no file is found in the assigned path, or if there is no description of the specification given, then an exception, called *SpecificationNotFoundException*, is thrown. *SpecificationLoader* also stores information on all of the specifications, which are processed during the life cycle of the given class, and, if it is necessary, then one may obtain information from it by the name of processed specification. The class is also responsible for communication with ANTLR's library. *SpecificationLoader* initialises classes that are described next:

1. SpecificationLanguageLexer – lexical analyser for specification language; this class prepares tokens for next class;
2. SpecificationLanguageParser – class that receives a stream of tokens and does the actual parsing;
3. ParseTreeWalker – the class is responsible for traversing the syntax tree; it invokes methods of the listener (see next point);
4. SpecificationLanguageListenerImpl – an implementation of a tree traversal listener.

4.1.2 Integration with CoCoViLa

Previously, the author of the given thesis presented an architectural overview of Class diagram of ANTLR classes, in this section the author will describe another Class diagram - CoCoViLa classes (see Figure 4.2). In this depicted scheme one may see how the author-created classes interact with CoCoViLa/ANTLR classes.

Similarly to the first depiction of diagram, the author used in this diagram colours as well: blue and green. Classes, coloured in blue, represent classes that

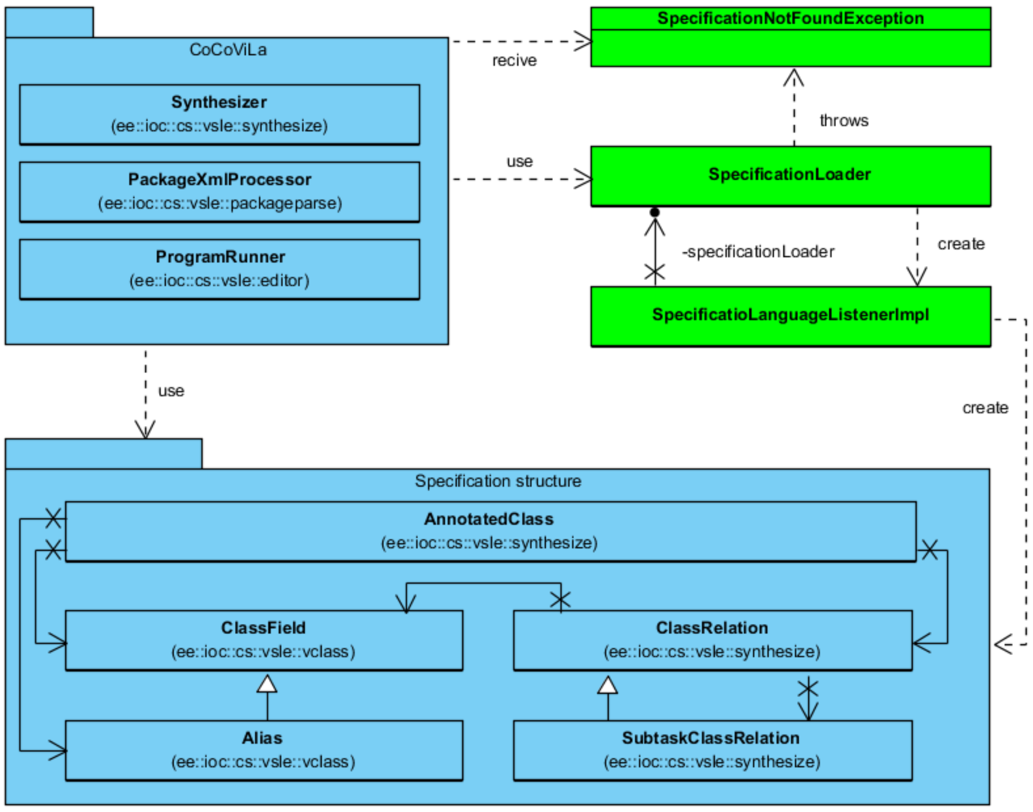

CoCoViLa
Synthesizer
(ee::ioc::cs::vsle::synthesize)
PackageXmlProcessor
(ee::ioc::cs::vsle::packageparse)
ProgramRunner
(ee::ioc::cs::vsle::editor)
recive
use
SpecificationNotFoundException
throws
SpecificationLoader
-specificationLoader
create
SpecificatioLanguageListenerImpl
use
create
Specification structure
AnnotatedClass
(ee::ioc::cs::vsle::synthesize)
ClassField
(ee::ioc::cs::vsle::vclass)
ClassRelation
(ee::ioc::cs::vsle::synthesize)
Alias
(ee::ioc::cs::vsle::vclass)
SubtaskClassRelation
(ee::ioc::cs::vsle::synthesize)

Fig. 4.2: CoCoViLa classes relation

existed in CoCoViLa; classes, coloured in green, represent classes created by the author for the new parser.

