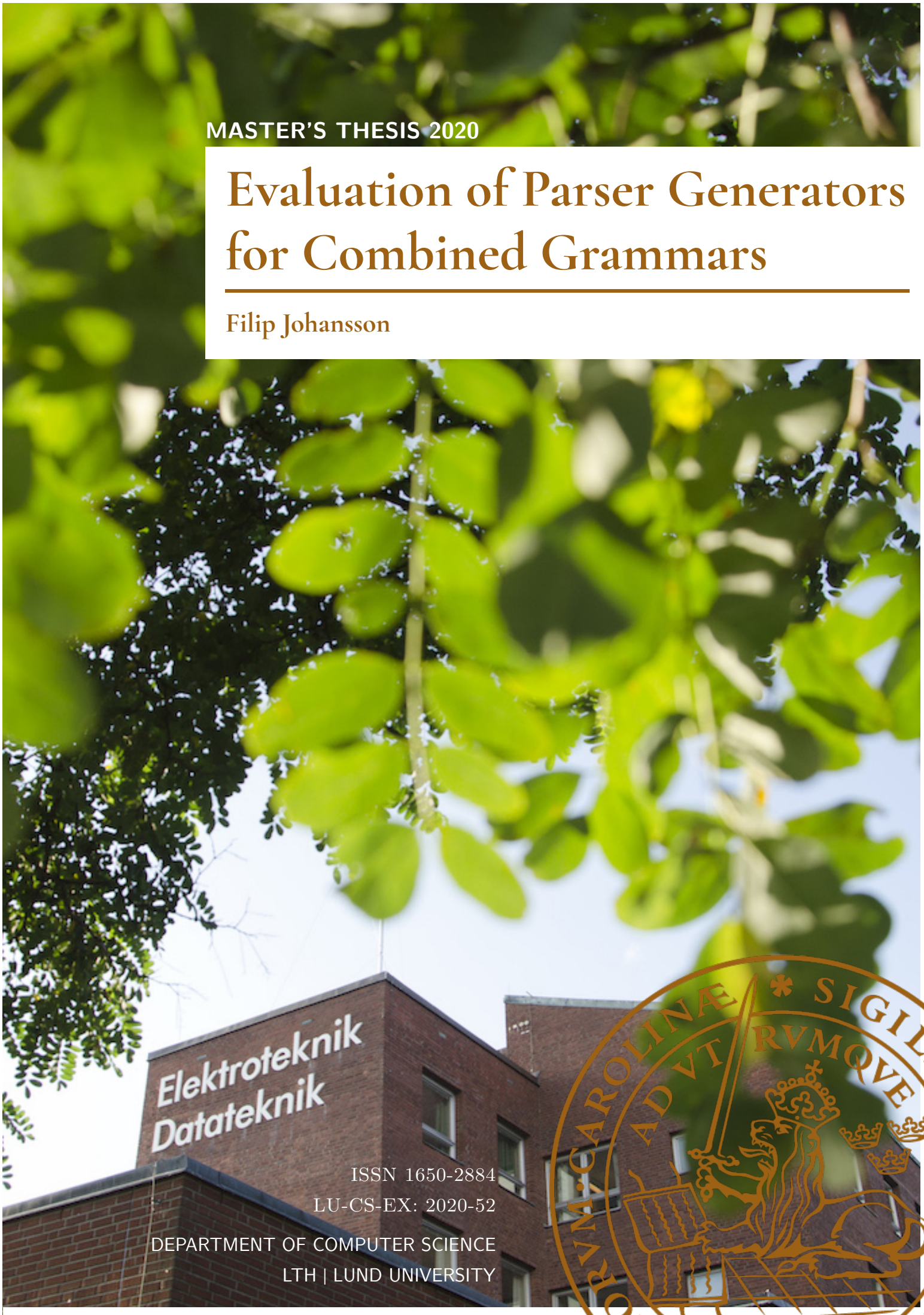


MASTER'S THESIS 2020

Evaluation of Parser Generators for Combined Grammars

Filip Johansson

ISSN 1650-2884
LU-CS-EX: 2020-52

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-52

Evaluation of Parser Generators for Combined Grammars

Filip Johansson

Evaluation of Parser Generators for Combined Grammars

Filip Johansson
fi2541jo-s@student.lu.se

September 4, 2020

Master's thesis work carried out at ABB AB.

Supervisors: Görel Hedin, gorel.hedin@cs.lth.se
Stefan Sällberg, stefan.sallberg@se.abb.com

Examiner: Niklas Fors, niklas.fors@cs.lth.se

Abstract

Parser generators are code generator tools that generate code for a parser based on an input grammar specification. However, problems can arise when one wants to combine grammars. Grammars may overlap in ways that make the combination challenging to implement with a parser generator. In this thesis, I have evaluated both commonly used parser generators and experimental tools from an industrial perspective and the ability to connect to other code generation tools. The generators have been tested by producing implementations for two test languages and one industrial case. The results show that all evaluated generators can parse combined grammars in cases with clear separators between the grammars. However, when it comes to languages where there are no clear separators, both the popular and experimental tools struggle; though experimental algorithms can parse a larger portion of these cases.

Keywords: Parser generators, scanning, combined grammar, code generation, lexical states

Acknowledgements

I want to thank my supervisor Görel and my examiner Niklas for helping me plan my thesis. I also want to thank Görel for her invaluable help along the way. I also want to thank Stefan, Christina, and Mårten for a warm welcome to ABB and enjoyable weekly meetings.

Contents

Chapter 1
Introduction

One essential part of a compiler is the parser. The parser's role is to organise the input code text according to some grammar rules. One can often generate parsers using parser generator tools. Such tools create the code for a parser based on an input grammar specification; thus, allowing the developer to both construct the parser faster and with less effort. Moreover, the parser becomes easier to understand and to change compared to if the parser would be written by hand.

There are cases when one wants to combine different languages in the same code. For example, one might want to embed SQL into a general programming language [13]. However, it then can become troublesome to use parser generators since often the grammars of these component languages are incompatible. Furthermore, it is difficult to know which parser generator resolves this issue in the best way. For these reasons, this thesis is dedicated to evaluating parser generators for the parsing of combined grammars.

To gain insight into how to best solve the problem of parsing combined grammars in industry, I have also conducted a case study at a company called ABB. Here I have evaluated the parser generators along factors determined to be important by ABB. I result with a recommendation of the parser generator tool most suited for the company.

Since parsing is only one piece of a compiler, parsers must connect to other tools, for example, JastAdd [15]. I have thus, also, explored how one can integrate the parser generators with other compiler building tools.

1.1 An Industrial Context

A team at ABB, Malmö is in the process of evaluating parser generators for parsing IEC 61131-3 [16] languages. IEC 61131-3 is a standard which outlines several programming languages, both textual and graphical, for use in programmable auto controllers. These are *Instruction List*, *Structured Text*, *Ladder Diagram*, and *Function Block Diagram*. Furthermore, ABB has also specified their own graphical language. Due to how the developer tool is constructed, user-

specified code blocks in these languages are inserted into an XML document as XML elements. Other, relevant data, such as globally defined variables, is also stored in the XML. It is these XML files that serve as the input to their compiler.

The company generated a parser for these XML files using the JavaCC parser generator [8]; however, the team ran into issues. The grammar specification for the XML structure and the grammar specifications for the embedded code blocks are not compatible. The developers managed to solve this issue with the technique of *lexical states*. And even though this solution works, ABB's choice of JavaCC was unresearched. Thus, they want to determine if any other parser generator is better suited for them. The factors that are most important for the team are functionality, ease-of-use, performance, type of license, quality of error messages, ease of altering error messages, and amount of available support. Moreover, the company uses JastAdd for many compiler tasks running after the parsing. Therefore, a parser generator for ABB must be able to integrate with JastAdd.

1.2 Goals

There are two main goals of this thesis:

1. Determine parser generators suitable for parsing combined grammars from an industrial perspective.

2. Determine how parser generators can connect and integrate with other tools.

1.3 Delimitations

Before the start of this thesis I decided, in conjunction with my supervisor and the people at ABB, to limit the evaluation to a few of the most popular parser generators (ANTLR 4 [25], JavaCC [8], CUP [23], JFlex [34]) and a few research projects with new experimental techniques on how to solve combined grammar parsing (Copper [37], MetaLexer [6]). All of the tools above generate Java code. We did this since we desired to connect the parser generators to JastAdd, which also generates code in Java. Furthermore, all the selected tools are open source.

1.4 Structure of Report

Presented next is Chapter 2 Background in which I describe the theory and background needed to understand this work. The background topics include the compiler pipeline, explanations of scanners and parsers, abstract syntax trees, an introduction to the tools used in the thesis, and lastly an explanation of JastAdd. In Chapter 3 General Evaluation, I present an evaluation of the generators. I describe how I constructed test languages and used the tools to implement parsers for these. I outline the results of the implementations in terms of functionality, ease-of-use, and how well the parsers were able to integrate with JastAdd. After this, I also look into how much the tools are maintained and what kind of licenses they have. This chapter concludes in an overview of the results of the evaluation. In Chapter 4

Industrial Case Study, I evaluate the tools selected from Chapter 3 on ABB's use case. To do this, I look into the customisation of error messages and a performance evaluation. I conclude Chapter 4 with a recommendation for a tool for ABB. In Chapter 5 Discussion, I discuss the methodology and results of the evaluation. I propose ways to improve the thesis, compare it to related works, and discuss potential future work. Lastly, the report ends with Chapter 6 Conclusion, where I give a summary of the thesis and answer how well the goals have been achieved.

