Informatics

Parsing von Konfigurationsdateien

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

René Schwaiger, BSc.
Matrikelnummer 0425176

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam
Mitwirkung: Senior Lecturer Dipl.-Ing. Dr.techn. Markus Raab, BSc.

Wien, 27. November 2019

René Schwaiger	Franz Puntigam

Informatics

Parsing of Configuration Files

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

René Schwaiger, BSc.
Registration Number 0425176

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam
Assistance: Senior Lecturer Dipl.-Ing. Dr.techn. Markus Raab, BSc.

Vienna, 27th November, 2019

René Schwaiger Franz Puntigam

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

René Schwaiger, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. November 2019

René Schwaiger

Note: We use hyperlinks quite heavily in this thesis. This makes it easier for you to quickly jump between relevant parts of the text and allows you to access various linked online resources. Please consider this advantage before you print this text.

Acknowledgements

I would like to thank my family for their great support, especially my brother, who always encouraged me, even when this thesis did not progress in the way it should.

A big thank you to my supervisor Markus Raab, who was always there, when I needed a helping hand. His kind words and support were an integral part of the process of writing and finishing this thesis. I would also like to thank Franz Puntigam for giving me the chance to work on this interesting research topic and for his thorough reviews of the text.

Thank you to all the people who helped me in the preparation and implementation phase of the thesis. I greatly appreciate the work of the other Elektra developers, be it documentation text or support code. Your work was integral in helping me to implement the parsing code. A special thank to all the Elektra developers that took part in the discussion about YAML features. I also like to thank the communities of PEGTL and YAEP, which implemented features and bug fixes that eased the implementation phase of the thesis considerably.

Last but not least, I would like to thank my friend Johannes Eder for his input in the implementation phase, debugging assistance, and for keeping me company while I worked on this thesis.

Kurzfassung

Nahezu jede Applikation verwendet Konfigurationseinstellungen um ihr Verhalten einem spezifischen Kontext anzupassen. In immer komplexer werdenden Computersystemen kann die große Anzahl verschiedener Einstellungen leicht zu Fehlkonfigurationen führen. Die verschiedenen Möglichkeiten Softwaresysteme zu konfigurieren und die große Anzahl verschiedener Konfigurationsformate verbessert diese Situation sicherlich nicht. Elektra ist ein auf Plugins basierendes Konfigurationsframework, das Konfigurationseinstellungen in eine globale hierarchische Datenbank integriert und eine einheitliche Schnittstelle für diese anbietet um Fehlkonfigurationen zu vermeiden.

Damit Elektra ohne oder nur mit geringen Modifikationen an einer Applikation arbeiten kann, muss es das Konfigurationsformat der Applikation verstehen. Um das zu bewerkstelligen verwendet Elektra ein auf Plugins basierendes System um Konfigurationsdateien in seine internen Datenstrukturen umzuwandeln. Veränderungen der internen Datenstrukturen werden dann zurückgeschrieben, um so die Konfiguration einer Applikation über Elektras Schnittstelle zu ändern. Nachdem eine Vielzahl von verschiedenen Konfigurationsformaten existiert, müssen wir uns Gedanken machen, wie wir diese mit möglichst geringem Aufwand syntaktisch analysieren, also parsen, können.

Um zu untersuchen, welches System die beste Lösung für die Aufgabe des Parsens von Konfigurationsdateien liefert, analysieren wir zunächst erfolgversprechende Parsing-Techniken in einer Literaturrecherche. Als Beispielsyntax für weitere Untersuchungen verwenden wir ein Subset der populären Sprache YAML, das wir ursprünglich mit Hilfe der Daten einer Befragung von einigen Elektra-Entwicklern ermitteln wollten. Da das gewählte Subset aus Sicht der Syntaxanalyse uninteressante Eigenschaften mit hohem Entwicklungsaufwand aufweist und wegen Problemen mit der Aussagekraft der Umfrage spezifizieren wir schließlich das YAML-Subset selbst. Wir schreiben, generieren und integrieren Parsing-Code für 8 neue Elektra-Plugins und erweitern eines der existierenden Plugins. Schließlich vergleichen wir fünf der Parser-Plugins in einer detaillierten Untersuchung.

Der Vergleich der Features der Parsersysteme zeigt, dass für unsere Beispielsprache ANTLR den größten Funktionsumfang und gute Fehlermeldungen liefert, ohne Grammatikänderungen zu benötigen. Während der Code des Generators weder die schnellste Laufzeit, noch den geringsten Speicherverbrauch aufweist, zeigen unsere Messungen eine angemessene Leistung. Um 10 000 Zeilen YAML zu konvertieren benötigt der Parser ungefähr 120ms auf unserem Testsystem.

Abstract

Almost any application uses configuration settings to adapt its behavior to specific contexts. In ever growing complex computer systems, the large amount of different settings for different software can easily lead to misconfiguration. The different methods to configure systems and the large amount of different configuration file formats does certainly not improve the situation. Elektra is a plugin-based configuration framework that integrates configuration settings into a global hierarchical database and provides a common interface for configuration settings to make misconfiguration less likely.

For Elektra to work without, or with only minor modifications to an application it needs to understand the configuration file format of said application. To do that Elektra uses a plugin-based system to parse configuration files and convert them into its native data structures. Changes to these data structures are then written back, effectively changing the configuration of an application via Elektra's interface. Since there exists a plethora of different configuration file formats we need to think how we can parse them with as little effort as possible.

To evaluate which parsing system offers the best fit for the task of configuration file parsing, we first study promising techniques in a detailed literature research. As example syntax for further analysis we use a subset of the popular language YAML. Originally we wanted to determine this subset using data from a survey we conduct with some of Elektra's developers. Since the chosen subset includes items with high development effort that are not interesting from a parsing point of view, and because of problems with the conclusiveness of the survey, we choose the implemented subset ourselves. We write, generate and integrate parsing code, creating 8 new Elektra plugins, and extend one of Elektra's existing plugins in the process. We then compare five of our parser plugins in a detailed evaluation.

The comparison of the various features of the parsing systems shows that for our example syntax ANTLR provides the most complete feature set and good error messages without requiring any changes to the grammar. While the parser generated by ANTLR was neither the fastest parser, nor the parser with the lowest memory overhead, the benchmarks in our comparison still show acceptable performance for our example data. To convert 10 000 lines of YAML the parser needs roughly 120ms on our test system.

Contents

CHAPTER 1

Introduction

1.1 Motivation & Problem Statement

Parsing of programming languages is an often studied and revered technique in the computer sciences. The same is not true for configuration file parsing. The answer to the question “How do I solve this problem?” does usually not contain the task of creating and parsing a new programming language in the process. On the other hand, the answer to “How do I configure this?” often seems to be thinking about a new configuration file format and then implementing handwritten parser code for it.

While handwritten parsing code can be the proper tool to convert code, for example both Clang [Ben12] and GCC [Mye08] use handwritten parsers, often the person writing configuration parsing code is not an expert in parsing. As consequence, configuration file parsers often suffer from problems such as

- bad or no error message support,
- no error recovery, and
- no proper encoding support.

One of the reasons for these problems is that computer programmers usually think of configuration file parsing as easy and not that important. After all, misconfiguration is also often seen as a problem of the user and not of the application [Xu+13].

In this thesis we will look at the configuration file parsing problem and evaluate different options to convert configuration settings. Notable examples include:

- bidirectional programming [Fos+05; BPV06; Lut08; KZH16; Raa16],

- code produced by a parser generator [DM08; PHF14; WDG16; RB17],
- serialization libraries [SM12; Pac+15], and
- handwritten parsers [Mye08; Ben12].

Currently the possibilities to compare different parsing techniques are limited. The naïve approach would be to just run different parsers on the same data. In practice however, this approach is not usable, since parser tools tend to produce very different data structures. Some of them do not produce data structures at all, instead they let the user specify subroutines that should be called when the parser matches parts of the grammar.

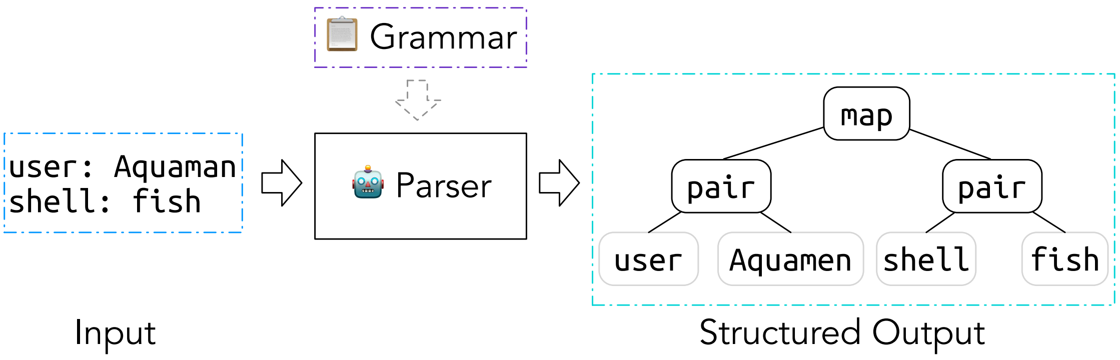


Figure 1.1: Simplified view of a parsing process

As part of this thesis we will tackle this problem, using different parsing techniques within a common configuration framework. This integration eliminates the problem of comparing the parsing process under different circumstances, since the data structures the parsers create will always be the same. We will use Elektra, a key-value database, as configuration framework. Elektra's storage plugin interface will act as foundation for the parsing process. In the end the thesis should provide answers about which parsing techniques provide an ideal balance between performance and usability.

1.2 Aim of the Work

Elektra [Raa10; Raa17] is a plugin-based framework that stores configuration parameters in a Key Database (KDB). Elektra reads and stores configuration data via so-called *storage plugins*.