The box named CoCoViLa contains three classes:

1. Synthesizer;
2. PackageXmlProcessor;
3. ProgramRunner.

These three classes are needed for the interaction between a user and CoCoViLa. During the time, when the program runs, these classes may invoke Specification loader in order to receive inner data structure, obtained from the source code of specification.

The data structure, needed for description of specification, is represented by AnnotatedClass, which is the main container-class for storing structure of specification. It stores types of data, which are described by next classes:

1. ClassField - class, needed for storing information on declared variables;

2. Alias - class that extends ClassField, and it is needed for storing information on declared aliases;

3. ClassRelation - one of the key classes of CoCoViLa, which is used for setting arithmetic dependencies between variables that can be set with the help of equation or for setting functional dependencies between variables that can be set with the help of an axiom; it stores the following information:

   (a) from which variables (called $inputs$), declared in specification, could be calculated other variables (called $outputs$);

   (b) which operations have to be executed with inputs in order to receive output;

4. SubtaskClassRelation - it is needed for setting subtasks inside of an axiom.

4.2 Grammar implementation

The author of the given thesis had two main problems, which had to be solved:

1. create new grammar for the specification language that would completely reflect the old functionality of the specification language;

2. add to the specification language new syntactic opportunities (e.g. one of the new opportunities that was added is initiation of a variable in the same line, where it was initially declared).

Before the two main problems, which were depicted above, were solved, it was utterly difficult for a developer to manage the code in a way a developer can manage it now, e.g. in the situation, described in the brackets on the second position of the enumeration, by using regular expressions. It can be depicted in a way that if one would take an example of a regular expression (shown in the beginning of the Implementation section), and would add a possibility for initiation of a variable in the same line, where it was initially declared, then the graphic representation of the regular expression would become very difficult to understand. It means that it would become incomprehensible to the developer of the code and more incomprehensible to another developer, who will overtake the code in order to continue developing it, using regular expressions.

Next the author explains how these two problems were solved, what was used, and also explains, how it works.

4.2.1 Porting of existing functionality

In order to create new grammar for language specification parser generator, the author took the grammar, which was not complete and was not being used, a large number of programs, written in specification language, and also had a detailed study of source code of the given parser and of regular expressions that were used in it. As a result the new grammar was created, and this grammar will be described in this section. In addition to the description of grammar, the author will describe actions, which are performed in class $SpecificationLanguageListenerImpl$, which, as it had been described in the previous section, is the listener for the $ParseTreeWalker$ class that traverses parse tree, which was created from the source code of specification with the help of the grammar.

Despite the fact that although ANTLR allows to inline tokens, it would be more comfortable, if some tokens were defined for the lexer. Next is presented the part of grammar that defines tokens for the lexer.

```
NUMBER
    : INTEGER ('.' INTEGER)?
    ;

INTEGER
    : '0'..'9'+
    ;

STRING : '"' (ESC|.)*? '"' ;
fragment ESC : '\\"' | '\\\\' ;

IDENTIFIER
    :  LETTER LETTER_OR_DNUMBER*
    ;
fragment LETTER : [a-zA-Z$_];
fragment LETTER_OR_DNUMBER : [a-zA-Z0-9$_];

ALIAS_ELEMENT_REF
    :  '.' NUMBER+
    ;
//IGNORED TOKENS
WS : [ \t\r\n]+ -> skip ;
COMMENT : '//' .*? '\r'? '\n' -> skip;
BLOCK_COMMENT : '/*' .*? '*/' -> skip;

JAVA_BEFORE_SPEC : .*? '/*@' -> skip;
JAVA_AFTER_SPEC : '@*/' .*? -> skip;
```

Tokens INTEGER and NUMBER are needed for defining numbers with floating point. Token STRING defines how a string can be defined in the language. It can be any text between double quotes, and a helping token ESC is needed so that inside a string could be written a double quotes symbol, but before that it is needed to escape it with the help of reverse solidus (\). Also is defined the token $IDENTIFIER$. After the comment $IGNORED\ \ TOKENS$ there are defined symbols that have to be ignored by the lexer. These symbols are spaces, tabulation symbols, newlines, carriage return, and also comments.