Chapter 2
Background

In this chapter, I describe the background necessary to understand this thesis. First I describe the compiler pipeline to give the context of a parser role in a compiler. Then I go on to explain how a scanner and parser work, and to describe why attributed syntax trees are used. Lastly, I introduce the tools I will evaluate and explain how JastAdd works.

2.1 Compiler Pipeline

A compiler can be described as a pipeline of several different components, where the output of one component is the input into another. Aho et al. [2] divides these components as follows:

Lexical Analyser: Also known as a scanner. This component uses the code as input in the form of a character stream. Strings of at least one character are packed into *tokens*, for example the string `"+"` is represented as an `ADD` token. The output of this stage is a stream of such tokens. Considering a function call `func(90 * 4 - 1);`, the resulting token stream would look like `[IDENT(func) LPAR INTEGER(90) MUL INTEGER(4) SUB INTEGER(1) RPAR SEMICOL]` (where `IDENT` stands for identifier, `LPAR` stands for left parenthesis, and `RPAR` stands for right parenthesis).

Syntax Analyser: Also known as a parser. Here the token stream is organised by constructing a tree data structure, called a *parse tree*. This tree represents the grammatical structure of the code. Leaf nodes in a parse tree are tokens, while the other nodes are more general categories in the grammar. Such a category could be, for example, statements or expressions. The token stream from above would result in a parse tree shown in Figure 2.1.

Semantic Analyser: This step uses the syntax tree to check language semantic rules. Such checks could, for example, include name-checking, where the compiler ensures that

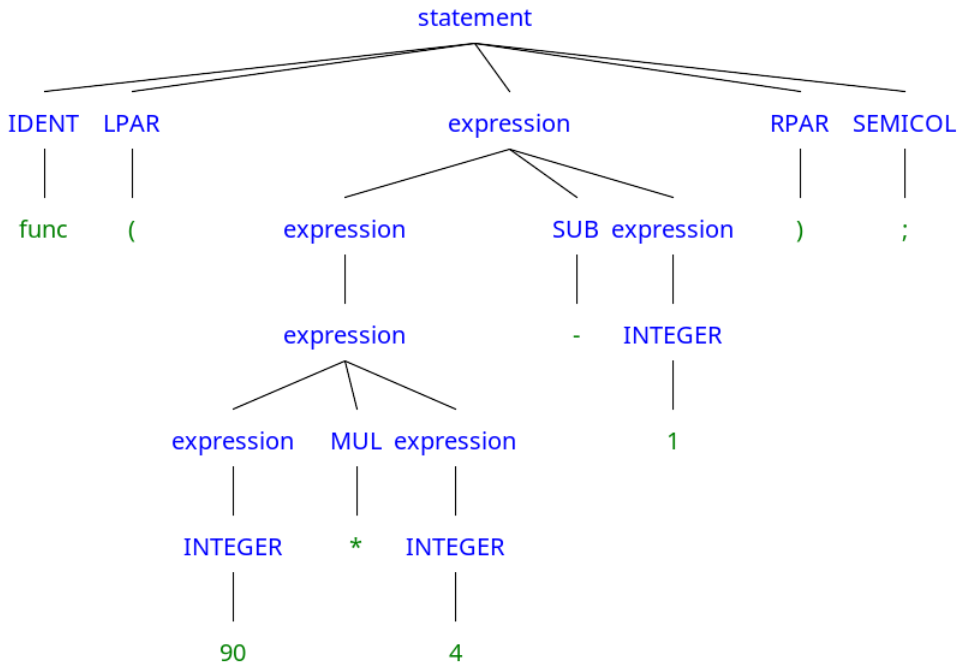


Figure 2.1: An example parse tree for the function call statement `func(90 * 4 - 1)`.

identifiers are defined before use and that they are not doubly defined. Another example would be type checking.

Intermediate Code Generator: Here the syntax tree is converted into an intermediate code format. Intermediate code is often similar to machine code; however, it is hardware independent.

Machine-Independent Code Optimiser: This step optimises the intermediate code. It is an optional step, but can help to produce better machine code compared to using optimised intermediate code.

Code Generator: Generates machine dependent code based on the intermediate code.

Machine-Dependent Code Optimiser: Optimises the machine dependent code to improve performance. Technically, this step is optional as well.

To solve the problem of parsing combined grammars, the lexical analyser or the syntax analyser usually make use of special techniques. Thus, these two steps of the compiler will be the main focus of this thesis. Still, JastAdd is one way to do semantic analysis. So, since JastAdd will be used to test the parser generators ability to connect with other code generation tools, also semantic analysis will be touched upon, though briefly.

2.2 Scanner

The scanner transforms the code in the form of a character stream into a stream of tokens [2]. Appel et al. [3] explain that regular expressions are one way to represent tokens. Regular expressions can be acyclic, that is they can match finitely many strings, or cyclic, which means they can match infinitely many strings [6]. Keywords and punctuation tokens tend to be acyclic while identifiers and literals tend to be cyclic. See Table 2.1 for examples.

Token	Regular Expression	Acyclic / Cyclic	Example Strings
`FOR`	"for"	Acyclic	`for`
`LPAR`	"("	Acyclic	`(`
`IDENT`	[A-Za-z][A-Za-z0-9]*	Cyclic	`a, var, i2, aaggg444bbb`
`INTEGER`	[0-9]+	Cyclic	`1, 4545, 05`

Table 2.1: Examples of regular expressions for tokens.

Several regular expressions can match the same string. For example, both the regular expressions for `FOR` and `IDENT` in Table 2.1 matches "for". There are two common rules to solve ambiguities like this [3]:

Rule Priority: The regular expression that is defined first has priority. So, if `FOR` is defined before `IDENT`, "for" will match `FOR` and vice versa.

Longest Match: The scanner always matches the longest token. So, for the string "for5", `IDENT` is matched instead of for example `[FOR INTEGER]`.

There are a few regular expressions that may be described as matching non-tokens [3]. Common examples of this are white space and code comments. These are not necessary for parsing the syntax of the code; thus, when the scanner matches such a regular expression, no token is passed to the parser, i.e. they are skipped.

Appel et al. [3] go on to explain that finite automata provide a framework for efficiently representing regular expressions in code. A finite automaton consists of states and edges. The automaton changes state when the input character matches one edge. If the automaton cannot pair with any edge, it throws an `ERROR` token. Typically, the user declares tokens with their regular expression in a scanner specification. A scanner generator then generates finite automata for all tokens in code from this specification.

2.2.1 Lexical States

Lexical states is a technique used in many lexical analyser generators, including several evaluated in this thesis. The method allows the user to specify which regular expression will be active at what context in the input. Lesk's and Schmidt's [21] scanner generator Lex (an early scanner generator that produces scanners in C or Ratfor) was one of the first tools that made use of lexical states. Though it should be noted, in the case of Lex, lexical states are referred to as start conditions.

The technique implies that every lexical rule in the specification can be given one or several states. The lexical rule is then only active when the scanner is in the same lexical state

of said rule. Consider, the example specification below in pseudo-code, where <> denotes lexical states:

```
<STATE_A> TOKEN_A = [a-zA-Z] -> CHANGE_STATE(STATE_B);
<STATE_B> TOKEN_B = [a-zA-Z] -> CHANGE_STATE(STATE_A);
```

Here, the state changes to `STATE_B` when a `TOKEN_A` token is produced and to `STATE_A` when a `TOKEN_B` is produced. This shuffling of the lexical state allows both lexical rules to utilised at different times even though they are identical. This phenomenon is noticeable if assuming the scanner starts in `STATE_A` when the scanner encounters the string "aaa" the following token stream will be produced: `[TOKEN_A TOKEN_B TOKEN_A]`.

Lexical states can be beneficial when dealing with combined grammars. A lexical state can then correspond to the entire set of tokens corresponding to one grammar. The lexical state is changed when one wants to start parsing another grammar.

2.3 Parsing Algorithms

The parser's role is to organise the incoming token stream into a data tree structure called a parse tree [2]. For this purpose, there exists two different main algorithms tend to be used in parsers. These are left-to-right, leftmost derivation (LL) and left-to-right, rightmost derivation (LR). Aho et al. [2] explain that both algorithms are based upon context free grammars (CFGs). CFGs consists of a start symbol, production rules, terminals, and non-terminals. Terminals are tokens and also leaf nodes of the parse tree. Non-terminals are the categories that form the internal nodes of the parse tree. A production rule consists of a left-hand side (LHS) which is a non-terminal and a right-hand side (RHS) which is a list of non-terminals or terminals. A production rule for a while statement in Java could look like the following:

```
while_stmt := WHILE LPAR expr RPAR stmt_list
```

The start symbol of a CFG refers to the LHS that the parser tries to construct first. The start symbol also forms the root node of the parse tree.

Both LL and LR parsers have a specific number of lookaheads. Lookaheads are the next coming tokens on the token stream. A parser can check the lookahead(s) to predict what to do. Parsers with a k number of lookaheads tokens are denoted as LL(k) or LR(k) respectively.