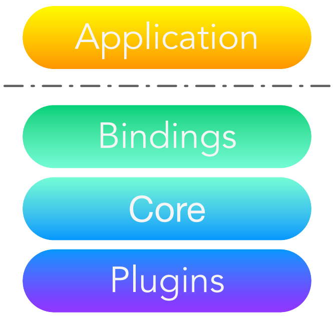


Figure 1.2: The diagram above shows an architecture overview of Elektra. Plugins, are among other things, responsible for parsing and writing configuration data. The core is written in C and provides a low-level Application Programming Interface (API) to access configuration settings. Bindings provide higher-level access to configuration data in C and other languages, such as Java, Lua, Python and Ruby.

As part of this thesis we compare various ways of parsing. For that purpose we wrote and generated parsing code for different storage plugins. All of these storage plugins parse a minimal subset of YAML, a human readable language used to specify data. We looked at the following state of the art parsing technologies:

- handwritten recursive descent parser (yaml-cpp),
- ALL(*) parser generator (ANTLR),
- LR parser generator (Bison),
- Earley parser (YAEP),
- PEG parser (PEGTL),
- parser combinator (mpc), and
- bidirectional programming (Augeas).

We compare the parsing code according to the following criteria:

- runtime performance,
- memory usage,
- code size,
- overall code complexity,
- ease of extensibility and composability, and

- error reporting.

In the scope of the above comparison we answer the following main research question.

? Main Research Question. *Which parsing systems allows us to create a configuration parsing plugin that is easily extendable, has low maintainability cost and provides good error messages, while offering decent runtime performance and low memory overhead.*

To answer the main research question we will first look at the auxiliary research questions below.

? RQ 1. *How does the theoretic runtime complexity of the parsing methods compare to the actual measured runtime of the parsing code?*

? RQ 2. *How does the peak memory usage of the algorithms compare to each other? Do some of the algorithms show nonlinear memory usage?*

? RQ 3. *How much work does it require to implement the plugins, i.e. how many lines of code do we have to write to support our YAML subset for each parsing engine? How do the amounts of handwritten code for the plugins compare to each other?*

? RQ 4. *Which parsing technique allows us to stay closest to the definition of the configuration language? Does staying close to the given definition allow us to extend and improve the parser and its support code more easily?*

? RQ 5. *What are the error handling capabilities of the parsing engines? How well can they handle multiple syntax errors? How do the generated error messages compare to each other?*

1.3 Methodological Approach

The methodological approach for this thesis consists of the steps given below.

Literature Review We determine the current status of parsing techniques suitable for configuration file parsing. We then choose appropriate libraries for the parsing techniques listed in the section “Aim of the Work”.

Discussion To determine a minimal usable subset of YAML we decide about common features required for a new Elektra storage plugin with some of the current developers. For that purpose we demonstrate YAML features and ask, which ones should be supported in a survey.

Implementation We write, generate and integrate parsing code that handles our minimal YAML subset. In this phase we also add other necessary support code to Elektra.

Comparison As noted in "Aim of the Work" we evaluate the different implementations of our minimal YAML subset parsers.

Runtime Benchmark: For the runtime comparison we create a benchmark framework to determine the speed of the different parsing code. In this part of the thesis, we answer RQ 1.

Memory Profiling: For the memory comparison we use a memory profiling tool to determine the heap memory usage of the YAML plugins and answer RQ 2.

Code Count: We count the number of code lines with a code line counting tool. This method allows us to consider only actual code, ignoring blank lines and comments. At the end of this task we answer RQ 3.

Complexity Measurement: We measure the Cyclomatic Complexity (CC) of the code using a static analyzer.

Extensibility & Composability Check: To analyze the extensibility and composability of the parsing code we look at the code difference of commits for certain features and bug fixes. We count the amount of updated code lines to determine the extension effort. This measurement, together with a comparison between the grammar specification of YAML and the code created in this thesis, helps us to determine the answer for RQ 4.

Error Reporting: To determine the quality of the error messages we create erroneous files and compare the quality of the resulting error output. The purpose of this task is to answer RQ 5.

1.4 Contributions

Parsing Engine Integration We added plugins for four general purpose parsing engines (ANTLR, Bison, YAEP, PEGTL) to the Elektra framework. These plugins parse a subset of YAML, and can be adapted to parse other configuration languages.

Benchmark Framework We created a parser benchmark framework. This framework compares configuration parsers under fair circumstances, since we require and verify that the parsers produce the same data for the same YAML input.

Support Plugins In this work we

- created a plugin that adds full support for mapping node-based configurations formats, such as JavaScript Object Notation (JSON) and YAML, to Elektra's data structures,
- extended a plugin to convert Base64 encoded YAML data into binary Elektra data, and
- created a plugin that deserializes Elektra's data structures into YAML data.

Free/Libre and Open-Source Software (FLOSS) Contributions During the work of this thesis the author

- reported over 100 problems in Elektra's issue tracker,
- reviewed over 80 pull requests,
- committed over 4 000 times to Elektra,
- opened over 370 merged pull requests, and
- added over 100 000 lines to, and removed over 110 000 lines from Elektra's code base.

1.5 Structure of this Thesis

In chapter 2 we provide background information about the state of art in configuration file parsing. We then describe the configuration framework Elektra, and give some examples about other parser comparison related research.

In chapter 3 we describe the design challenges we faced and the decisions we made when we implemented the parser plugins. First we describe how we decided about the YAML subset the parsers should be able to handle. Then we explain the mapping between YAML data and the data structures of Elektra. After that we discuss the problems we had when we implemented the plugins and how we solved them. At the end of the chapter we talk about the additional Elektra plugins we wrote to improve the integration of the parser plugins into Elektra.

Chapter 4 provides a detailed evaluation of the parsers. It first describes the goals of the evaluation, detailing different important criteria of the parser plugins. Afterwards we analyze each of the criteria in their own section, answering our research questions in the process. At the end of the evaluation we describe the parser plugins that best fit our criteria.

In chapter 5 we conclude the thesis with an overview of the results and describe possible future work.

CHAPTER 2

Background

In the first part of the thesis we look at the current state of configuration file parsing according to the literature. After that we provide a short overview of Elektra, the key-value configuration framework we use in the thesis. At the end of this chapter we provide some examples of other research papers that evaluate parsers and parser engines.

2.1 State of the Art

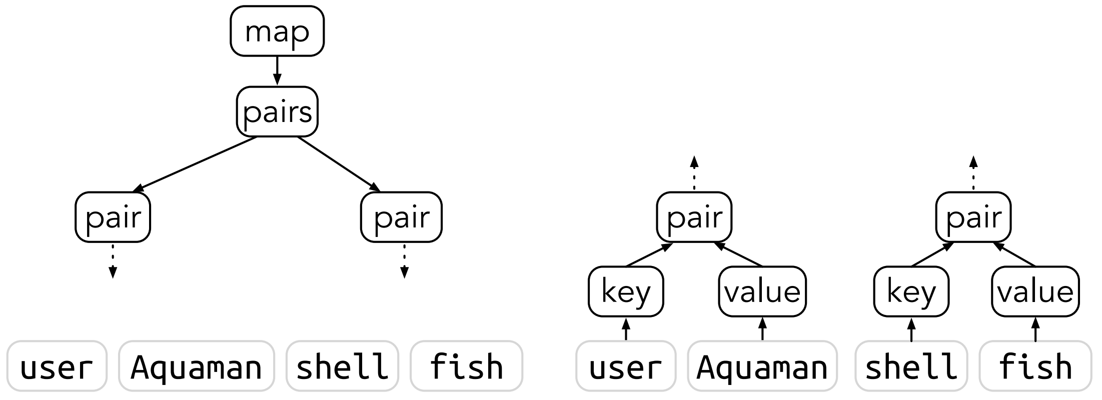
2.1.1 Parsing

The book "Parsing Techniques" [GJ08] provides a good overview of various up-to-date parsing algorithms. It covers the most popular techniques and also less well known methods up to 2006. The book also describes various classification possibilities for parsing techniques [GJ08, p. 85]. The most common classification is the division into bottom-up and top-down parsers.

In *top-down parsing* the parser starts with a hypothesis about the structure of the given data. The parser then tries to predict and match parts of the structure, starting from larger parts working its way down to smaller elements.

We can further categorize parsing into *directional* and *non-directional* methods. Directional methods read the input from left to right, while non-directional methods can use an arbitrary order. This implies that directional methods are simpler and faster, but less powerful than their non-directional counterparts. As part of this thesis we only consider directional methods, since they are faster and powerful enough to parse configuration data.

One of the most popular directional top-down methods is *LL parsing*. While this technique is quite old – "Parsing Techniques" (p. 584) mentions a paper from 1961 belonging to the LL



(a) A top-down parser predicts and matches rules from the start symbol downwards.

(b) A bottom-up parser recognizes text starting with the terminal symbols.

Figure 2.1: Matching in top-down and bottom-up parsers

category – it is still actively used and researched. The basic idea behind LL parsing is simple: Begin with the start symbol of the grammar and the first character/token of the text. Then predict the next grammar rule, looking at the text to the right of the current position. We can categorize the technique further according to the number of characters/tokens the parser uses to predict the next rule. If the parser uses one token of look-ahead we speak of an LL(1) parser, if it uses k tokens of lookahead we speak of an LL(k) parser [RS69].

Two common methods to create an LL parser are:

1. Implement the parser code using a set of mutually recursive procedures (recursive descent parser). The code for this is either written by hand or produced by a parser generator such as ANTLR (Another Tool for Language Recognition) [Par13a].

2. Use a parser generator to create a table-based parser.

Examples of popular active projects that use a handwritten recursive descent parser include Clang [Ben12] and GCC [Mye08]. The about page for ANTLR mentions some projects that use its generated recursive descent parsers. The list includes Twitter, which uses ANTLR for query parsing and parsers for the languages used in the Apache Hadoop projects Hive and Pig [Par13b].