After all of the tokens are defined, basic production rules have to be described. These production rules will be used very often in further definition of the grammar. Next, one can see corresponding code:

```
type
    :   (classType | primitiveType) ('[' ']')*
    ;

classType
    :  IDENTIFIER ('.' IDENTIFIER )*
    ;

primitiveType
    :  'boolean'
    |  'char'
    |  'byte'
    |  'short'
    |  'int'
    |  'long'
    |  'float'
    |  'double'
    ;

variableIdentifier
    :  IDENTIFIER ('.' IDENTIFIER )* ALIAS_ELEMENT_REF*
       ('.' variableIdentifier)?
    ;

variableIdentifierList
    :  variableIdentifier (',' variableIdentifier )*
    ;
```

Next two presented production rules are needed in order to describe declarations of class of specification. Production rule, named *metaInterface*, determines the structure of specification's declaration, after the keyword *specification* is set the name of the specification, and after the name of specification's class it is possible to set a name of the class that contains basic specification. In order to describe links to basic specification, a production rule, named *superMetaInterface*, is used. Next, one can see an example:

```
metaInterface
    :  'specification' IDENTIFIER (superMetaInterface)?
       '{' specification '}'
    ;
superMetaInterface
    :  'super' classType (',' classType)*
    ;
```

At that moment, when ParseTreeWalker invokes *SpecificationLanguageListenerImpl*, it will create a new instance of *AnnotatedClass*'s class, and will save it in the class variable, also to the *AnnotatedClass* will be added information about basic specification.

Next, a production rule, called $specification$, is specified. It defines that specification can consist from zero or more statements, and every statement ends with semicolon.

```
specification
    :  ( statement ';' )*
    ;
```

The production rule for $statement$ consists of following cases:

```
statement
    : variableDeclaration | constantDeclaration |
       variableAssignment
    | axiom | goal | aliasDeclaration | aliasDefinition |
       equation
    ;
```

Next are presented the production rules to which the production rule $statements$ refers to.

Declaration of variables and constants is handled by the following production rules:

```
variableDeclaration
    :  'static'? type IDENTIFIER (',' IDENTIFIER)*
    ;
constantDeclaration
    :  'const' type IDENTIFIER '=' expression
    ;
```

Variable can be declared as follows: at first, type of a variable has to be written, then, separated by a comma, comes name of these variables. Also, before the type of a variable, a keyword $static$ may stand. One constant can be defined, and to do that, before it's type has to be written keyword $CONST$, and after the type should come the name for that constant, after that equals sign, and then the value itself. When a node (corresponding to one of two above-listed production rules) of the parse tree is visited, then $SpecificationLanguageListenerImpl$ creates a new instance of $ClassField$ class, and adds it to the instance of $AnnotatedClass$, which is previously saved as class variable.

Although it had been said before that there is no such thing as assigning a value to a variable in a specification language, and instead of that equations are used, not all of data types allow the use of arithmetic operations. In order to make possible the use of those data types in specification language, next production rules are used:

```
variableAssignment
    :  variableIdentifier '=' variableAssigner
    ;

variableAssigner
    :  array
    |  'new' classType '(' expression (',' expression)* ')'
    |  STRING
    |  'true'
    |  'false'
    ;

array
    :  '{' (inArrayVariableAssigner (','
       inArrayVariableAssigner)* )? '}'
    ;

inArrayVariableAssigner
    :  variableAssigner | variableInitializer
    ;

variableInitializer
    :  array
    |  expression
    ;
```

Production rule *variableAssignment* determines that on the left hand side of equals sign the identifier of a variable can be written (defined by $variableIdentifier$

production rule), and on the left hand side of equals sign can be written either an array, or code of creating new instance of class in Java, or string, or $true$, or $false$ can be written. Also the example above defines how an array should look like. This is the structure that is set in curly braces and is either a zero or more elements, divided by a comma. Each element can be either an expression or another array, or a string, or $true$, or $false$, or a code that creates a new instance of the class in Java. When a node (corresponding to production rule $variableAssignment$) of the parse tree is visited, new $ClassRelation$ is created, containing information about the name of variable and about its value. This class relation is also added to an instance of $AnnotatedClass$ in $SpecificationLanguageListenerImpl$.