LL: An LL parser builds the parse tree top-down [2]. It starts with constructing the starting non-terminal and then the first child and so on in a depth-first manner. A common way to realise a recursive parser is as a rcursive descent parser [3]. When an LL parser has several alternatives, it can use the lookaheads to predict which rule to enter. For example, if an LL(1) parser has `X` in its lookahead it knows to enter alternative (1) in the following production rule:

```
a := X Z        (1)
   | Y a        (2)
```

However, there are a few grammar constructs that LL(1)-parsers cannot parse. For example, they cannot handle common prefixes. If the parser has `Y` in its lookahead it cannot predict whether to choose an alternative (1) or (2) in the rule below:

```
a  :=  Y  Z          (1)
    |  Y  a          (2)
```

Another construct that no LL-parsers can parse is left recursive rules. A left recursive rule could look like:

```
a  :=  a  Y          (1)
    |  Y             (2)
```

Here LL parsers can become stuck in an infinite loop by selecting alternative (1) repeatedly.

LR: LR parsers are slightly more advanced than their LL counterparts. Appel et al. [3] describe that the LR-algorithm utilises a stack where it pushes tokens upon. When an entire RHS of a rule is present on the stack, these symbols can be reduced to the corresponding LHS non-terminal. In this way, the parser builds the tree is bottom-up. Furthermore, rules do not need to be entered, so LR parsers do not have the same issues of common prefixes and left recursion. However, LR parsers can have problems in terms of shift-reduce conflicts. These occur when the parser algorithm is not certain whether to push a new token upon the stack or whether it should reduce the current ones.

Aho et al. [2] describe that in practice, LR(1)-parsers tend to result in large state transition tables. Therefore, it is often more reasonable to produce an LALR(1)-parser instead. LALR(1) is a subset of LR(1) that has a less heavy implementation. The expressiveness lost from going from LR(1) to LALR(1) is usually not noticeable.

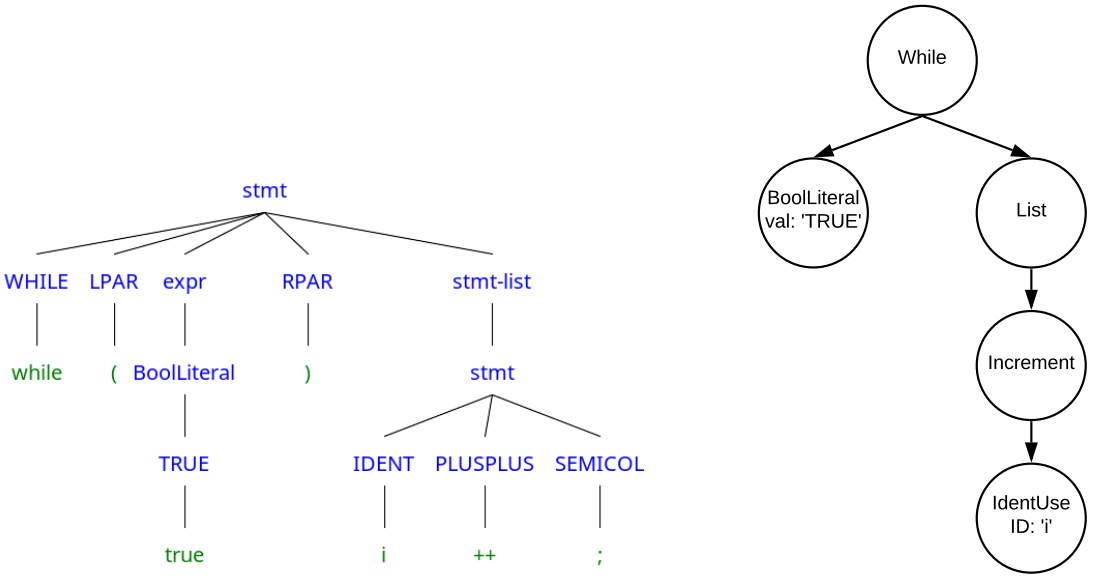
2.3.1 Combined Grammars

The purpose of this report is to examine the use of parser generators on combined grammars. In this work, I define a combined grammar as a set of at least two CFGs, where at least one CFG uses terminals or non-terminals of another CFG in its own production rules. For example, one can combine a grammar for Java with grammars of Swul or XML as done in [5].

2.4 Abstract Syntax Tree

Parse trees carry unnecessary information and do not have the best structure for semantic analysis. For example, the parenthesises and semicolons do not carry important information in a while statement of the form `while ( true ) i++;`. Parsers can overcome this issue by constructing abstract syntax trees (AST). In an AST the nodes represent programming constructs rather than non-terminals like in a parse tree [2]. Consider the parse tree for the following expression: `90 * t + 1` in Figure 2.2a. An AST of this might look as Figure 2.2b.

In the practical sense, an AST node is usually constructed as a Java object at appropriate places in the parser code. The user can often add such code through the use of semantic actions. Semantic actions are snippets of code in the target language of the generator that can be embedded in a parser specification. These snippets of code are then executed when the parser uses the rule corresponding to said AST node.



(a) Parse Tree **(b)** AST

Figure 2.2: Parse tree and corresponding AST of the statement `while ( true ) i++;`.

2.5 Tools

This section aims to give some background on the tools that are evaluated in the thesis. I have evaluated the following tools: ANTLR 4, JavaCC, CUP, JFlex, Copper, and MetaLexer.

ANTLR 4: ANTLR is a popular parser generator, first developed by Terence Parr [25]. It is now in its fourth version. ANTLR 4 uses a parsing algorithm called ALL(*) [27], which stands for Adaptive LL-parsing with unlimited lookaheads. The adaptive part refers to that the algorithm can perform more advanced rule predictions, compared to regular LL algorithms, that are parser call stack sensitive and calculated in run-time; however, as all LL algorithms, ALL(*) cannot handle left recursion. ANTLR 4 resolves this by rewriting direct left recursive rules before parsing, thus allowing the user to utilise these constructs in the specification. Furthermore, common prefixes will not cause errors either, due to the unlimited lookaheads. From the parser specification, ANTLR 4 generates both a scanner and a parser [32], thus taking care of both the lexical and syntactical analysis. ANTLR 4 parsers construct a parse tree and lack inherent functionality to build an AST as it instead aims to use the visitor or listener pattern. Lastly, it should be noted that ANTLR can generate code to several target languages, but in this thesis, I only focus on the Java output.

JavaCC: JavaCC was first developed at Sun Microsystems Inc. but is now maintained as an open-source project on GitHub [8]. It generates recursive descent parsers that are LL(k) where $k = 1$ by default. Using LL parsers means that the user can have issues with both left recursion and common prefixes at times. To overcome common prefixes, JavaCC allows the user to increase the lookahead at places locally in the grammar.

Additionally, JavaCC includes scanner generation, so the tool handles both the lexical analysis and as well as the syntax analysis. Developers can easily add AST building functionality by using JJTree. JJTree is a pre-processor to JavaCC which injects the needed code for AST construction into the JavaCC specification.

CUP: Scott Hudson first developed the tool Construction of Useful Parsers (CUP); however, it is now maintained at the Technical University of München [23]. CUP is used to generate LALR parsers from a grammar specification. As all LALR parsers, CUP parsers will not have issues with common prefixes and left recursion. Instead, problems might manifest in the form of shift-reduce conflicts. In contrast to ANTLR and JavaCC, CUP only generates a parser, i.e. a syntax analyser. Thus, if using CUP, one needs to construct or generate a lexical analyser in another way. In this thesis, I use the scanner generators JFlex, which is said to work particularly well with CUP, and MetaLexer together for this purpose. Finally, one should note that CUP allows AST building through semantic actions.

JFlex: JFlex is a popular scanner generator [34] inspired by the older tool JLex [4]. JFlex generates a lexical generator from a specification of tokens and regular expressions. The tool is designed to work together with CUP, which will be used together with JFlex in the evaluation.

Copper: Copper generates LALR(1) parsers with specifications similar to CUP. It is developed and maintained by Van Wyk and Schwerdfeger at the University of Minnesota [37]. Unlike CUP, Copper includes both a parser generator and a scanner generator in one package. Copper is unique in that it has an experimental algorithm to resolve combined grammar parsing issues. The context-aware algorithm works through a connection between the scanner and the parser. The scanner takes as input a set of valid lookahead tokens, determined by the parser, and is only able to return tokens in this set. As output, the scanner can return a variable number of tokens. In case of zero returned tokens, a parse error is thrown, in case of one token, this token is used by the parser, and in case of several tokens, the parser must utilise so-called disambiguation functions to choose which token to use. The scanner generator follows non-standard priority rules to facilitate this algorithm. Specifically, it does not follow the first rule declaration priority. Instead, these priorities are specified manually.

MetaLexer: MetaLexer was developed as a master thesis at McGill University by Andrew Casey [7]. It is a pre-processor to JFlex that allows lexical analysers to use new experimental features, such as inheritance. When using MetaLexer, the developer can easily combine modules of scanners [6]. MetaLexer solely generates scanners and thus needs to be used with a parser generator. In the evaluation, MetaLexer will be used together with CUP.