Some of the latest research developments in LL parsing include LL(*) parsing [PF11] and its successor Adaptive LL(*) (ALL(*)) [PHF14]. Both of these algorithms use dynamic lookahead [PF11, p. 1]. While LL(*) parsing uses a static algorithm for rule prediction, ALL(*) analyzes the input at runtime to improve prediction. As consequence of this, parsers that use the ALL(*) algorithm will be faster after an initial warm-up phase [PHF14, p. 3]. LL(*) is part of ANTLR 3 [PHF14, p. 3], while ANTLR 4 uses Adaptive LL parsing.

As we already mentioned before, the other popular parsing technique besides top-down-parsing is *bottom-up parsing*. In bottom-up parsing the parser builds a structure starting with the smallest elements of the grammar (terminals). The parser then combines these elements into larger parts. One of the earliest entries in the *linear bottom-up* parser category is the LR(k) parser [Knu65]. Just as in LL(k) parsing, k specifies the number of lookahead symbols the parser uses.

Unlike LL parsers, LR parsers are usually not created by hand, but generated by a parsing tool such as Bison or Yacc. Since LR(k) tables are very large, even for a small numbers of k, the parser tools mentioned before usually generate less powerful but smaller and faster LALR(k) [DeR69] parsers.

LR(k) parsers are able to handle more grammars than LL(k) parsers for the same constant k [Hab13, section "Lookahead"]. However, LR parsers are still not able to handle ambiguous grammars. For this purpose Lang describes the Generalized LR (GLR) [Lan74] method that is also able to process these types of grammars. GLR parsers are sometimes also called Tomita parsers [Tom85] after the author that described the first implementation of a generalized LR parser.

Recent research in the space of directional bottom-up parsing includes improved versions of techniques that are almost as powerful as canonical LR(1). One of the most promising methods is IELR(1) [DM08]. The advantage of IELR(1) over LALR(1) is that it handles conflict resolution better. Parser tools such as Bison use conflict resolution to handle non-LR grammars, i.e. grammars that contain rules where the parser is not able to decide what to do next. To handle these types of conflicts the grammar designer manually specifies which decision the parser should take. The current version of the parsing tool Bison supports an experimental version of IELR(1).

Most parsing techniques can be categorized as either top-down or bottom-up. However, some techniques use a combination of both approaches. Others are usually not listed under one of the label top-down or bottom-up, because they provide other features that the designers of these parsers deem more important, or they use features that do not fit well within either of these groups. In the remainder of this section we will discuss some of these techniques.

A method that can be categorized as either top-down technique with bottom-up recognition, or bottom-up technique with a top-down component [GJ08, p. 206] is *Earley Parsing* [Ear70]. Earley parsing is able to handle any context free grammar. This means the technique is as powerful as GLR parsing. This advantage comes at the cost of runtime. While LL parsing and LR parsing run in linear time depending on the length of the input ($O(n)$), Earley Parsing has an upper boundary of $O(n^3)$. However, in 1991 Leo showed that an improved version of the algorithm handles most LR(k) grammars in linear time [Keg11; Leo91]. In 2002 Aycock and Horspool described improvements to the algorithm [AH02]. Their version of Earley Parsing boosts the runtime in cases where the grammar contains nullable (empty) grammar rules. Recently Kegler incorporated the changes proposed by Leo, Aycock and Horspool in Marpa [Keg11].

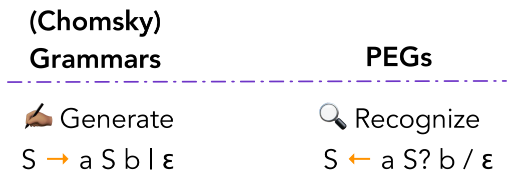


Figure 2.2: Both the Chomsky grammar on the left and the PEG on the right describe the same language $\{a^n b^n | n \geq 0\}$.

All the methods we mentioned until now work with a description that is based on a (context-free) Chomsky grammar. These grammars describe a way to *generate* words and sentences of a given language. Another way to specify the structure of a language is to give a description on how to *recognize* [GJ08, p. 506] words and sentences. One popular recognition system is Parsing Expression Grammars (PEGs). They were introduced by Ford in the paper "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation" [For04]. Ford also describes how to write an efficient (top-down) parser that handles these types of grammars in linear time [For02]. This method, called Packrat parsing, uses a specialized version of memoization [For02, p. 1] to save intermediate results of the parsing process. It is generally not clear, if memoization provides a runtime performance advantage in practice for a certain grammar. In the article "DCGs + Memoing = Packrat Parsing but Is It Worth It?" [BS08] Becket and Somogyi show that not using memoization can decrease the actual runtime of a PEG parser.

Just like Packrat parsing, *combinatory parsing* [Fro92; Hut92] specifies a method to create recursive descent parsers. As the name suggests, the focus in combinatory parsing is the composability of parsers. The technique is usually used in functional programming languages, such as Haskell. These languages support higher order functions, i.e. functions that take other functions as their parameters [GJ08, p. 564]. In combinatory parsing the parser creator typically starts by specifying parsers (functions) for the simplest parts of the grammar (terminals). She or he then goes on to combine these simpler parsers into more powerful parsers for more complex rules (non-terminals). Combinatory parsing has similar problems as other top-down techniques such as LL parsing. One of these problems are left recursive grammar rules, i.e. rules that include references to themselves in the leftmost part of the right-hand side. Recently Frost, Hafiz, and Callaghan described a method to handle left recursive rules in combinatory parsing efficiently in the article "Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars" [FHC07].

A method that is not a parsing technique per se, but a way to specify conversions of data from a source structure to a target structure and back is *bidirectional programming* [Fos+05; BPV06]. The specification that allows this conversion is called a lens [Fos+05].

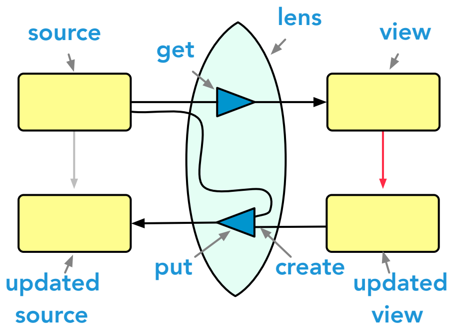


Figure 2.3: Lenses provide a way to both parse (get) and write (put) structured data (Source: Boomerang Programmer's Manual).

A programming language used to specify such lenses is Boomerang [Boh+08]. The research of the Boomerang project lead to the creation of another project that uses lenses to parse configuration data: Augeas. Augeas converts configuration data into a tree like representation. Berlakovich implemented an Augeas plugin for Elektra as part of his Bachelor thesis [Ber16].

2.1.2 Configuration File Parsing

The literature overview in the previous section focused on parsing in general. It is now time to take a look at the current state of configuration file parsing and how it differs from the well know problem of parsing (general purpose) programming languages.

There exists extensive literature about parsing of programming languages and compiler design. The topic is even part of famous computer science books such as *Principles of Compiler Design* [UA77] and *Compilers: Principles, Techniques, and Tools* [ASU06], commonly also know as "Dragon" books [Par09], named after the Dragon (waiting to be slain) on their covers.

While parsing and compiling computer languages is seen as "slaying a Dragon", configuration file parsing is arguably a much simpler task. The reason for this is that the purpose of configuration languages, usually only specifying data, is a subset of the purpose of programming languages, which also manipulate data. This superset feature of programming languages makes them sometimes also interesting for the purpose of storing configuration data [Bal13]. Since configuration data is used by many programs that usually do not ship with, or require an interpreter or compiler, most configuration files do not use the syntax of a general purpose programming language. These "data only" configuration file formats can be classified roughly into three categories.

Custom configuration file formats specify data for a specific application. Examples are the `fstab` file used by the Unix program `mount` or the `hosts` file used for name resolution.

General purpose configuration formats such as INI, and the Property list file format are used mainly to store configuration data of many different applications.

Serialization formats such as JSON, YAML Ain't Markup Language (YAML), and eXtensible Markup Language (XML), store all kinds of data, not only limited to configuration purposes.

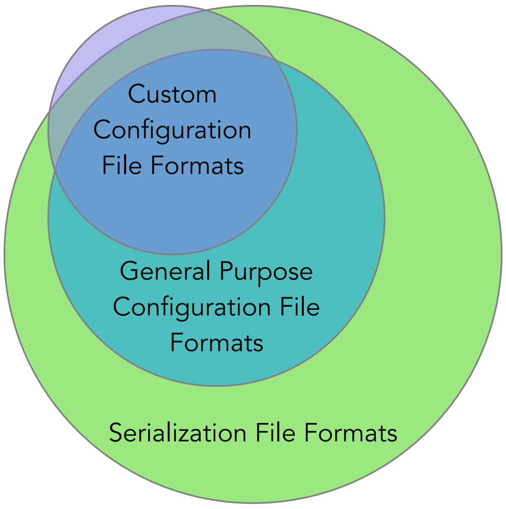

Custom Configuration File Formats
General Purpose Configuration File Formats
Serialization File Formats

Figure 2.4: The Venn diagramm above shows an overview of the overall expressiveness of the formats usually used for configuration data. Please note, that the size of the circles *does not* show the level of expressiveness of a certain category of file formats, i.e. a circle twice as large does not represent twice the level of expressiveness.

In general we can use serialization formats, to specify the same data as we can using general purpose configuration languages. For example, on macOS, Property lists can be stored on disk using JSON or XML. The same is not true for custom configuration formats, which are often very simple, but can sometimes also use programming language code. Popular examples for this category are the configuration files of shells, such as `bash`, `fish` and `zsh`. These shells use the same syntax for programming and configuration data.

We already said that most configuration file formats do not use the syntax of general purpose programming languages, but rather syntax only able to specify and *not* to modify data. The simplicity of some of these languages often means that parsing code for them is not written by an expert. Especially for simple custom configuration file formats, the same person that implements a program often also implements the parser for its configuration file format. This can result in configuration code that integrates tightly into a program, decreasing the extensibility of the application. Other problems of parsers not written by experts include: bad or no error message support, no support for error recovery, and insufficient support for different file format encodings.