The next group of rules is needed for defining axioms. A production rule, called $axiom$, describes that on the left hand side of symbol $->$ possible inputs of axiom are described, also in the case, if to the input of axiom is passed a list of variables, which is described by production rule $variableIdentifierList$, then to the list of these variables will be given a label $inputVariables$. On the right hand side of symbol $->$ output data of axiom is described, and also that after the list of output variables in parentheses may be written an exception class, that can be thrown, during function's run time, and the name of the realisation is presented in curly brackets in the end of the axiom. Also to the list of output variables is given label $outputVariables$.

```
axiom
    :  ( inputVariables = variableIdentifierList |
       subtaskList
    |  (subtaskList ',' inputVariables =
       variableIdentifierList) ) '->'
    outputVariables = variableIdentifierList (','
       exceptionList)? '{' method = IDENTIFIER '}' ;

subtask
    :  '[' (context = classType '|-')? inputVariables =
       variableIdentifierList '->' outputVariables =
       variableIdentifierList ']'
    ;

subtaskList
    :  subtask (',' subtask)*
    ;

exceptionList
    :  '(' classType ')' (',' '(' classType ')')*
    ;
```

When a node (corresponding to production rule *axiom*) of the parse tree is visited, new *ClassRelation* is created, a list of input variables, output variables, and the name of a function that implements functional dependency, which is implemented by a current axiom are set. Also, if it is needed, then to the *ClassRelation* is added class *SubtaskClassRelation*, which, in turn, contains information about subtasks that are passed to axiom's input. This *ClassRelation* is also added to an instance of *AnnotatedClass*'s class, which is already stored in a variable of *SpecificationLanguageListenerImpl* class.

Sometimes an axiom can have simplified view, that is, it can have only a list of input variables and a list of output variables, which are divided by a symbol $->$. In this case axiom is called goal, and it is described by a following production rule:

```
goal
   :  inputVariables = variableIdentifierList? '->'
      outputVariables = variableIdentifierList
   ;
```

In *SpecificationLanguageListenerImpl* the goal is processed in a similar way to the axiom, the difference is that no information about subtask and functions is saved in *ClassRelation*, as goal does not have them.

Next unit of production rules processes everything related to aliases. Alias can be declared together with variables, which are stored in it. For this process production rules *aliasDeclaration* and *aliasStructure* are needed. The production rule *aliasDefinition* is used for aliases previously declared without the given structure.

```
aliasDeclaration
 : 'alias' ('(' type')')? IDENTIFIER ('=' aliasStructure)?
 ;

aliasStructure
 : '(' ( variableAlias=variableIdentifierList |
    '*.'wildcardAlias=IDENTIFIER )')'
 ;

aliasDefinition
 : variableIdentifier '=' '[' variableIdentifierList ']'
 ;
```

When a node (corresponding to production rule *aliasDeclaration*) of the parse tree is visited, a special object of class, called *Alias* is created. It stores its structure, that is parsed by *aliasStructure* production rule, and this alias is stored in

AnnotatedClass. If the structure of the alias is not defined at the moment of declaration (defined later), then it is done during traverse of the parse tree, which corresponds to production rule *aliasDefinition*, and at this moment *SpecificationLanguageListenerImpl* finds in *AnnotatedClass* a needed instance of *Alias*'s class, and then saves its structure.

In order to define equations in specification language, production rule *equation* is used. This production rule defines that two expression, divided by a symbol =, can be written. Production rule *expression* describes all of the possible arithmetic expressions, which can be described in specification language. Alternatives in production rule *expression* are composed in a way and have such an order that during the process of building of parser tree for this production rule, the tree reflects a correct order of performing arithmetic operations.

```
equation
    :  left = expression '=' right = expression
    ;

expression
    :  term
    |  '(' expression ')'
    |  '-' expression
    |  IDENTIFIER '(' expression ')'
    |  left = expression op=('*' | '/') right = expression
    |  left = expression op=('+' | '-') right = expression
    |  expression '^' expression
    ;

term
    :  NUMBER
    |  variableIdentifier
    ;
```

During the process of visiting a node (corresponding to production rule *equation*) of the parse tree, *SpecificationLanguageListenerImpl* invokes equation solver. Equation solver is a component in CoCoViLa, which tries to solve a given equation for every variable that occurs in it. As a result, for each variable in equation (if such equation is solvable for this variable), an instance of *ClassRelation* is created. After that all instances of *ClassRelations* are added to *AnnotatedClass*.