2.6 JastAdd

JastAdd is a useful tool for constructing compilers. It is especially helpful for semantic analysis and intermediate code generation. The tool has been used in many compiler related projects [33]. For example, the ExtendJ Java compiler is written with JastAdd [10].

The tool generates Java classes for the AST-nodes that are based upon a user-written abstracted grammar. Here the user defines all the classes of AST-nodes by specifying the classes of their children. Furthermore, object oriented inheritance can be used by extending AST-nodes classes with others [14]. Consider the excerpt from an AST-grammar below. The JastAdd generated classes from this excerpt will have the structure shown in Figure 2.3.

```
abstract Expr;
IntLiteral : Expr ::= <Value:Integer>;

abstract BinaryExpr : Expr ::= Left:Expr Right:Expr;
AddExpr : BinaryExpr;
MulExpr : BinaryExpr;
```

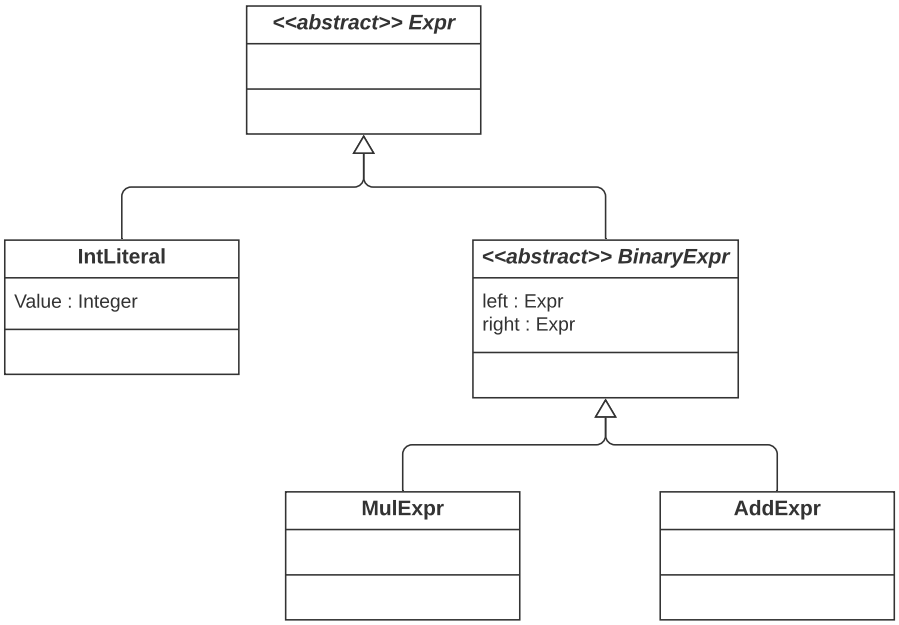


Figure 2.3: UML class diagram for JastAdd AST-specification.

The power of JastAdd lies in its ability to define properties declaratively, called attributes, of the nodes. There are different kinds of attributes: synthesised and inherited. Synthesised attributes are defined in the node it is accessed in, while inherited attributes are defined in ancestor nodes [14].

Reference attributes point to other AST-nodes. They can be of both the synthesised and inherited variety. Reference attributes can be useful during different parts in the semantic analysis. Instead of using a symbol table as Aho et al. [2] describe, one can simply point to the node where the variable of the other construct is defined [14].

Chapter 3
General Evaluation

This chapter presents the method and results of a general evaluation of the tools. The evaluation investigates the tools in terms of functionality, ease-of-use, how widely used the tools are, and the type of license. First, the test languages used in the evaluation are presented, followed by a showcasing of the implementations. Lastly, there is an investigation of the usage of the tools and their licenses.

3.1 Test Languages

To test the parser generators, I have constructed two languages. I have then implemented parsers for these languages to determine the tools' functionality and ease-of-use. One language, which is inspired by ABB's set up, consists of code blocks embedded in an XML structure. This test language is referred to as *XML Blocks*. The code blocks in XML Blocks are either Oberon-0 code [38] or State Machine Language (SML) code [14]. The other language is a combination of Oberon-0 and an extended version of SML (ESML). This test language is referred to as the *Mixed Language*.

Oberon-0 is a subset of the language Oberon, which is inspired by Modula-2 and Pascal [38]. Furthermore, only a smaller subset of Oberon-0, as defined by L0 and L1 in the LDTA 2011 Tool Challenge [36], is implemented. This subset includes constants, types, variables, expressions, assignment statements, while statements, and if statements. It is not vital to implement the entirety of Oberon-0 since it is not the amount language features themselves that are of importance, but rather the combination of and the switching between grammars.

SML is a small example language that describes state machines, used to highlight JastAdd features [14]. I have constructed an extended version of SML called ESML by adding OBJECT and WHEN statements. ESML is used to introduce more difficult parsing cases in the Mixed Language.

3.1.1 XML Blocks

XML Blocks consists of an outer XML structure that can contain code blocks of either Oberon-0 or SML. In XML Blocks the borders between the different components are clear, so the main parsing problem is how to determine which grammar to operate in and at which points to change grammars. Furthermore, to complicate things, XML cannot be represented as one grammar by itself, but rather a combination of inside- and outside-tag grammars [24]. One example of XML Blocks with a simple SML and a simple Oberon-0 block is:

```
<Program>
  <Name>One SML and One Oberon-0 Block</Name>
  <Description isEmpty="true"/>
  <Variables/>
  <CodeBlocks>
    <CodeBlock>
      <Name>CodeBlock1</Name>
      <Language>SML</Language>
      <Code>
        state S1;
        state S2;
        trans t1: S1->S2;
        trans t2: S2->S1;
      </Code>
    </CodeBlock>
    <CodeBlock>
      <Name>CodeBlock2</Name>
      <Language>Ob0</Language>
      <Code>
        MODULE EmptyModule;
        END EmptyModule.
      </Code>
    </CodeBlock>
  </CodeBlocks>
</Program>
```

I have constructed a set of tests in XML Blocks to be used to evaluate the functionality of the parser generators. The tests cover a multitude of cases but can be categorised as follows:

- **A1:** XML with no code blocks.
- **A2:** XML with Oberon-0 code blocks.
- **A3:** XML with SML code blocks.
- **A4:** XML with both Oberon-0 code blocks and SML code blocks.
- **A5:** SML keywords used in XML tag element text.

3.1.2 Mixed Language

Mixed Language is a direct mixture of ESML and Oberon-0. There are no clear separations upon where one language starts and the other one ends. The issue here will be to notice when the switch between component language occurs. Mixed Language is purposely designed to introduce complex parsing situations. This complexity is highlighted by that the user can nest languages inside one another. A straightforward example of the Mixed Language looks like:

```
1  MODULE NextToModule;
2  VAR x : INTEGER;
3  BEGIN
4    STATE s1;
5    STATE s2;
6    TRANS t1 : s1 -> s2;
7    TRANS t2 : s2 -> s1;
8    OBJECT o1 {
9      START = s1;
10     END = s2;
11   };
12   WHEN o1 IS IN s1 THEN {
13     NEXT_TRANS = t1;
14   };
15
16   IF 1 < x THEN
17     x := x + 1;
18   ELSE
19     x := x - 1;
20   END;
21   ;
22   x := x * 2;
23   WHILE x < 1 DO
24     ;
25   END
26
27 END NextToModule.
```

In the example above, row 4-14 is ESML, and the rest of the code is Oberon-0. It might be possible to write the Mixed Language as a single grammar, but in the implementations, the grammars will be separated to allow investigation of the issues of combining grammars.

Like for XML Blocks, a test set with difficult cases has been constructed for the Mixed Language as well. The categories of these tests are presented below:

- **B1:** ESML or Oberon-0 keywords are used as identifiers in the other language.
- **B2:** ESML and Oberon-0 code is mixed next to each other.
- **B3:** No semicolons at the transition between ESML to Oberon-0 or vice versa.

- **B4:** Recursion, meaning Oberon-0 code can be constructed inside an ESML code block and vice versa.
- **B5:** ESML and Oberon-0 share the same keyword.
- **B6:** ESML and Oberon-0 share the same operator.

3.2 Implementations

To evaluate the functionality of the parser generators, I compare the ability of the implementations to parse the test sets composed by the test languages. Any tests that cannot be parsed counts against the parser generator's functionality. Furthermore, to measure the ease-of-use of the tools, I count the lines of code (LOC) of implementations. The more LOC, the more difficult the implementation is perceived to be. The repository with all the implementations is available at [18].

3.2.1 Handling Combined Languages

The implementation can be organised into three categories: plain lexical states, semantic action-driven lexical states, and context aware scanning. Plain lexical states solutions change lexical states when tokens are matched. These transition between states are usually implemented through specific statements in the lexical specification. Semantic action driven lexical states implementations are very similar to plain lexical states, the difference being that lexical states are changed with semantic actions. This allows custom logic in the form of code to give more control over lexical state transitions. The last type, context aware scanning, is a custom algorithm means once does not need to specify state transitions. Which tool is what type is shown in Table 3.1.

Implementation	Category
ANTLR 4	Plain Lexical States
Copper	Context Aware Scanning
JavaCC	Plain Lexical States
JFlex/CUP	Semantic Action-Driven Lexical States
MetaLexer/CUP	Plain Lexical States

Table 3.1: Category of implementations.