Some of these problems can be solved using a configuration file library that is available for most of the more popular general file formats. Such a library takes care of the parsing, loosening the integration of an application and its configuration code. Some problems of theses libraries are that they

- often do not consider order, and
- almost always throw away comments

when they write back data into a configuration file. This is a problems, since for example, comments in configuration files can be very helpful in averting misconfiguration.

A configuration library that keeps the order of entries and comments intact is Augeas [Lut08; Ber16]. As we already wrote in the previous section Augeas uses so-called lenses to both parse and write back configuration. The problem of lenses is that they only support regular languages [Cho59] properly. For example, as we found out during the development of the parsing code for the thesis, the YAML lens of Augeas only supports mapping data with a nesting level of two.

Compared to Augeas, Elektra [Raa10; Raa17], with its plugin-based parsing system also supports context free and even context sensitive configuration languages. This is the case, since Elektra can use a general purpose programming language to parse code. Using metadata Elektra is also able to store the content of comments and keep the order of configuration data intact, when writing data back. The code for this has to be implemented in the plugin itself, which can become problematic. For example, while Elektra's INI plugin preserves comments and the order of entries, its code is also quite complicated and error-prone. As we already stated in "Aim of the Work" we want to improve this situation using state of the art parsing techniques for our YAML subset plugins.

2.1.3 Error Handling

The usual goal in parsing is the processing of grammatically correct input. However, since editing configuration data is often done by hand, there is always a possibility of erroneous input. A method to inform the user about such errors is essential. As pointed out by Terence Parr in *The Definitive ANTLR 4 Reference* [Par13b, p. 151] we also need to inform the user about the reason for an error:

> In other words, a parser that responds with "Eh?" and bails out upon the first syntax error isn't very useful for us during development or for our users during deployment.

The process of reacting to errors is called *error handling*. Error handling can be grouped into four dependent stages [Rüf16; Pot16]:

1. error detection,
2. error diagnosis,
3. error recovery, and
4. error correction.

We are mainly interested in the first three items, since error correction is generally not possible without the possibility of fixing errors incorrectly.

Error reporting is the results of

- detecting that input is incorrect (error detection),
- finding what part of the input was incorrect (error diagnosis), and
- trying to resume the parsing process in case of errors (error recovery).

The current state of error reporting capabilities depends strongly on the parsing technique.

In general, top-down recursive descent parsers *without backtracking*, especially those written by hand, are able to produce good error messages. The reason for this is that such parsers have information about the higher level grammar rules that matched until the error point. While the error messages produced by recursive descent parsers can be good, writing error logic by hand makes the code more complicated. If we want to react to multiple errors in a configuration file we also need to implement error recovery, which can be "tedious and easy to screw up" [Par13b, p. 160]. ANTLR produces parsers that integrate techniques such as token deletion, token insertion, and resynchronization to produce parsers that provide "a good error reporting facility and a sophisticated error recovery strategy for free" according to its main author [Par13b].

Bottom-up parsers produced by a tool such as Bison are not suited to provide good error messages [Jef03]. Error recovery in Bison produced parsers even requires changes to the grammar [DS19]. There exists promising research work to improve the current situation. In "Generating LR Syntax Error Messages from Examples" [Jef03] Jeffery shows how his tool merr can help a user to specify error messages based on example input in a C based parser produced by Yacc or Bison. Jeffery's semi-automated approach is based on error states of the parser and improves on the previous technique used in the Icon programming language, where developers manually modified the source code produced by a custom version of Yacc. Since Jeffery's work looks promising, another similar approach was also used in the old Bison parser[1] for the programming language Go [Cox10]. Recently Pottier [Pot16] extended Jeffery's work and added example based error reporting in the LR parsers of the CompCert C compiler [Käs+18]. In "Reachability and Error Diagnosis in LR (1) Parsers" Pottier [Pot16]

[1]The developers of Go switched to a handwritten parser written in Go in 2016 [Pik16; Pro16].

writes that he thinks that "the quality of CompCert's diagnostic messages is now on par with that of `clang` and `gcc`".

For now we only talked about the error capabilities of deterministic parsers – such as LL and LR parsers – i.e. parsers that do not back up. These parsers read the input deterministically from left to right. They therefore report the first position, where the parsed input is not part of the language described by the grammar anymore. This behavior is also known as (longest) correct/viable prefix property [SS90; Rüf16; Mai+16; Pot16]. Parsers with backtracking, such as recursive descent parsers *with backtracking*, PEG parsers and the closely related combinatory parsers usually do not have this property. While backtracking makes these parsers more powerful, the quality of error messages suffers. The problem is that backtracking can occur both because of a valid choice in the grammar or because of an error in the input.

There has been research to improve the situation, especially in PEG parsers. In his master thesis [For02] Ford already describes one option to produce meaningful error messages. His parser records all parsing results and uses the one that matched the farthest to the right in the input for error messages. In "Error Reporting in Parsing Expression Grammars" [Mai+16] Maidl et al. show that this error strategy can also be added to every PEG library that supports semantic actions. They also introduce a form of error reporting, inspired by the excepting handling mechanism of programming languages, based on grammar annotations called labeled failures [Mai+16].

2.2 Elektra

In this section we describe some of the concepts of Elektra, the software that provides the common storage facility for our YAML parsers, further. Elektra is a framework that stores data in a *global hierarchical key-value database*.

2.2.1 `Key`

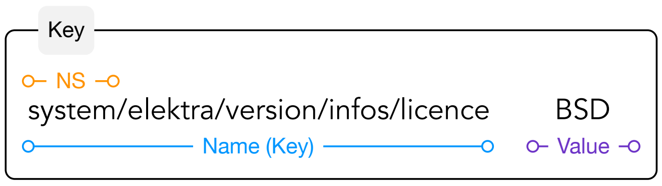


Figure 2.5: In Elektra a `Key` stores a single key-value pair. The first part of the name specifies the namespace (NS).

The most basic entity in Elektra is the `Key` structure. A concrete instance of a `Key` contains at least one non-empty attribute, which is the *name* of the `Key`. In this thesis we will also often use the term *key* – please notice the non-gray background – to refer to the name of a `Key`.

Besides the name an instance of a `Key` usually also stores a value. Figure 2.5 shows an example `Key` with the name `system/elektra/version/infos/licence` and the value `BSD`.

Since Elektra stores data in a hierarchical database, a key consists of parts – separated by `/` – that determine the location in the database. The key in Figure 2.5 consists of 5 parts. The first part `system` is the namespace (NS) of the key. Elektra uses namespaces to specify context dependent data. For example, user specific data is stored under the namespace `user`. Elektra uses 5 namespaces to separate data:

`system`	specifies data values for the whole system,
`user`	contains data for the current user,
`dir`	stores data for the current directory,
`spec`	contains specification of other keys, and
`proc`	stores in-memory data.

We can use a so-called *cascading* keys to select the most appropriate namespace. A cascading key does not start with a namespace but rather with a leading slash. Let us look at an example. We assume our database contains the keys:

- `system/key`, and
- `user/key`.

If a non-superuser requests the cascading key `/key`, then Elektra will select `user/key`. If the database also contains a key `dir/key` for the current working directory, then Elektra will select this key instead.

2.2.2 `KeySet`

As we already saw in the example before, usually we store not only one, but multiple key-value pairs in the database. For this purpose Elektra provides the structure `KeySet`. A `KeySet` contains a set of `Key` objects. The name of each `Key` has to be different. If we add a new `Key` with an already existing name to the `KeySet`, then Elektra will just overwrite the old `Key` with the same name.

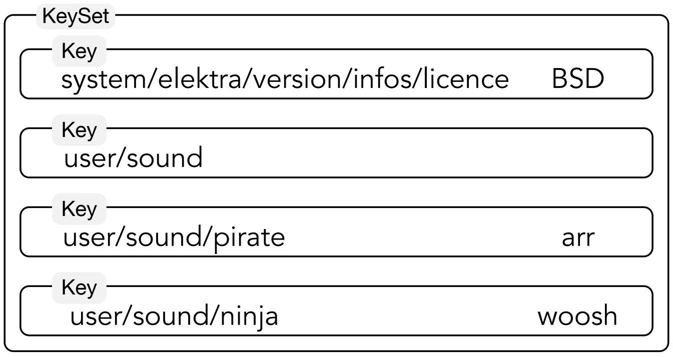


Figure 2.6: Elektra uses `KeySet` structures to save multiple key-value pairs.

A `KeySet` allows us to structure data in a fashion similar to a map (aka hash, map, hashmap, dictionary). Maps are an important data structure, especially in high level programming languages like Python or Ruby. Let us look at an example. We take the last three `Key` objects in Figure 2.6, remove the namespace and translate the data to a Python dictionary:

```
sound = {'pirate': 'arr', 'ninja': 'woosh'}
```

We see that a value in a `KeySet` also maps to a value in the dictionary, while the last part of the key (name) maps to the key in the dictionary.

Apart from the map, another important data structure is the array. Elektra also supports arrays. For that purpose Elektra adds the character `#` and the index to array elements. For example, the following Python list:

```
characters = ['pirate', 'ninja']
```

would translate to the `KeySet` shown in Figure 2.7.

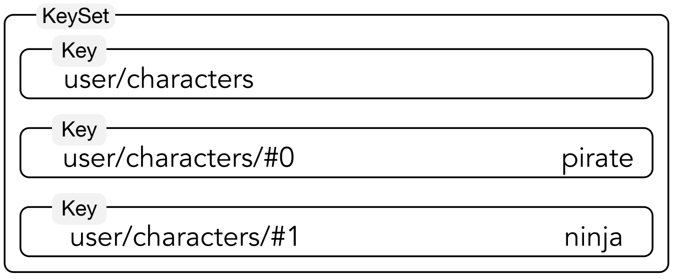


Figure 2.7: Elektra uses the character `#` to mark array elements

Since Elektra orders a `KeySet` alphabetically, a `Key` with name `#10` appears before a `Key` with name `#2`. To fix this problem Elektra adds underscores to keys with larger indices. For example, the 11th element of an array ends with the name `#_10`.