4.2.2 Implementation of new features

Implementation of the context-free grammar for the specification language, described in the previous section, completely reflected already existing functionality and capabilities of the language in CoCoViLa. With the previous parser based on regular expressions, it was very difficult and time-consuming task to extend the specification language by adding new syntactic structures. Keeping the main goal in mind, it was decided to add new functionality to CoCoViLa by introducing new features to the specification language.

Improvements to the language include:

1. unified syntax for declaring alias with and without assigning its structure (earlier, if the structure of alias was set at the time of declaring the alias, then it was described in parentheses, and if the structure of alias was set after the declaration of alias, then it was necessary to use square brackets);

```
alias x = (a, b);
alias y;
y = (c, d); //previously: y = [c, d];
```

2. assigning values to variables in the same line with their declaration;

```
//previously:
String s;
s = "example";
//now:
String s = "example";
```

3. possibility to declare several constants in one line;

4. possibility to make more convenient setting of variables of one specification, if it is used in source code of another specification. For example, if there is specification A, and there are defined two integer-type variables, named $innerVariable1$ and $innerVariable2$, then during the process of writing a code for another specification, e.g. for B, it would be possible to write a part of code as follows:

```
A specA (innerVariable1=12, innerVariable2=5);
```

The most simplest task was implementing a requirement, presented under first item of enumerated list. In order to create a unified form of alias' structure definition syntax, the author had just to change the production rule $aliasDefinition$, and now it is presented as follows:

```
aliasDefinition
    :  variableIdentifier '=' aliasStructure
    ;
```

Also the author modified code in *SpecificatioLanguageListenerImpl*. Now, when a node (corresponding to production rule *aliasDeclaration*) of the parse tree is visited, then *SpecificatioLanguageListenerImpl* creates an alias and adds it to *AnnotatedClass*, and additionally saves a reference of the alias in its class variable. When a node (corresponding to production rule *aliasDefinition*) of the parse tree is visited, *SpecificatioLanguageListenerImpl* finds in *AnnotatedClass* an alias with corresponding name and saves it in its class variable. Next, when a node (corresponding to production rule *aliasStructure*) of the parse tree is visited, then *SpecificatioLanguageListenerImpl* simply assigns structure to the alias (*SpecificatioLanguageListenerImpl* already has a reference to the alias in its class variable). Changing brackets might seem a trivial task, however it was not the case with previous parser, since adding new or modifying existing regular expression lead to conflicting situations and lots of effort in identifying and fixing them.

In order to implement points 2 and 3 (i.e. to set values to variables during the time of their declaration and to declare several constants as one string), it was necessary to remove production rule *constantDeclaration*, and to change production rule *variableDeclaration*. Also the author added two new production rules: *variableModifier* and *variableDeclarator*.

```
variableDeclaration
    :  variableModifier? type variableDeclarator (','
       variableDeclarator)*
    ;

variableModifier
    :  'static'  # staticVariable
    |  'const'   # constantVariable
    ;
variableDeclarator
    :  IDENTIFIER ('=' variableInitializer)? #
       variableDeclaratorInitializer
    |  IDENTIFIER ('=' variableAssigner)  #
       variableDeclaratorAssigner
    ;
```

Now the *SpecificatioLanguageListenerImpl* uses for declaration of variables a helper-class, called *ClassFieldDeclarator*. When a node (corresponding to production rule *variableDeclaration*) of the parse tree is visited, then *SpecificatioLanguageListenerImpl* sets a type of variable in *ClassFieldDeclarator* (either it is *constant* or *static*). As it can be seen from the example above, after the type of the variable comes a list that is determined by the production rule

$variableDeclarator$. This production rule has two alternatives: *variableDeclaratorInitializer* and $variableDeclaratorAssigner$. When a node (determined by either of these two alternatives) of the parse tree is visited, then *SpecificatioLanguageListenerImpl* will give a command to $ClassFieldDeclarator$ to create a variable with a name that was transmitted with the token $IDENTIFIER$. The only difference is that in the alternative of $variableDeclaratorAssigner$, in recently created $ClassField$, is set the value that is processed with the help of production rule $variableAssigner$, but if there is $variableDeclaratorInitializer$, then it is regarded as equation which needs to be handled. When a parse tree walker finishes visiting a node (corresponding to production rule *variableDeclaration*) of the parse tree, then $SpecificatioLanguageListenerImpl$ resets all data that is stored in $ClassFieldDeclarator$.