Implementing a parser for XML Blocks was straight forward for the tools that use lexical states because there is a clear limit upon where one language starts and the other ends. All plain lexical state implementations, make use of a token that may look like (for SML) `<Code>SML</Code>`. When the scanner reads such a token, the lexical state changes to another language (in this case SML). Though this is an easy solution, there are limitations with this way of doing it. By representing that entire XML statement as one token, one effectively removes this part from standard XML grammar. In real XML, arbitrary white spaces are allowed at certain places. For example, the XML above should be equivalent with `<Code >SML</Code>`; however, the scanner would not match that XML as token due to

technically being another regular expression. This flaw makes the lexical state implementations fragile at these transition tokens. The semantic action driven lexical state implementation is observably a bit better. Due to JFlex's ability to change lexical states through semantic actions, rather than as a command, one can postpone the changes of lexical states with a bit of added logic. Thus one can keep a larger part of the XML statements inside the actual XML state.

For the Mixed Language, the lexical state implementations have a harder time. Here there is no hard border for when a new language should start; ESML and Oberon-0 code can come in any order possible. This means that there is no equivalent to a `<Code>SML</Code>`. To try to circumvent this issue, I implemented an intermediate state which shall be referred to as the *ambiguous state*. In the ambiguous state, the parser does not know what component language it is currently trying to parse. However, based on the token that is parsed in this ambiguous state, the parser deduces what language it is parsing and thus change lexical state accordingly. Let us say for example, that an `IF` token is scanned; then one transfers the lexical state to Oberon-0 as this is the only component language that has IF-statements.

In a reversed fashion compared to the lexical states implementations, Copper's implementation struggled more with XML Blocks than with the Mixed Language. In particular, for XML Blocks Copper must employ a disambiguation function. On the grammar level, the Copper implementations were very similar to the lexical states' parsers. However, in the lexical specification, there are more considerable differences. Here, the transitions between lexical states do not need to be specified, but instead, one must specify all the priorities between the tokens. Furthermore, one must implement disambiguation functions when these priorities are not enough.

3.2.2 Results for XML Blocks Language

The Table 3.2 outlines which tests the implementations were able to handle in the XML Blocks test suite. As the table shows, all implementations were able to handle all the tests in the test set. This result indicates that all the categories of implementation are capable of parsing combined grammars with clear borders between the different languages.

Implementation	A1	A2	A3	A4	A5
ANTLR 4	✓	✓	✓	✓	✓
Copper	✓	✓	✓	✓	✓
JavaCC	✓	✓	✓	✓	✓
JFlex/CUP	✓	✓	✓	✓	✓
MetaLexer/CUP	✓	✓	✓	✓	✓

Table 3.2: Successfully parsed tests in the XML Blocks test set.

3.2.3 Results for Mixed Language

In Table 3.3, the results from the Mixed Language test suite are shown. Unlike XML Blocks, the implementations were not able to parse all examples. Specifically, the implementations struggled with B1 and B3.

Implementation	B1	B2	B3	B4	B5	B6
ANTLR 4		✓		✓	✓	✓
Copper		✓	✓	✓	✓	✓
JavaCC		✓		✓	✓	✓
JFlex/CUP		✓		✓	✓	✓
MetaLexer/CUP		✓		✓	✓	✓

Table 3.3: Successfully parsed tests in the Mixed Language test set.

All the implementations struggled with the context-based preference of keywords in ESML and Oberon-0. The parser cannot determine when a keyword in one language should be considered as an identifier in the other. Below is one example that the implementations struggle with:

```
1  MODULE IdentKeywordModule;
2  VAR STATE : INTEGER;
3  BEGIN
4    STATE := 2;
5    STATE s1;
6    STATE s2;
7    TRANS t1 : s1 -> s2;
8    TRANS t2 : s2 -> s1;
9    OBJECT o1 {
10     START = s1;
11     END = s2;
12   };
13   WHEN o1 IS IN s1 THEN {
14     NEXT_TRANS = t1;
15   }
16 END IdentKeywordModule.
```

On line 2 and 4 `STATE` should be considered an identifier in Oberon-0, while on line 5 and 6 `STATE` should be considered an ESML keyword. The ambiguous state approach does not work here because one cannot know which state to transfer to when receiving a STATE token as this changes depending on the type of statement. Furthermore, the Copper implementation also fails here since the scanner returns both a STATE and an IDENT token; however, there is no exact way to disambiguate between them.

Another example that only Copper can handle is when the semicolons are not after the last statement in a block. Observe that Oberon-0 is one of those languages that have semicolons between statements instead at the end of them. This language property means that the last statement does not need to end with a semicolon. One test testing this looks like:

```
1  MODULE NextToModule;
2  VAR x, y : INTEGER;
3  BEGIN
4    x := 1;
5    y := 2;
6    IF x < y THEN
```

```
7       STATE s1;
8       x := y
9     ELSIF x # y THEN
10      x := y
11    ELSE
12      x := y
13    END;
14    OBJECT o1 {
15      START = s1;
16      x := y
17    };
18    x := y
19  END NextToModule.
```

Observe that there are no semicolons at the end of the statements on lines 8, 10, 12, 16, and 18. The reason for this is that the scanner changes the state back to the ambiguous state when it matches the SEMICOL token. So when this token is missing the state is remaining in the Oberon-0 state. When the scanner is stuck in the wrong state, it can cause issues since the following lines might obtain the other language. The Copper implementation was able to parse this example, though.

The results seem to show that no tool is mature enough to generate parsers for a combined language where there are no clear separators between them. Nevertheless, it should be noted that context aware scanning seems to be able to parse more cases than implementations with lexical states.

3.2.4 Results for Building ASTs

As stated in the background, generators such as Copper, CUP, and JavaCC all can construct ASTs, either through semantic actions or the use of pre-processors. However, when it comes to ANTLR 4, the tool constructs the parse tree instead of an AST. One could use the visitor or listener pattern to construct an AST, but this solution results in much boilerplate code. To circumvent this, I have implemented code that allows one to use a stack-based approach to build the AST with semantic actions in ANTLR 4.

I have implemented an `ASTBuilder` class that contains a stack. The stack has entries of tuples consisting of an AST node and an ANTLR parser context, which is a node in the parse tree. Before a new node is pushed onto the stack, some nodes already on the stack may be popped and added as children to the new node. Three different categories determine how many nodes are popped:

1. Nodes with a set number of children (this includes all user-specified nodes): will pop off the number of nodes that the new node is supposed to have as children.

2. List nodes: will pop off nodes until it peeks a node that is not in the new node's parser context. Being in the parser context means that the node was created below the new node in the parse tree.

3. Opt nodes: will pop off one node if it is in the new node's parser context.

3.2.5 Results for Ease-of-Use

The LOC count is a measure of easy something is to implement. Table 3.4 presents the LOC of different scanner and parser specifications for the different implementations. As observed, ANTLR 4 has the most concise implementation compared to MetaLexer, which has the most verbose. The results indicate that ANTLR 4 is the easiest to use, and the MetaLexer / CUP combination the most difficult.

	XML Blocks	Mixed Language
ANTLR 4	234	196
Copper	352	227
JavaCC	368	292
JFlex/CUP	411	303
MetaLexer/CUP	473	364

Table 3.4: The Line of Codes for the implementations of the different tools.

3.3 Usage of Tools

Investigating how much a tool is used is interesting since it is correlated to the amount of support available for the tool. To determine this, I have studied how often the tools are mentioned on Stack Overflow, how much they are used on GitHub and if the tool is regularly maintained.

3.3.1 Stack Overflow

People can query Stack Overflow using SQL commands [17]. Furthermore, Stack Overflow posts can have tags. So by querying for the posts that have tags belonging to the parser generator tools, I can determine how many questions are asked about each tool. The result of this querying is shown in Figure 3.1.

As seen, ANTLR 4 is asked about much more than the other tools. Compare this to Copper and MetaLexer that did not have any tag belonging to them. Their relative unpopularity is most likely due to that they are experimental tools implemented for research. Moreover, it also indicates that they are not used much in practice.