One important difference between Python's list type and Elektra's array type is that arrays do not need to be continuous. For example, it is possible for an Elektra array to only contain elements with the names `#1` and `#3`, while `Key` entries with name `#0` and `#2` are missing.

Elektra also uses the `KeySet` structure to add *metadata* to single keys. For this purpose each `Key` may store a `KeySet` containing simple key-value pairs. Figure 2.8 shows an example `Key` containing two meta keys `comment` and `check/type`.

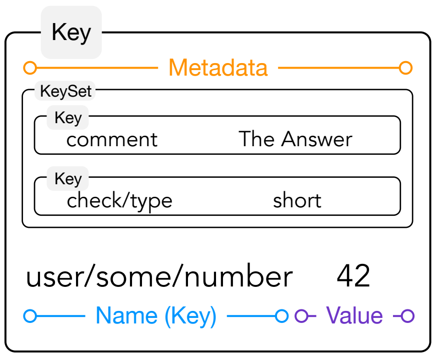


Figure 2.8: Elektra uses a `KeySet` to save metadata for a `Key`.

2.2.3 Plugins

Apart from basic features, most of Elektra's functionality is realized as *plugin*. This has the advantage, that Elektra's core can stay minimal only requiring C99, while plugins are able to implement and use OS specific features.

Many different plugin categories exist. Elektra needs at least one resolver and one storage plugin. The resolver plugin handles filenames and replaces files on disk. Storage plugins, on the other hand, parse configuration files and convert read data to a `KeySet`. They are also responsible for writing a modified `KeySet` back to a given file.

In this thesis we are mostly interested in storage plugins. However, we will also use other plugins to implement common functionality for our YAML storage plugins. For this purpose we use the plugin interface of Elektra to pass key sets between plugins.

The order in which Elektra calls a certain plugin is specified via the *contract* of the plugin. For example, a typical storage plugin will use the positions `getstorage` and `setstorage`. Plugins at the position `getstorage` will be called when Elektra tries to read a configuration file, while Elektra calls `setstorage` plugins when it is time to write a `KeySet` back to a file. A plugin that wants to further process data will usually use the position `postgetstorage` right after `getstorage`, and `presetstorage` the position before `setstorage`.

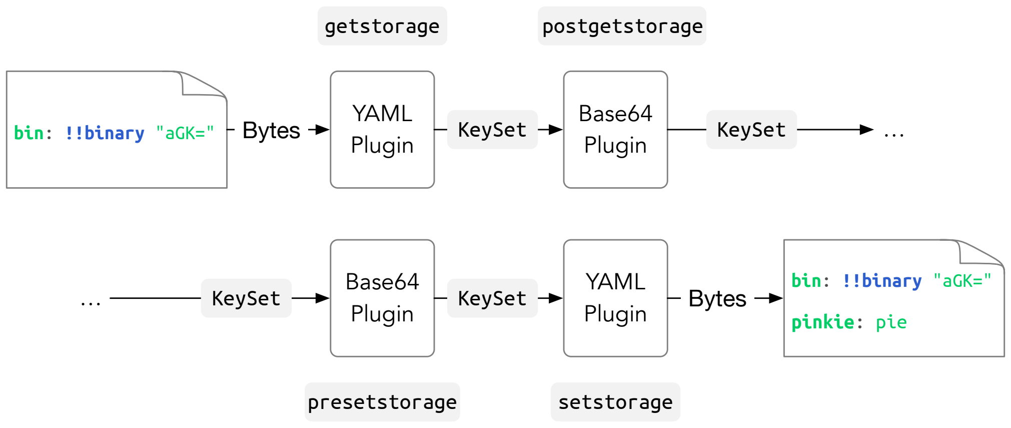


Figure 2.9: Elektra uses multiple plugins to process data.

Figure 2.9 shows a example, where Elektra uses a YAML plugin to read and write data, while a Base64 plugin (see also section "Base64") encodes and decodes binary values. Such a combination of multiple plugins working together is called a *backend*.

In Elektra we *mount* a backend at a certain position of the key-value database. For example, if we mount the backend described above at `user/yaml`, then the YAML plugin is responsible for storing and retrieving values below this mountpoint. If we want to save a new `Key` with the name `user/yaml/pinkie` and the value `pie`, then Elektra

1. uses the YAML plugin to convert the current YAML configuration file to a `KeySet`,
2. decodes every binary `Key` with the Base64 plugin,
3. adds `user/yaml/pinkie` to the `KeySet`,
4. encodes every binary `Key` with the Base64 plugin, and
5. then stores the result in the configuration file using the YAML plugin.

2.3 Related Work

Most of the parser comparison related papers evaluate some form of new parser engine or parser generator and compare it with existing parsers or libraries. A recent example of this type of paper is "Parsing Gigabytes of JSON per Second" [LL19]. In this paper Langdale and Lemire use Single Instruction/Multiple Data (SIMD) instructions to accelerate the parsing of JSON data. In the "Experiments" section of this paper the authors compare their implementation with other JSON parsers, first only using the pure parsing speed, meaning that they ignore the different output of the tested parsers. They also provide a comparison where they show how fast the tested parser are able to find the same data in one of the converted documents from their data set.

While parsing one file format using a specialized parser provides information about how fast we are able to convert certain file formats, this kind of optimization can require substantial manual work. For the plethora of different configuration file formats, it is usually not possible to handcraft parsers by hand. Even if it would be possible, some parsers need to store information that others do not (e.g. comments), making the process of creating a parser that handles all these tasks by hand even harder. To fix this problem we can generate code for the parser.

There are some papers that compare parser generators themselves. For example, in "Full LR(1) Parser Generator Hyacc And Study On The Performance of LR(1) Algorithms" [CP11] Chen and Pager compare the table size and the time to generate parser code for the tools Hyacc, Menhir, MSTA and Bison. This paper does not evaluate the execution speed, error messages or other important criteria of the generated parsers.

In "A Comparison Between Packrat Parsing and Conventional Shift-Reduce Parsing on Real-World Grammars and Inputs" [Flo14] Flodin compares a modified version of the PEG parser Treetop – adapted to produce C++ code – called Hilltop, and Yet Another Compiler Compiler (Yacc). He measures both the execution speed and the heap memory usage of the generated parsers for two different grammars. He does not compare other important criteria, such as the error messages, mainly because the parsers generated by Treetop only print information about the last successful production.

CHAPTER 3

Design Challenges & Decisions

In this chapter we first detail how we determined the subset that our YAML parsers should be able to parse. Afterwards we describe the mapping between YAML and Elektra's `KeySet` structure. In the last part we then explain the challenges we faced implementing the parser plugins and describe the additional plugins we created to improve the YAML support of Elektra.

3.1 YAML Subset

The YAML standard is extensive. The document describing the serialization language includes about 200 parameterized Backus-Naur Form (BNF) grammar rules [BEN09]. To simplify the parser development we decided to first determine a subset of YAML that is useful for storing configuration data. For this purpose we discussed the language with other Elektra developers.

3.1.1 Method

We used a requirement analysis to determine useful YAML features. For that purpose we created a questionnaire that lists YAML features. For each feature we added a checkbox that a participant should check, if they deemed it useful for a YAML subset that stores configuration data. We also added one free form field participants could use to specify additional data types they think should be supported. Since YAML is a complex language we introduced the YAML syntax in a presentation to the Elektra developers first. After we talked about a certain part of YAML, we answered questions the participants had about the information presented so far. Afterwards we asked the participants to fill in the parts of the questionnaire about the newly introduced feature set.

3.1.2 Participants

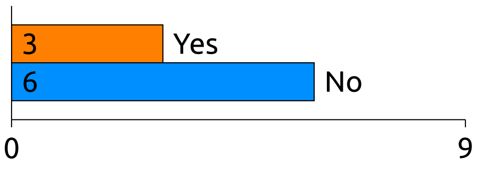
Nine people participated in the requirement analysis. All of the participants were at least partially familiar with Elektra. Some also had previous experience with YAML. Seven of them listened to the presentation, while one participant was late and another one participated via email. The email participant received a copy of the presentation slides and the questionnaire.

3.1.3 Results

In the following bar charts the term "Yes" refers to a checked box for the specific feature. The term "?" means that the participant did not know enough about a part of YAML and therefore marked the checkbox for one feature, or the heading for multiple features, with a question mark. The value before the term "No" specifies the number of unchecked boxes minus the number of boxes marked with "?".