In order to implement the last point, a new alternative $specificationVariable$ was added to production rule $variableDeclarator$, and also two additional production rules were defined: $specificationVariableDeclaration$ and *specificationVariableDeclarator*.

```
variableDeclarator
    :  IDENTIFIER ('=' variableInitializer)? #
       variableDeclaratorInitializer
    |  IDENTIFIER ('=' variableAssigner)  #
       variableDeclaratorAssigner
    |  IDENTIFIER ('(' specificationVariableDeclaration
       ')')?  # specificationVariable
    ;

specificationVariableDeclaration
    :  specificationVariableDeclarator (','
       specificationVariableDeclarator)*
    ;

specificationVariableDeclarator
    :  IDENTIFIER '=' expression
    ;
```

In this case, in some part the actions are similar to the procedure of declaring a variable, but in some part they have some differences. Production rule $variableDeclaration$ sets a type of a variable in $ClassFieldDeclarator$ (in this case type of a variable is the name of another specification's class). Next, when a node (corresponding to production rule $specificationVariableDeclarator$) of the parse tree is visited, then the data, processed by this production rule, is processed as a regular equation. It is not needed to declare a variable (which name was transmitted with the token $IDENTIFIER$), as it is declared in another specification.

5. EVALUATION OF APPROACH

Usage of parser generator brings a valuable effort for future development and enhancement of CoCoViLa specification language. Below is presented a list of benefits:

1. **Grammar as a language syntax reference.** One may just open the grammar file and understand the syntax of specification language. Even if one is not familiar with ANTLR and EBNF notation, it is possible to get familiar with basics of them in just a couple of minutes in order to read and understand the grammar. In contrast to this, regular expressions may be difficult to understand even for an experienced developer, as it is troublesome to get a clear picture of regular expressions;

2. **Labels instead of groups.** In production rule it is possible to use labels in order to simplify access to parts of production rule. In regular expressions one should use groups and sequence number (index) of a group in order to retrieve access to a part of expression. If later one new group is added between two already existing groups, then one should review whole source code of a system in order to find locations, where this regular expression is used, and update indexes;

3. **Reusable code.** In contrast to regular expressions, production rules can be easily split in smaller and simpler parts. This is not only more convenient and more transparent way to define grammar, but it also presents an opportunity to reuse those parts in case of necessity in different production rules in various parts of grammar, allowing to lead one of the most important principles of software design: “Don’t repeat yourself” [19]

4. **Fixed bugs caused by the old parser** Some bugs from the previous parser based on regular expressions were fixed. For example, old parser was unable to handle a situation, when a value of a variable of type *String* contained a semicolon symbol or double quotes symbol. Second example, if a statement was followed by a comment on the same line, then next line was completely ignored. Below a code example that cannot be parsed with old parser is presented:

```
String str1 = "hello \"CoCoViLa\" user";
String str2 = "String with ; mark";//next line ignored
int a,b,c;
```

5. **Improved error handling.** If during the parsing process a regular expression does not find a matching substring, then it just fails without any description. ANTLR, on the contrary, provides information on any particular token that causes an error, and suggests token that is expected instead of an erroneous one. For instance, if one tries to parse following code, it will obviously fail:

```
int varname 1;
```

But the error text shown with old and new parser, is different. Old parser shows uninformative message:

```
Syntax error on line 'int varname 1'
```

On the contrary, new parser gives additional information that helps to find an error:

```
Specification parsing error:
int varname 1;
            ^
extraneous input '1' expecting ',', ';', line: 3
```

6. **Less boilerplate code.** ANTLR provides implementation of tree traversal and tree walker patterns that could be easily used in a program. Thus it is not needed to write boilerplate code over and over again.