3.3.2 GitHub

From GitHub, I was able to collect information by using the website's search function and the dependency graph function. I collected the number of code and repositories hits I got for the searches and also recorded the number of repositories and packages that were dependent on the tools. The system for searching this is by setting the search phrase until the 20 first results were related to the tool in the 20 first search results. This methodology worked for every tool except for CUP. It is estimated that 20% of the search results for CUP is false positives. The

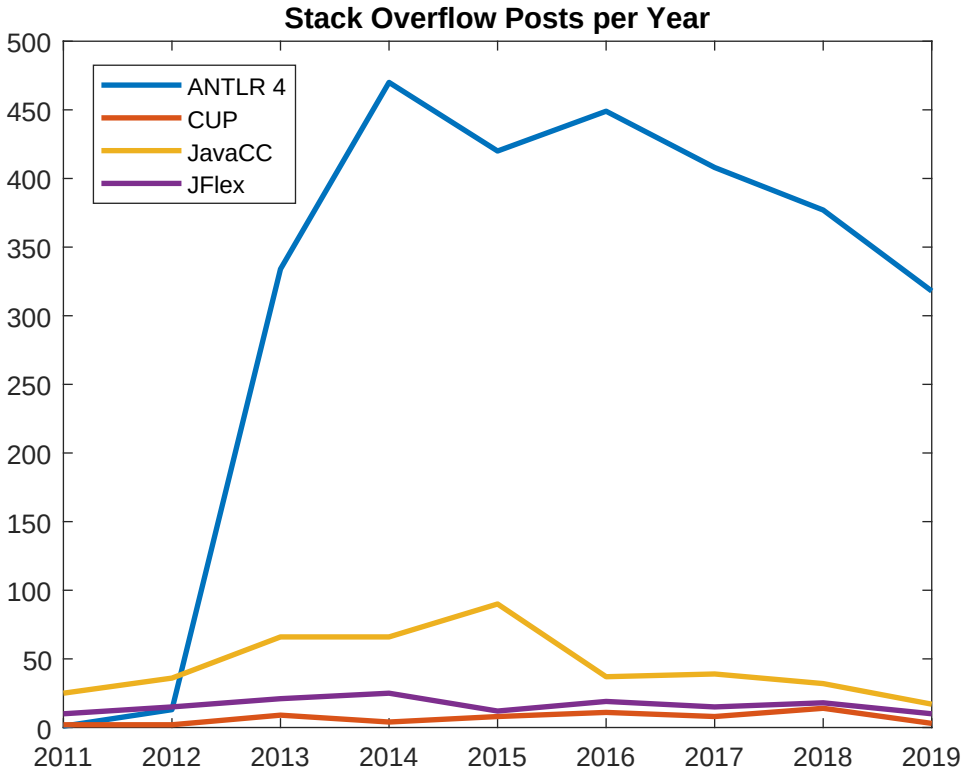


Figure 3.1: Number of Stack Overflow posts per year for tools.

search results are not too reliable as often unrelated projects matched the searching phrase. Thus, I tried to mitigate this issue by filtering out the most common false matches. The results are shown in Table 3.5. Also here, as with the Stack Overflow survey, Copper and MetaLexer gave no indications of use and are thus not shown in the table.

Tool	JavaCC	CUP	JFlex	ANTLR 4
Search Phrase	JavaCC	java_cup NOT cups4j NOT EE NOT IPP NOT Marko NOT Android	jflex	antlr4
Search Hits (repos.)	471	244	285	1 k
Search Hits (code)	393 k	129 k	158 k	446 k
Dependents (repos.)	427	N/A	281	1675
Dependents (packages.)	0	N/A	0	182

Table 3.5: Results of GitHub survey.

Since CUP is not stored on GitHub, the dependency graph function could not be used; therefore, showing N/A in the table. Similarly to the Stack Overflow survey, ANTLR 4 is shown to be the most used tool followed by JavaCC.

3.3.3 Maintenance

Knowing how often a tool is maintained can be crucial in case one has questions or needs a bug to be fixed. I have therefore looked at the repositories of the tools to see how often updates occur:

- **Copper:** The latest release was in 2018-11-28 and there does not seem to have been much development since. However, the developers seem to respond and fix the bugs that are reported on GitHub [12].
- **CUP:** The last stable release, 11b, was released on 2016-06-15 and there currently does not seem to be any active development [23]. The maintainer seems to have fixed a few small bugs in the repository after this [28], but nothing major.
- **JFlex:** The scanner generator JFlex seems to be actively developed by a team on GitHub. Their latest release, 1.8.2, being released on 2020-05-03 containing a set of bug fixes [34].
- **MetaLexer:** There has not been any maintenance since the tool was developed. The last release was in 2009-09-12 [22].
- **ANTLR 4:** The ANTLR project is actively maintained by a flourishing community lead by Terrance Parr on GitHub [26]. The latest release was 2020-01-20.
- **JavaCC:** The project has a few active developers working from GitHub [8]. The latest stable release, 7.0.9, was released 2020-06-26 and version 8.0.0 is in the making.

3.3.4 Conclusion

To conclude this survey I classify the use of the tools along four categories: no use, little use, some use, and much use. I also use the same classifiers for the amount of maintenance. The results of these classifications are shown in Table 3.6.

Tool	Amount of Use	Maintenance
ANTLR 4	Much	Much
Copper	Little	Some
CUP	Some	Little
JavaCC	Some	Much
JFlex	Some	Much
MetaLexer	None	None

Table 3.6: Classification of Usage and Maintenance.

3.4 Licenses

All the evaluated tools are Open Source Software (OSS). This allows the tools to be used for free as long as the terms of the tools' licenses are honoured. But, as Schöttle [29] explain, if the licenses are not uphold, the offender can face legal trouble.

The parser generators generate Java files, but some like, for example, ANTLR 4 [25] and Copper [30] also need run-time libraries to operate. The libraries are naturally covered by the license of the tool, but as Kolassa et al. [19] explain the license of the generated code depends on whether it is copied directly from the tool's source code or if it is purely a product based on the specification. Any copied code will be covered by the same license of the tool, while the license of the purely generated code is up for the user to decide.

Of the tools that are evaluated CUP [23], JavaCC [8], MetaLexer [22], and ANTLR 4 [25] use either a BSD license or a BSD-like license . Under this license form the tools can be used in any form, free of charge as long as the copyright notice is present and that the license of the tool is distributed. Copper, on the other hand, is licensed under a LGPL license [12]. This is a more restrictive license. Under the LGPL, the source code needs to be disclosed, any changes to the code needs to be disclosed and derivatives of the tool's source code must be of the same license. Using a run-time library does not make an application into a derivative work [35]. However, one should be careful, as it is not clear whether the files generated from Copper includes copied code and therefore should be covered by LGPL and all that it implies.

3.5 Conclusion

To summarise the results of the general evaluation can be condensed into a table that looks like Table 3.7. The tool specific results are show in Table 3.8.

	ANTLR 4	Copper	JavaCC	JFlex/CUP	MetaLexer/CUP
XML Blocks Test Cases	5/5	5/5	5/5	5/5	5/5
Mixed Language Test Cases	4/6	5/6	4/6	4/6	4/6
XML Blocks LOC	234	352	368	411	473
Mixed Language LOC	196	227	292	303	364
Can connect w/ JastAdd	✓	✓	✓	✓	✓

Table 3.7: Summary of results for implementations.

	ANTLR 4	Copper	CUP	JavaCC	JFlex	MetaLexer
Amount of use	Much	Little	Some	Some	Some	None
Maintenance	Much	Some	Little	Much	Much	None
License	BSD	LGPL	BSD	BSD	BSD	BSD

Table 3.8: Summary of results for tools.

The results indicate that all tools can parse XML Blocks, i.e. a language where there are border known transitions between the languages. For, the Mixed Language it is more complicated, here no language seems to have all the desired functionality, but the Copper shows the most promise. The investigation has also shown that all tools can connect with other code generation tools, for example, JastAdd.

Chapter 4
Industrial Case Study

Based on the results from the general evaluation, my supervisors at ABB were most interested in the tools ANTLR 4 and JavaCC. The combined language that they wanted to parse is quite similar to XML Blocks; however, instead of Oberon-0 and SML, there are blocks of IEC 61131-3 [16] languages such as Structured Text. To choose between the two parser generators, they wanted to test the parsers along different dimensions. Other than the aspects explored in the general evaluation, they were interested in how one might customise error messages and how to change the row number in the error messages. Furthermore, they also wanted a performance evaluation of the tools in the performance comparison. Based on the result of these surveys, I come with a recommendation with the tool most suited for ABB.

4.1 Custom Error Messages

Since ABB's XML language mix is only constructed in the back-end, the actual users of their platform only concern themselves with the actual content of the code blocks. For example, they might only see a Structured Text block. This causes a few complications when dealing with error messages from the parser. If there is a syntax error in the code block, this will be noticed by the parser and it will return an error message. This error message has an accompanying line number where the error occurs. However, since it is the XML file that is parsed it will be the line number in the XML file that is returned. This is not the same as in the code block. Thus, I will investigate how to change the error messages and change the error line number to make it more appropriate for the end user.

JavaCC was difficult to change the error messages. JavaCC throws ParseExceptions when errors are found. The way these ParseExceptions are thrown are via a method called `generateParseException()`. I override this method in by constructing a subclass of the parser class. If the error is inside a code block I change the message of the exception. I can determine this by utilising a boolean variable in the parser `isInCodeBlock`. I update the status of `isInCodeBlock` with semantic actions in the grammar. In the new messages I just change

the line numbers.

ANTLR 4 has more of a framework for customising errors. The scanners and parsers have error listeners that one can define and add to the parsers. So I have defined my own. The difference from the regular error listener is that it changes the line numbers to be relative the code block. To know when to do this I utilise a `isInCodeBlock` variable similarly to JavaCC.

Ideally you also want the new line numbers to be in semantic errors as well. Semantic errors are handled with JastAdd. The line numbers to these errors are transferred from the parser into JastAdd. I have thus added one line of code to give return the line number relative to the code block instead. This is done as a JastAdd attribute.

Using these implementations both tools are able to give custom error messages. However, even though both give good result, the implementation was more straight forward in ANTLR. In case the implementations would be needed to be updated or so forth, ANTLR would probably have the best way to do it.