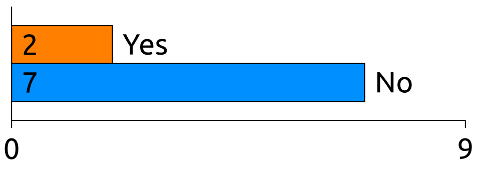
3.1.3.1 Scalars

Flow Scalars



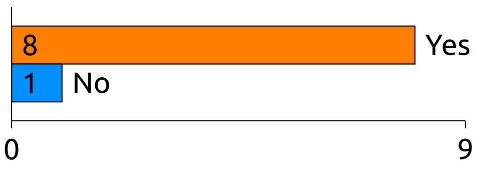
```
Plain String
```

Figure 3.1: Plain Flow Scalar



```
'Single Quoted ''String'''
```

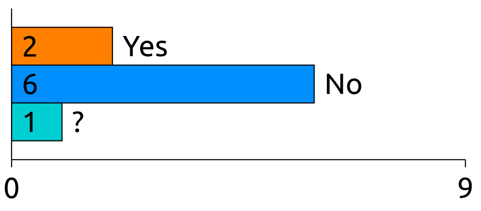
Figure 3.2: Single Quoted Flow Scalar



```
"Double\n Quoted\n \"String\""
```

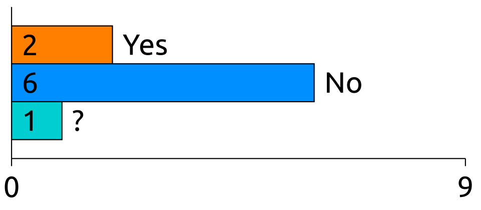
Figure 3.3: Double Quoted Flow Scalar

Block Scalars



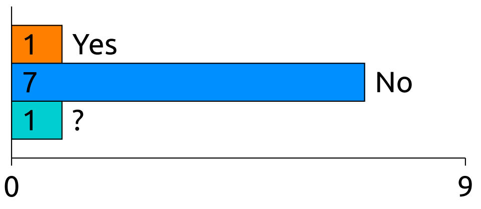
```
> # "Folded Style"
  Folded
  Style
```

Figure 3.4: Folded Block Scalar



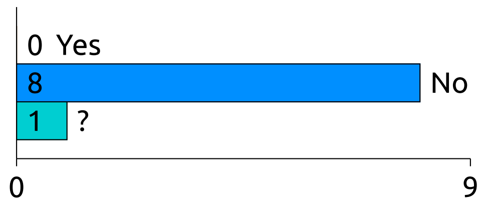
```
| # "Literal\nStyle"
  Literal
  Style
```

Figure 3.5: Literal Block Scalar



```
>1·#·"··1·Space·Indentation"
···1·Space·Indentation
```

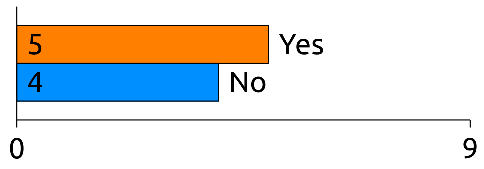
Figure 3.6: Indentation Header



```
>-·# "No·Trailing·Whitespace"
···No·Trailing·Whitespace
···
···
#·⬆️·Newlines·Above·Stripped
```

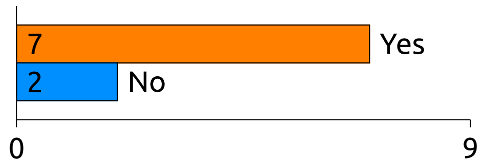
Figure 3.7: Chomping Header

3.1.3.2 Lists



```
[🍎, 🍊,
  [Sugar, Eggs, Chocolate]
]
```

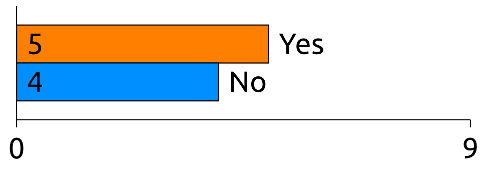
Figure 3.8: Flow Style



```
- 🍎
- 🍊
- - Sugar
  - Eggs
  - Chocolate
```

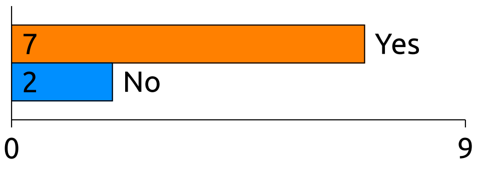
Figure 3.9: Block Style

3.1.3.3 Mappings



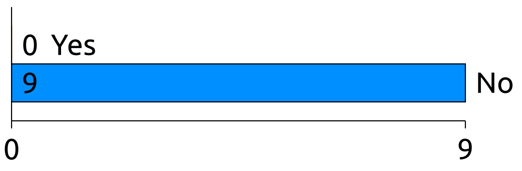
```
{ Austria: Vienna,
  South Africa: {
    Executive: Pretoria,
    Judicial: Bloemfontein,
    Legislative: Cape Town }
}
```

Figure 3.10: Flow Style



```yaml
Austria: Vienna
South Africa:
  Executive:   Pretoria
  Judicial:    Bloemfontein
  Legislative: Cape Town
```

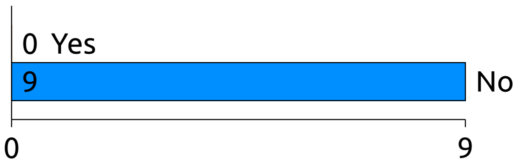
Figure 3.11: Block Style



```yaml
?
- { 'pretty': complex key }
- - 😱
- Still part of the key
: value
```

Figure 3.12: Support for Complex Keys

3.1.3.4 Multiple Documents

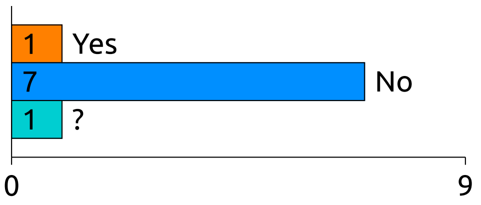


```yaml
"Hello First Document"
...
'Second Document'
...
Third Document
```

Figure 3.13: Support Streams

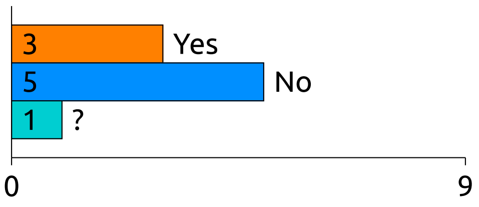
3.1.3.5 Types

Directives



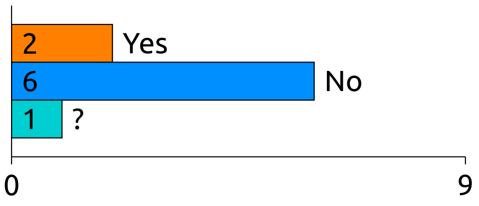
```yaml
%YAML 1.2
```

Figure 3.14: YAML Version



```
%TAG !      tag:yaml.org,2002:
%TAG !!     tag:yaml.org,2002:
%TAG !name! tag:yaml.org,2002:
---
```

Figure 3.15: Tag Handle Definition

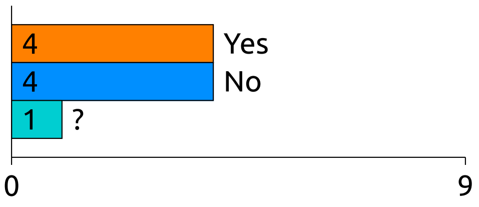


```
%TAG !name! tag:yaml.org,2002:
---
!name!str 6 # "6"
```

Figure 3.16: Named Tag Handle

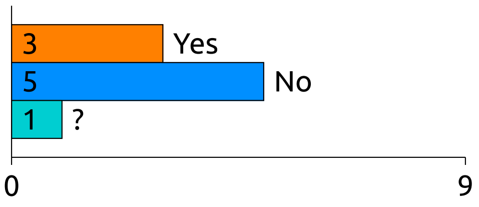
Tags

Tag Shorthands



```
!suffix value
```

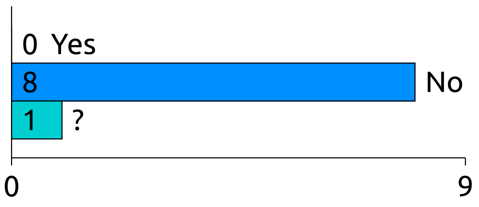
Figure 3.17: Primary Tag Handle



```
!!suffix value
```

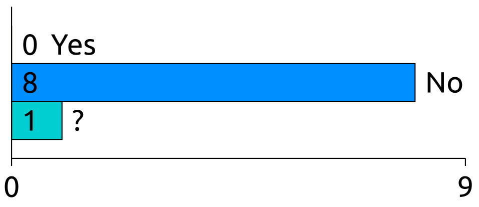
Figure 3.18: Secondary Tag Handle

Verbatim Tags



```
!<!ruby/object:Set> value
```

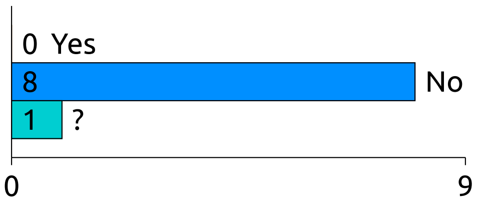
Figure 3.19: Local Verbatim Tags



```
!<tag:yaml.org,2002:str> value
```

Figure 3.20: Global Verbatim Tags

Other Tags

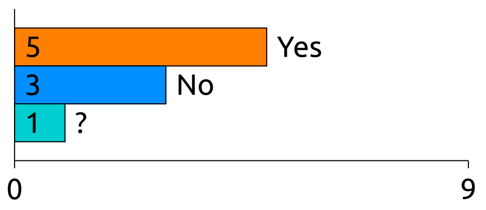


```
! value
```

Figure 3.21: Non-Specific Tag

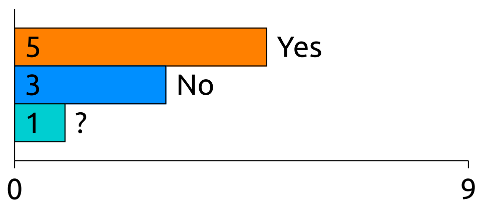
Schemas

Remark: One participant checked the box for the core schema without ticking the boxes for the failsafe and JSON schema. Since the core schema is an extended superset of the other two schemas, we counted the participants answers as a "Yes" vote for the failsafe and JSON schema.