5.1 Performance comparison

Performance comparison was made to check, whether new parser works faster or not. The results of this comparison are presented in Table 5.1. Performance of parser, based on regular expressions, and parser, generated with ANTLR, was measured and compared in the following way. To measure the performance, the author of the given thesis used 15 different specifications (from a CoCoViLa package of modelling and simulation of complex hydraulic-mechanical systems). Each specification had different length (number of lines). Every specification was parsed 10 times with each parser, and time needed for parsing, was measured, and

Specification length (lines)	New parser (ms)	Old parser (ms)
205	315,4	49,6
240	370,1	52
427	572,1	103,2
502	779	170,5
586	810,5	138,3
631	819,9	142,8
826	853,2	156,5
943	998,7	165,8
1261	1310,3	258,2
1460	1553,6	404,4
1705	1882,2	368,7
2083	1932,3	434,6
2196	2095,5	483,1
2217	2083	474,2
2308	2320,7	526,4

Tab. 5.1: Performance comparison

as a result the average time was calculated. The comparison was performed on a standard laptop with Intel i5 2.40GHz CPU and 4Gb of RAM.

Despite the fact that new parser is slower, this difference is negligible, considering the following reasons:

1. the parsing process is only one step among other automatic steps performed by CoCoViLa, running the generated program, especially complex simulations might take minutes, so in practice, time used for parsing process is not noticeable;

2. slower performance of new parser is compensated with the fact that new parser reduces effort to language development process.

6. CONCLUSIONS

In this thesis, a context-free grammar for the specification language of CoCoViLa was formalised and a parser was generated automatically from the grammar.

First, the author studied computing-related grammars from the area of formal languages, and as a result decided in favour of one type of grammar, i.e. context-free grammar, as it appeared to be the most suitable type for achieving the goals that were set. In addition, the techniques designed for analysis of formal languages were studied.

Second, a comparative analysis of existing parser generators that suited the set goals was conducted. The criteria of choosing a generator were presented, principles of their operating processes and examples of their operation were demonstrated. The comparison resulted in choosing a specific parser generator (ANTLR) for implementing CoCoViLa parser.

In the third part of the thesis author introduced the CoCoViLa specification language and how parser, based on regular expressions, worked. Then, the process of creating a grammar for specification language of CoCoViLa by means of the chosen parser generator was explained. The examples of parse rules along with detailed explanation on how they are handled in code were given.

Finally, the author summarised the work that was done, evaluated results, compared parser, based on regular expression, with newly implemented parser, based on ANTLR. Both parsers were benchmarked in order to compare their operation speeds.

In conclusion, formal specification of CoCoViLa language gives much better understanding of language syntax and provides flexibility and reduces effort in adding new syntactic constructions to the specification language.

6.1 Future work

In the terms of future work, the following points and suggestions can be outlined for implementing in CoCoViLa:

1. Equation solver that is used in CoCoViLa accepts equations in string format only, parses it and builds tree structure that is needed for solving equations. But it is unreasonable to convert tree structure of an equation created by

generated parser back to string format. Thus author would recommend to replace it and implement new equation solver, which could accept equations in tree structure;

2. There are still a lot of places in CoCoViLa outside of parser that still use regular expression for parsing. For example, during code generation, equations are parsed again to find variables in order to replace them with their actual values. It would be better to create special structure for equations that could track all variables and provide convenient way to assign values for those variables without regular expressions;

3. Generated parser provides good error handling for syntax only, but semantic errors are handled in different methods, and do not have standart logic of sending error messages to an end user. Often it is not possible to provide a clear error message about a reason of an error, and also to specify exact occurring position in code (line number). In order to make it more centralised, it is suggested to use Aspect Oriented Programming, and benefits of using such an approach are presented in the author's Bachelor's thesis [20].

CoCoViLa spetsikatsioonikeele parseri realiseerimine kontekstivaba grammatika põhjal

Magistritöö

Ilja Nafigin, 121841IAPM

Resümee

CoCoViLa on Java keeles realiseeritud mudelipõhine tarkvara arendamise platvorm. See võimaldab visuaalsete valdkonnaspetsiifiliste keelte arendamist ning nende keelte kasutamist erinevate arvutuslike probleemide määratlemisel deklaratiivsel viisil. CoCoViLas on visuaalsed spetsifikatsioonid tõlgitud tekstilisteks *spetsifikatsioonikeelteks*. See on Javaga sarnanev keel, mis võimaldab muutujate deklareerimist, määratledes muutujatevahelisi funktsionaalseid sõltuvusi jne. CoCoViLa kasutab *spetsifikatsioonikeele* parsimiseks regulaaravaldistel põhinevat süntaksianalüsaatorit ning see on kitsaskoht (*"bottleneck"*), mis teeb keeruliseks spetsifikatsioonikeele arendamist ja täiustamist tulevikus.