4.2 Performance Test

To determine the performance of the implementations, I have conducted both a start-up and steady state evaluation. The two methodologies measure slightly different things. When running an invocation of the Java Virtual Machine (JVM) there are several factors that impact run-time. Such factors can be Just-In-Time (JIT) compilation and class-loading. As Georges et al. [11] explain, a start-up measurement is done by measuring the time of one JVM invocation executing a benchmark once. This type of measurement includes JIT compilation and class-loading, and is thus suitable for when you want to determine the performance of Java program as a whole. On the other hand, I have also conducted a steady state evaluation. Steady state measurements does not include JIT compilation and class loading. This achieved by running a JVM invocation several times on a benchmark. Only the measurements were steady state has been reached are recorded. This type of measurement is suitable when the JVM will run for a long time, executing the same code repeatedly. As of right now, the way ABB's parser will be used is not certain and therefore it cannot be known whether a start-up or steady state measurement is the most suitable for this case. Thus, both methodologies have been used.

The performance evaluations were performed on a Windows 10 PC with an Intel Core i7-8850H with 2.60 GHz clock frequency, 6 cores and 12 logical processors. The programs were compiled and run using the Adopt Open JDK version 11.0.6 and each JVM invocation was given 1024 MB of RAM. The start-up evaluation was done by determining the mean of 100 JVM invocation times and the steady state by the mean of 50 JVM invocation steady state times, where each invocation parsed the benchmarks 25 times. Moreover, the confidence intervals of the means were calculated using best standard practise as outlined by Georges et al. [11]. In the following subsection I present how the benchmarks were formed and following that I will present the results.

4.2.1 Benchmarks

The benchmarks were composed of the parser tests obtained by team at ABB. To test the impact of file length on performance, four different benchmarks were constructed. These benchmarks are referred to as *All*, *Long*, *Mid*, and *Short*. All encompasses all the test files, Long consists of all files with more than 10000 lines, Mid has files between 1000 and 10000 lines, and Short is formed by the files with less than 1000 lines. The total number of lines and the number of files per benchmark is presented in Table 4.1.

Benchmark	Number of Files	Lines of Code	Size (MB)
All	120	140 k	4.89
Long	1	89 k	3.14
Mid	7	39 k	1.37
Short	112	12 k	0.37

Table 4.1: Number of files, lines of code, and size per benchmark.

4.2.2 Results

The results of the start-up evaluation is shown in Figure 4.1 and the result of the steady state evaluation in Figure 4.2.

As seen JavaCC is performing better than ANTLR 4 in the start-up cases for all benchmarks. The difference in execution time differ with what benchmark was parsed. The difference is the largest for the Mid benchmark and the lowest for the Long benchmark. This could be due to that ANTLR 4 does caching in run-time to improve execution time [27]. It is possible that this will have a larger impact when parsing longer files. When it comes to the steady state there is no significant difference between the two implementations.

4.3 Recommendation

Even though the performance of JavaCC seems to be better than ANTLR, all other things point to that ANTLR 4 is superior than JavaCC, the amount available support, and ease-of-use, also the way to alter the error messages. Therefore, I recommend that ABB use ANTLR 4 as their new parser generator.

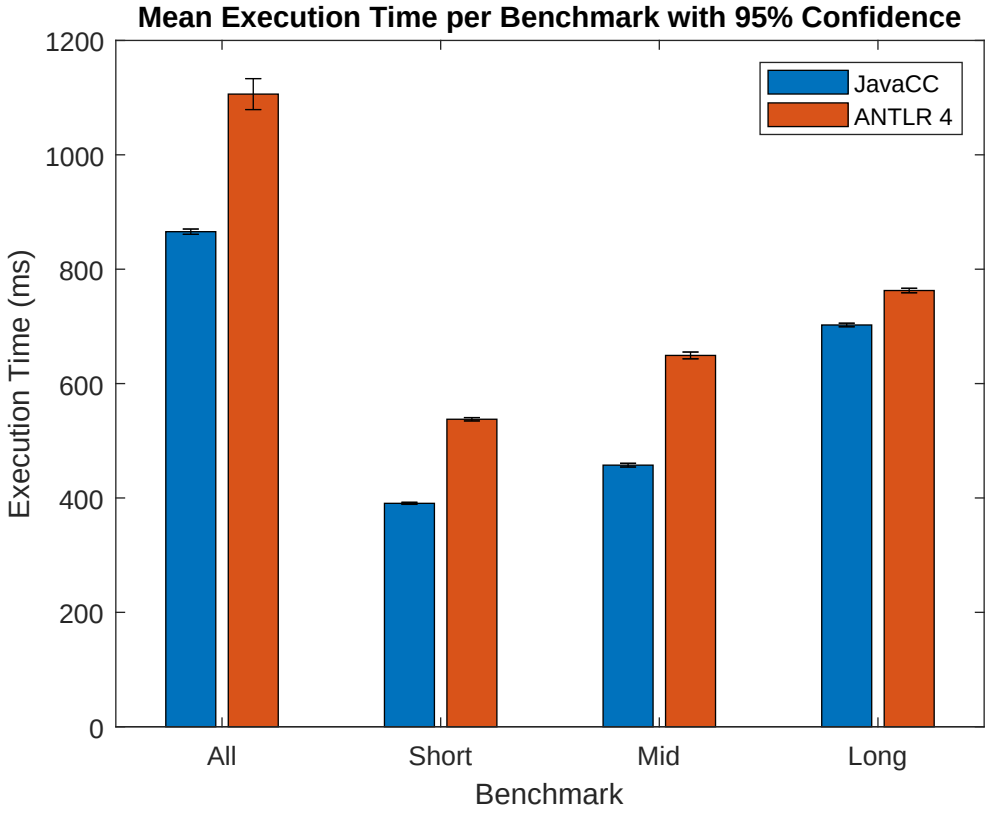


Figure 4.1: Mean start-up performance for implementations per benchmark with 95% confidence intervals.

Figure 4.2: Mean steady state performance for implementations per benchmark with 95% confidence intervals.

Chapter 5
Discussion

In this chapter I discuss potential limitations to the evaluation in 5.1 Limitations, compare the results to any related works in 5.2 Related Works, and also discuss plans for future works in 5.3 Future Works.

5.1 Limitations

To improve the evaluation, I could have evaluated more tools. There are more tools available to examine, such as SableCC [9] and Rats! [1]. The reason why the tool selection was limited as it was due to time restrictions; notwithstanding, including more tools would provide a more complete picture of available tools. Furthermore, more experimental techniques that could have been explored, for example, generalised parsing methods such as GLL and GLR, PEG parsers and even functional parser combinators.

5.2 Related Work

The authors of ANTLR 4 [27] have also conducted a steady state performance evaluation on several parser generators. Their results differ from the results of the performance evaluation done in this report. They show that ANTLR 4 outperforms JavaCC. Reasons for this difference could be that the authors parse Java grammars and utilise more extensive benchmarks. The results also indicate that parser generators that utilise generalised parser algorithms such as GLL and GLR perform much worse compared to conventional parser generators such as JavaCC and ANTLR 4.

Bravenboer and Visser [5] have spent time on researching how to best embedded languages into others. They have mostly focused on embedding domain-specific languages into general-purpose languages. The authors recommend a scannerless approach to parsing combined languages. Scannerless means that there is no scanner in the compiler pipeline. Instead,

so are all characters in the text to be parsed counted as terminals in the grammars. The scannerless approach is essential according to them since it avoids all cases of lexical ambiguities when for example, one keyword in one grammar is supposed to be an identifier in another as experienced in the general evaluation. Furthermore, the authors argue that generalised parsing algorithms GLR is also vital to use. This is because different components of a combined grammar might be implemented in different subsets of CFG, for example, LL or LR. However, this does not seem to be the main factors to why the Mixed Language was difficult to parse but rather the lexical disambiguation aspect. One should also note that as Terrence Parr has shown GLR parsers seem to be slower than regular parser generators [27].

Other techniques can be useful for parsing multiple languages. One such way could be parsing combinators as this is a technique that allows for context-sensitivity [31]. Parsing combinators allow the user to write parsers that look like grammars without having to use external parser generators such as ANTLR 4 and JavaCC. Parser generators are often implemented in functional languages, for example, Haskell. Like regular parser generators, one can use CFGs. However, using parsing combinators, one can use the result of one parser combinator to construct another parsing combinator in runtime, which thus allows for context sensitivity. One such example would be Kurs et al. [20] that have implemented a PEG parser combinator that allows parsing of context-sensitive parsing for things such as XML using top-down parsers. Their implementations utilise parsing contexts as inputs to the parsing functions. This implementation allows for context-sensitive non-terminals to be defined.

5.3 Future Works

In the future, a formalism regarding lexical states needs to be established. In the relevant literature, on compiler construction, [2, 38, 3], the forms of grammars that can be parsed with lexical states are not explored. One needs to establish precisely when the lexical state method can be used with full functionality. The rule of thumb, when there are specific separators use lexical states, work fine; however, methods similar to the ambiguous state implementations need to be fully fleshed out.

This work can form as a basis to produce a framework which people can use to determine which tool might be best used for them based on the combined grammar they want to parse. This could include step-by-step with criteria to fulfil.