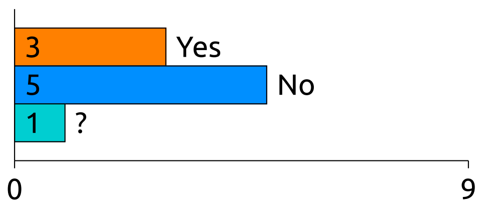
- String
- Sequence
- Map

Figure 3.22: Failsafe Schema



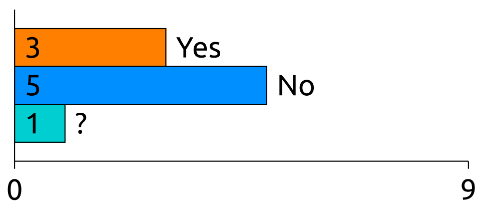
Failsafe Schema + JSON Types:
- Null
- Boolean
- Integer
- Float

Figure 3.23: JSON Schema



JSON Schema and
- Octal/Hex: `0o123`, `0xfefe`
- Multiple Notations for same value: `null`, `Null`, `~`

Figure 3.24: Core Schema



- Ordered Map
- Set
- Binary
- Time
- ...

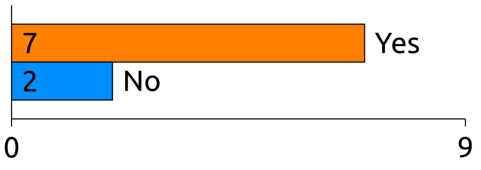
Figure 3.25: Additional Types

Which Additional Types:

- "" (No answer)

- “binary”
- “date (but implemented in plugins)”

3.1.3.6 References



```yaml
flowers: &flowers
  🌳🌸🌼
garden:
  - *flowers # 🌳🌸🌼
  - *flowers # 🌳🌸🌼
```

Figure 3.26: Support Anchors & Aliases

3.1.4 Interpretation

The results of the survey showed that the participants preferred double quoted flow scalars over single quoted and plain scalars. A reason for this could be that those scalars are familiar from other languages such as C, and that they are able to express arbitrary data. Asked about block scalar styles most of the Elektra developers did not think that any of the two styles were necessary.

In contrast to the decision about block scalars, the participants preferred the block styles of sequences and mappings (collections) over the respective flow style. However, they also decided that a useful YAML subset should include flow collections.

The Elektra developers decided against most of the specialized type features of YAML. Only the result count for and against primary tag handles resulted in a draw.

The questions about general type support (schemas) showed that a minimal YAML subset should include all types of the JSON Schema.

One of the few specialized features deemed necessary by the participants were anchors and aliases. These two elements can be used to reference the same data multiple times in the same document.

3.1.4.1 Summary

The list below contains a summary of the YAML features that should be part of a minimal YAML subset according to the results of the discussion:

- double quoted flow scalars,

- block and flow collections,
- JSON schema,
- primary tag handle, and
- references.

3.1.4.2 Problems of the Survey

The survey was done in an early phase of the thesis to gather some insights about YAML features for configuration data. While it showed some interesting results, we noticed problems that made the results unusable for the implementation phase.

One of these problems is the small sample size. With only 9 participants the maximum margin of error assuming a 95% confidence interval is approximately 32%:

$$SE = 1.96 \cdot \sqrt{\frac{\frac{4}{9} \cdot \left(1 - \frac{4}{9}\right)}{9}} \cong 0.32(32\%)$$

Another problem applies even when the margin of error is smaller, which is the case for results with a high percentage for one of the options. The participants were not experts in the area of parsing. At the time of the survey this was also true for the author of the thesis. As a consequence the results included features, such as references, that are not that interesting from a parsing standpoint, but would require significant work in the core of Elektra, something that we consider out of the scope of the thesis.

3.1.4.3 Decision

In the end the decision about the implemented YAML features was largely a results of the implementation phase (see Section Parsers). We built the parser starting with basic features such as scalar support and then added additional features. We decided to implement the following list of items for the YAML subset:

- double quoted scalars,
- single quoted scalars,
- plain flow scalars,
- block collections, and
- core schema (no tag support).

3.2 Mapping Between Elektra's Data Types and YAML

There are basically two more or less obvious solutions to map data between Elektra's `KeySet` structure and a YAML file. Since a `KeySet` behaves similar to a map (see also section “`KeySet`”), connecting a certain key to a certain value, we could use YAML's map type directly.

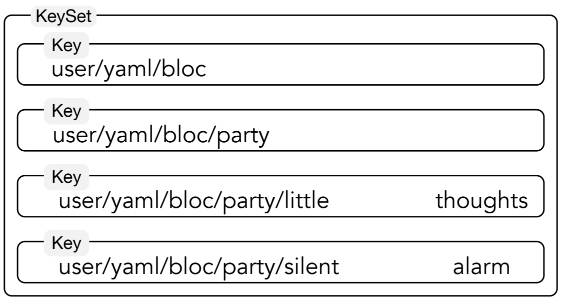


Figure 3.27: An exemplary `KeySet`

For example, the `KeySet` shown in Figure 3.27 would then map to the following YAML data, if we use `user/yaml` as mountpoint:

```yaml
bloc:
bloc/party:
bloc/party/little: "thoughts"
bloc/party/silent: "alarm"
```

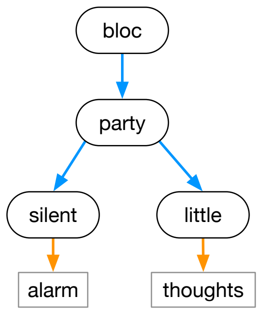
As we can see the resulting YAML file contains quite a lot of redundant data.

In our second solution we take the hierarchical nature of the database into account and split on each part of a key. The result of this approach is the following YAML file:

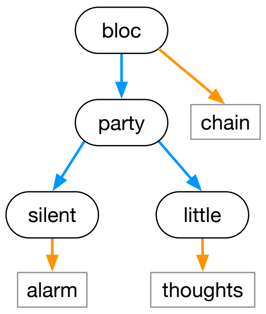
```yaml
bloc:
  party:
    little: "thoughts"
    silent: "alarm"
```

The second solution removes unwanted redundancy and reflects the hierarchy much better. However, the approach also has an obvious downside: What happens if we want to store a value in `user/yaml/bloc` or `user/yaml/bloc/party`? To answer this question, let us look at a tree representing the YAML data from above.

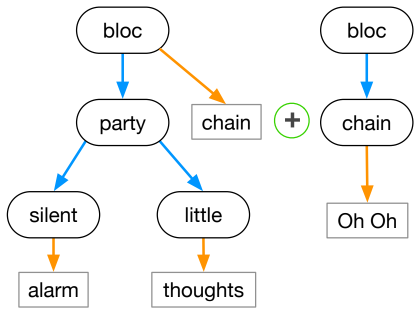
As we can see in Figure 3.28a the only nodes that store values are the leaves of the tree. Let us assume we also want to store the value `chain` in `user/yaml/bloc`. Figure 3.28b shows the resulting tree.



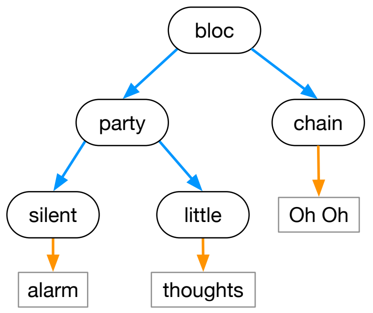
(a) Initial representation



(b) We add an additional value



(c) We add an additional `Key` containing a value



(d) The new `Key` overwrites the value of the node `block`

Figure 3.28: The tree-like representation of YAML data shows the problem of adding non-leaf values

We could now save `chain` as a map key inside `bloc`:

```yaml
bloc:
  chain:
  party:
    little: "thoughts"
    silent: "alarm"
```

However using this approach we are unable to differentiate between the name and the value of a `Key`. For example, if we add a new `Key` with the name `user/yaml/bloc/chain` it would just overwrite the value of `user/yaml/bloc` (see Figure 3.28d).

Another option to fix our problem would be to use YAML's sequence type, and to store the value of a `Key` and the data below the `Key` as first and second element of the sequence:

```yaml
bloc:
  - chain                     # First element stores value
  - party:                    # Second element stores data below
    -                         # `user/yaml/bloc/party` contains no value
    - little: "thoughts"
      silent: "alarm"
```

However, this format is quite complicated. If we add support for Elektra's array type – mapping arrays to YAML sequences – the situation is even worse.

To solve the problem we use another approach. We reserve the name `___dirdata` to save values in non-leaf nodes. The code below shows the mapping of our example data:

```yaml
bloc:
  ___dirdata: "chain"
  party:
    little: "thoughts"
    silent: "alarm"
```

Since we reserved the name `___dirdata` the value below this key will always be a leaf of the tree.

3.2.1 Mapping Arrays

Since Elektra's array type and YAML sequences are similar, we want to map between these data types.

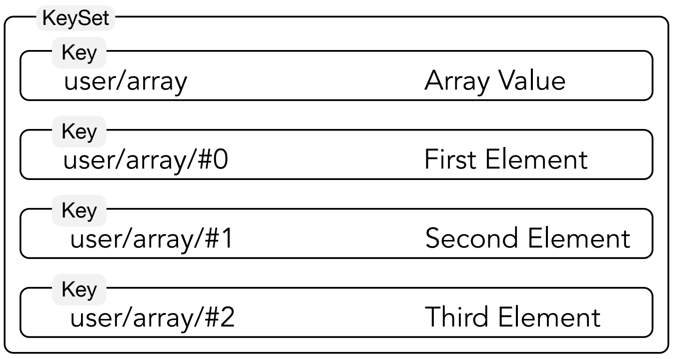


Figure 3.29: The `KeySet` above describes an array containing three elements

If we use this approach, then the `KeySet` shown in Figure 3.29 would result in the YAML data:

```yaml
array:
  - "First Element"
  - "Second Element"
  - "Third Element"
```

We are left with the problem, where to save the data of the *parent* `Key` of the array elements `user/array`. We cannot use the same approach as before:

```yaml
array:
  ___dirdata: "Array Value"
  - "First Element"
  - "Second Element"
  - "Third Element"
```

since the result would be a YAML node that is neither sequence nor map. To fix this problem we decided to convert the `___dirdata` node to a sequence element:

```yaml
array:
  - "___dirdata: Array Value"
  - "First Element"
  - "Second Element"
  - "Third Element"
```

This approach produces valid YAML data and allows us to distinguish between array parents that store values and parents that do not, by checking the first array element for the value prefix `___dirdata:`.