Antud töös on esitatud CoCoViLa spetsifikatsioonikeele uue süntaksianalüsaatori teostamise protsess. See protsess on jaotatud kaheks põhiosaks: teoreetiliseks ja praktiliseks. Esimeses osas on esitatud teoreetiline ülevaade vana süntaksianalüsaatori asendamise võimaluse leidmiseks. Lahendus peab manuaalse süntaksianalüsaatori teostamise asemel kasutama süntaksianalüsaatori generaatorit. Seega on vajalik leida olemasolev süntaksianalüsaatori genereerimise programm. Valimisprotsess on toodud töö teoreetilises osas.

Käesoleva töö teine osa kirjeldab süntaksianalüsaatori generaatori teostust, kuidas on see CoCiViLas intergreeritud. Kirjeldus sisaldab näiteid koodidest ja klassidiagrammidest. On esitatud hinnang antud tööle ning samuti tehtud kokkuvõte neist tulemustest, mis on saavutatud uue genereeritud süntaksianalüsaatori abil. Tulemused sisaldavad uusi spetsifikatsioonikeele teostusfunktsioone ja parsimise protsessist kõrvaldatud programmivigu.

Pakutud lähenemine võimaldab kontekstivaba grammatika kasutamisel spetsifikatsioonikeele formaalse kirjeldamise korral ja keele süntaksianalüsaatori automaatse genereerimise korral arendada spetsifikatsioonikeelt väiksema pingutuse ja kõrgema usaldusväärsusega võrreldes regulaaravaldistel põhineva süntaksianalüsaatoriga.

BIBLIOGRAPHY

[1] http://www.cs.ioc.ee/cocovila/, “CoCoViLa — Model-Based Software Development Platform.” `www.cs.ioc.ee/cocovila/`. [Online; accessed May 2014].

[2] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen, *Modern Compiler Design*. WILEY, 2000.

[3] Daniel I. A. Cohen, *Introduction to computer theory*. WILEY, 2 ed., 1997.

[4] Jacques Loeckx, Kurt Mehlhorn, and Reinhard Wilhelm, *Foundations of Programming Language*. WILEY, 1988.

[5] Michael Sipser, *Introduction to the theory of computation*. PWS, 1997.

[6] Dick Grune and Ceriel J. H. Jacobs, *Parsing Techniques: A Practical Guide*. Springer, 2 ed., 2007.

[7] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2 ed., 2001.

[8] Terence Parr and Kathleen Fisher, “Ll(*): The foundation of the antlr parser generator,” *SIGPLAN Notices*, vol. 46, pp. 425–436, June 2011.

[9] Terence Parr, Sam Harwell, and Kathleen Fisher, “Adaptive ll(*) parsing: The power of dynamic analysis.” `http://www.antlr.org/papers/allstar-techreport.pdf`.

[10] M. Kay, “Chart generation,” in *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ACL ’96, (Stroudsburg, PA, USA), pp. 200–204, Association for Computational Linguistics, 1996.

[11] S. L. Graham and S. P. Rhodes, “Practical syntactic error recovery,” *Communications of the ACM*, vol. 18, pp. 639–650, Nov. 1975.

[12] M. J. Fischer, "Some properties of precedence languages," in *Proceedings of the First Annual ACM Symposium on Theory of Computing*, STOC '69, (New York, NY, USA), pp. 181–190, ACM, 1969.

[13] Dick Grune and Ceriel J. H. Jacobs, *Parsing Techniques: A Practical Guide*. Ellis Horwood Ltd, 1991.

[14] F. Benhamou, ed., *CP'06: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, (Berlin, Heidelberg), Springer-Verlag, 2006.

[15] M.G.J. van den Brand and C. Groza, "The algebraic specification of annotated abstract syntax trees." `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.3593&rep=rep1&type=pdf`, 1994.

[16] Merriam-Webster inc, *Webster's Dictionary*. Konemann UK Ltd, 2000.

[17] E. Gagnon, "*SableCC, an object-oriented compiler framework*," Master's thesis, McGill University, 1998.

[18] Terence Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[19] Andrew Hunt and David Thomas, *The Pragmatic Programmer*. Addison-Wesley Professional, 1999.

[20] Ilja Nafigin, "*OOP ning AOP võrdlus: rakenduslik erinevuste analüüs, kasutades Java JDK ja AspectJ*," Bachelor's Thesis, Tallinnn University of Technology, 2012.