Chapter 6
Conclusion

In this chapter, I answer how well the goals of this thesis have been answered. The first goal was to determine parser generators suitable for parsing combined grammars from an industrial perspective. As seen by the evaluations, several tools provide sufficient functionality if one needs to parse combined grammars where there are clear borders between where the different grammars start and end. These types of languages can easily be parsed with the technique of lexical states. For cases like this, popular parser generators such as JavaCC and ANTLR 4 provide excellent functionality. Furthermore, it should be noted that these more popular tools also have more available support since they are used more than experimental tools. The license they have also is suitable for use in industry. When it comes to more complex combined grammars where there are not clear separators, then there is not any evaluated tool that seems to be able to parse all cases. One could use the experimental tool such as Copper. However, there is almost a complete lack of available support online since such tools are barely used. Furthermore, one such also be careful with Copper due to that its generated code might be having to be licensed as open source.

The second goal was to determine how the parser generators can connect to other tools. This has been tested by building ASTs using JastAdd. All the tested tools can quite easily integrate with JastAdd. This could be done via methods such as semantic actions, or in the case of JavaCC using a pre-processor (JJTree). The tool that seems to have the hardest to work with other tools was ANTLR 4. ANTLR 4 lacked inherent AST building functionality, instead of relying on just constructing the parse tree. One could use the visitor or listener pattern; however, this would result in a lot of boilerplate code. To circumvent this, one can add some extra code that allows one to construct a parse tree via semantic actions.

References

[1] Rats! https://cs.nyu.edu/rgrimm/xtc/rats-intro.html. Accessed: 2020-08-02.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, & tools*. Pearson Addison-Wesley, Boston, MA, USA, 2 edition, 2007.

[3] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in Java*. Cambridge University Press, 2002.

[4] Elliot Joel Barker and C. Scott Ananian. Jlex: A lexical analyzer generator for java(tm). https://www.cs.princeton.edu/~appel/modern/java/JLex/, 2 2003. Accessed: 2020-07-18.

[5] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, page 365–383, New York, NY, USA, 2004. Association for Computing Machinery.

[6] Andrew Casey and Laurie Hendren. Metalexer: A modular lexical specification language. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, AOSD '11, page 7–18, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Andrew Michael Casey. The MetaLexer Lexer Specification Language. Master's thesis, McGill University, Montréal, Canada, 2009.

[8] JavaCC Community. Javacc. https://javacc.github.io/javacc/, 2020. Accessed: 2020-07-28.

[9] SableCC Community. Welcome to the sablecc project. http://sablecc.org/, 7 2020. Accessed: 2020-07-06.

[10] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[11] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.

[12] The MELT Research Group. Copper. `https://github.com/melt-umn/copper`, 12 2019. Accessed: 2020-03-31.

[13] Jan L. Harrington. 15 - embedded sql. In Jan L. Harrington, editor, *SQL Clearly Explained (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 301 – 322. Morgan Kaufmann, Boston, third edition edition, 2010.

[14] Görel Hedin. An introductory tutorial on jastadd attribute grammars. In *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, GTTSE'09, page 166–200, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Görel Hedin and Eva Magnusson. Jastadd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37 – 58, 2003. Special Issue on Language Descriptions, Tools and Applications (L DTA'01).

[16] International Electrotechnical Commision (IEC). Programable controllers – part 3: Programming languages. Technical Report IEC 61121-3, 2013.

[17] Stack Exchange Inc. Stack exchange data explorer. `https://data.stackexchange.com/stackoverflow/query/new`, 7 2020. Accessed: 2020-07-28.

[18] Filip Johansson. Parserevaluation-exjobb. https://bitbucket.org/jastadd/parserevaluation-exjobb, 05 2020. Accessed: 2020-08-04.

[19] Carsten Kolassa and Bernhard Rumpe. The influence of the generator's license on generated artifacts. *CEUR Workshop Proceedings*, 1290, 12 2014.

[20] Jan Kurs, Mircea Lungu, and Oscar Nierstrasz. Top-down parsing with parsing contexts. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014.

[21] M. E. Lesk and E. Schmidt. *Lex - A Lexical Analyzer Generator*. Bell Laboratories, Murray Hill, New Jersey 07974, 7 1975.

[22] Sable McGill. Metalexer. `http://www.sable.mcgill.ca/metalexer/`, 09 2009. Accessed: 2020-05-13.

[23] Technische Universität München. Cup 0.11b. `http://www2.cs.tum.edu/projects/cup`, 7 2020. Accessed: 2020-07-06.

[24] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh, NC, 2013.

[25] Terence Parr. Antlr. `https://www.antlr.org/`, 2014. Accessed: 2020-07-01.

[26] Terence Parr. Antlr v4. `https://github.com/antlr/antlr4`, 07 2020. Accessed: 2020-07-28.

[27] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll(*) parsing: The power of dynamic analysis. *SIGPLAN Not.*, 49(10):579–598, October 2014.

[28] Michael Petter. cup. `https://versioncontrolseidl.in.tum.de/parsergenerators/cup`, 3 2019. Accessed: 2020-07-28.

[29] H. Schoettle. Open source license compliance-why and how? *Computer*, 52(8):63–67, 2019.

[30] August Schwerdfeger. Copper user manual version 0.8. `https://github.com/melt-umn/copper/blob/develop/doc/manual/CopperUserManual.md`, 7 2019. Accessed: 2020-03-31.

[31] S. Doaitse Swierstra. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, GPCE '07, pages 252–300, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[32] ANTLR Team. Antlr 4 documentation. `https://github.com/antlr/antlr4/blob/master/doc/index.md`, 2 2020. Accessed: 2020-03-24.

[33] JastAdd Team. jastadd. `https://jastadd.org/web/`, 2011. Accessed: 2020-07-01.

[34] JFlex Team. Jflex. `https://www.jflex.de/`, 7 2020. Accessed: 2020-07-06.

[35] David Turner. The lgpl and java. `https://www.gnu.org/licenses/lgpl-java.html`, 4 2014. Accessed: 2020-07-30.

[36] Mark van den Brand. Introduction—the ldta tool challenge. *Science of Computer Programming*, 114:1 – 6, 2015. LDTA (Language Descriptions, Tools, and Applications) Tool Challenge.

[37] Eric R. Van Wyk and August C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, page 63–72, New York, NY, USA, 2007. Association for Computing Machinery.

[38] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, Zürich, 6 1996. The slightly revised version of 2005.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-08-27

EXAMENSARBETE Evaluation of Parser Generators for Combined Grammars
STUDENT Filip Johansson
HANDLEDARE Görel Hedin (LTH), Stefan Sällberg (ABB)
EXAMINATOR Niklas Fors (LTH)

Parsergeneratorer för Kombinerade Programmeringsspråk

POPULÄRVETENSKAPLIG SAMMANFATTNING **Filip Johansson**

Parsergeneratorer är verktyg som generar en parser baserat på en grammatik skriven av en användare. Problem kan uppstå ifall man försöker kombinera ett flertal grammatiker som inte passar ihop.

Parsers tar text som input och organiserar den så att innehållet är möjligt att tyda. Detta är ett av de första stegen i en kompilator och därför väldigt viktigt. Parsers kan relativt snabbt och enkelt tillverkas med så kallade parsergeneratorer. Dessa gör det oftast överflödigt med handskrivna parsers. Parsergeneratorer kommer i olika typer och kan använda ett flertal olika algoritmer. Dock så tar de flesta en kontextfri grammatik som specifikation till parsern den generar. Grammatikerna kan visa sig att vara problematiska ifall man skulle vilja kombinera två eller fler språk. Man skulle till exempel vilja införa SQL kod i ett mer universellt språk som Java eller Python för att användas i databassammanhang.

I mitt examensarbete har jag utvärderat ett flertal parsergeneratorer utifrån ett industriellt perspektiv för att se ifall verktygen är mogna nog för att hantera kombinerade språk. Jag har testat parsergeneratorer som är både välanvända av många och verktyg med mer experimentell status. Verktygen har utvärderats enligt dimensioner som funktionalitet av att kunna parsa sammansatta språk, typ av licens, hur mycket verktygen används, hur enkla de är att använda, hur enkelt man kan ändra på felmeddelanden, samt prestanda.

För att testa parsergeneratorerna har jag konstruerat två testspråk. Genom att implementera parsers för dessa testspråk har jag kunnat komma fram till deras funktionalitet. Jag har även utfört ett industriellt case study. För att mäta hur mycket verktygen används så har jag undersökt GitHub och Stack Overflow. Sedan för att mäta hur enkla verktygen är att använda har jag mätt antalet rader för implementationsspecifikationerna. Till sist har jag även undersökt hur generatorerna kan kopplas med andra verktyg genom att granska hur enkelt det är att koppla dem till verktyget JastAdd.

Mina resultat visar att alla av de testade verktygen klarar enkelt av att parser sammansatta språk där det finns klara gränser mellan de olika språken. Dock, verkar det vara svårare i kombinationer där det inte finns någon sådan gräns. Det finns fall som inget av de testade verktygen klarar av att parsa men de experimentella verktygen verkar kunna parsa fler fall. Resultaten kan användas av personer som själva ska implementera ett kombinerat programmeringsspråk för att välja rätt parsergenerator.