3.3 Parsers

The next section describes some of the implementation challenges we faced when we developed our parsing plugins.

3.3.1 Recursive Descent Parser

The first YAML plugin we developed used a handwritten recursive descent parser. This technique is quite popular, since there exists a natural correspondence between code and grammar rules. Table 3.1 shows the correspondence between Augmented Backus-Naur Form (ABNF) grammar rules and matching C like pseudocode.

Grammar	Example	Code
Terminal	`a = "a"`	`bool a() { bool match = getc(file) == 'a'; if (!match) putc(file); return match; }`
Sequence	`seq = rule1 rule2`	`bool seq() { return rule1() && rule2(); }`
Alternative	`seq = rule1 / rule2`	`bool alt() { return rule1() || rule2(); }`

Table 3.1: Correspondence between grammar rules and code in a recursive descent parser

While Table 3.1 suggest writing a recursive descent parser is trivial, there are many problems that the code above does not take into account.

Recognizer Only: The pseudocode only implements a recognizer for the language. At the end of the parsing process we only know whether the input is part of the language produced by the given grammar or not. Usually we want to *build a data structure*, in our case a `KeySet`, from the given input.

Error Handling: The code does not contain any error handling. If a given input contains errors, then the author of the data wants to know *where these errors occurred*. Otherwise she or he has to check the whole input.

Left Recursion: If we translate a left recursive rules such as `rule1 = rule1 / rule2` using the correspondences given in Table 3.1, then the resulting code would never terminate. This is the case, since `rule1` calls `rule1`, which then calls `rule1`, and so on and so forth (infinite recursion).

All of the problems above apply regardless of the programming language of the parsing code. Since we implemented the recursive descent parser in C, another issue is the error handling in C itself. The language does not provide a native exception handling mechanism. We therefore used the return value to also transfer the error information between functions. This approach is quite cumbersome, since it basically means that we have to check for an error after each function call.

We used C macros to minimize the code overhead and complexity caused by the error handling. Still, for a very small part of the YAML syntax described in the ABNF grammar of

Figure 3.1 the parser contained about 374 lines of code (counted with `cloc` version 1.72). While smaller feature additions, such as supporting multiple key-value pairs instead of only one key-value pair were quite straightforward (14 additions, 1 deletion), other modifications, such as supporting block styles would take considerably more effort.

Since the first steps with a handwritten recursive descent parser showed that this approach takes considerable effort we decided against extending the first prototype. Instead we chose to use an already existing handwritten YAML parser.

```
NEL = %x85
WSLF = WSP / LF

; Printable characters from C0 set
printable = HTAB / LF / CR
; Printable ASCII
printable =/ %x20-7e
; Next Line from C1 set
printable =/ NEL
; Characters after C1 set – Surrogate pairs
printable =/ %xa0-d7ff
; Private use characters – Replacement character
printable =/ %xe00-fffd
; All Unicode Character Starting from the Supplementary Multilingual Plain
printable =/ %x10000-10ffff

pairs = *WSLF "{" pair optionalAdditionalPairs "}" *WSLF
optionalAdditionalPairs = *("," pair)
pair = key ":" value
key = doubleQuotedSpace
value = doubleQuotedSpace
doubleQuotedSpace = *WSLF doubleQuoted *WSLF
doubleQuoted = DQUOTE content DQUOTE
content = *printable
```

Listing 3.1: ABNF grammar for a very small regular subset of YAML

The official YAML website prominently lists known YAML parsers. Since we decided to use C or C++ as programming language – to improve the comparability of the parsers – we are left with three basic options:

- Syck (YAML 1.0)
- LibYAML (YAML 1.1)

- yaml-cpp (YAML 1.2).

Out of these options Syck is not actively maintained any more. This leaves only LibYAML and yaml-cpp. We decided to use yaml-cpp, since it supports the latest version of YAML (YAML 1.2). With the help of the library we added the first plugin with full YAML support called YAML CPP to Elektra.

3.3.2 ALL(*) Parser

The first parser generator we used as part of the thesis was ANTLR 4. As we already described in the section "Parsing", ANTLR 4 generates parsing code that uses an adaptive LL algorithm called ALL(*). For the generated parser we used the C++ target for ANTLR, which was added to the official repository of ANTLR 4 in 2016.

3.3.2.1 Initial Attempt

One of the first problems we encountered using ANTLR was the significant leading white space used to describe the structure of YAML block collection. In YAML increased leading white space starts a new child element, while decreasing amount of leading white space ends an element. Users of programing languages such as Python or Haskell should be familiar with this style. Unlike Python's grammar, YAML's reference grammar does not use `INDENT` and `DETEND` tokens, but uses parameterized BNF productions instead. According to the authors of the YAML specification this is necessary to describe the indentation rules of YAML [BEN09]:

> Many productions use an explicit indentation level parameter. This is less elegant than Python's "indent" and "undent" conceptual tokens. However it is required to formally express YAML's indentation rules.

We first tried to parse different levels of white space using *semantic predicates* and *rule arguments* [PQ95]. Semantic predicates allow us to disable certain parts of a grammar dynamically, while we can use rule arguments to specify different amount of white space. This approach worked, but only for a constant amount of white space. If we tried to specify a different amount of leading spaces in a grammar rule, dependent on the amount of leading white space matched in the current rule, then the generated parser would be unable to parse simple example input.

3.3.2.2 Indent & Detend Tokens

Since the initial attempt for our ANTLR grammar did not show promising results, we looked at how other ANTLR parsers handle significant white space.

Bart Kiers created an ANTLR grammar for Python 3, which captures the start of a document and newline characters inside the lexer. The grammar then uses application-specific code [Par13b, p. 48] to fill a stack with the current amount of indentation and emits `INDENT` and `DETEND` tokens accordingly. Since this method looked promising, we created a basic YAML parser that used the same algorithm. While this approach worked for some YAML documents, we found simple input where the algorithm did not show the expected result. Listing 3.2 shows an example input the parser was not able to handle.

```
1 primes:
2   -
3     one
4   -
5     three
```

Listing 3.2: Our indent/detend YAML parser is not able to handle the simple input above correctly. It expects an indent or detend token in line 4, since the lexer adds an indent token before the token `one` in line 3 and a detend token afterwards. Ideally the lexer would not add these tokens, since `one` is just a simple scalar that – unlike a sequence or map – does not start or end a new level.

Another problem of this approach are complex lexer rules for YAML scalars. The cause of this problem are the parameterized BNF rules of the YAML specification, which do not translate well to the basic lexer syntax used to specify characters and character ranges provided by ANTLR.

3.3.2.3 Custom Lexer

For the final version of our ANTLR parser we looked at the source code of various YAML parsing libraries. One of the most widely used parsing libraries is LibYAML. LibYAML is especially interesting since it was implemented by Kirill Simonov under the guidance of one of the authors of the YAML specification: Clark Evans [Sim18].

Unfortunately LibYAML's code uses C macros quite heavily and is therefore not very readable. A quick look at the code and comments showed that LibYAML's handwritten lexer already takes care of issues such as proper detection of plain scalars and simple keys. Since the lexer uses a turing-complete language to scan the input, it is also able to handle nested block collections.

The source code of LibYAML showed that we can take care of most of YAML's complexity in the lexer. For further information we looked at LLVM's YAML parser (written in C++) and SnakeYAML Engine (written in Java). The lexer of both of these tools use the same basic ideas as LibYAML. However, the code of these libraries is easier to read than LibYAML's, since LLVM's YAML parser and SnakeYAML Engine are written in higher level programming languages.

The most interesting part in the lexer of the libraries described above, is probably how they are able to add tokens for simple keys. The text below describes how the algorithm works (in SnakeYAML Engine).

- The lexer keeps a list of tokens and stores how many of those token it has emitted. The lexer can not simply emit a token unconditionally, since it might have to scan additional tokens to check if the current token starts a simple key.
- When the lexer scans a YAML token that can possibly start a simple key at the current position it adds this token as simple key candidate. For each simple key candidate the lexer also stores the current position in the token list. This way it can insert a key token at this position later, if a key candidate does indeed start a mapping key.
- The lexer only emits the current token, if there are currently no candidates for simple keys. If there are simple key candidates it scans further ahead. Later it
    1. removes simple key candidates if the current lexer position is more than 1024 characters[1] ahead of the start position of the key candidate, or if the key candidate does not start in the current line.
    2. adds a key token for a candidate, if it locates a key value symbol (`:`) for the key candidate.

We used the algorithm described above in the final version of our basic ANTLR storage plugin Yan LR. For that purpose we wrote a custom YAML lexer. Since the lexer takes care of most of the work, the YAML subset grammar itself is quite simple, spanning about 30 lines.

3.3.3 LALR(1) Parser

After we finished the initial version of the ANTLR plugin Yan LR we developed a C++ plugin, that uses a Bison parser, called YAMBi.

For the lexer of the plugin we used a slightly modified version of Yan LR's lexer. This was necessary since Bison, unlike ANTLR, does not provide helper methods to operate on a character stream. For that purpose we wrote a simple class that mimics the behavior of ANTLR's `ANTLRInputStream`.

Apart from the parsing algorithm, another difference between Yan LR and YAMBi is the approach on how the plugin translates YAML data to a `KeySet`. For YAN LR we use a listener interface that operates on the parse tree created by ANTLR. Bison only offers parser actions. These actions contain code that will be called after the parser matches certain parts of the grammar.

[1]The YAML specification specifies this value to limit the lookahead needed to parse a simple key.

YAMBi uses parser actions to call methods of a class that stores temporary data to create the `KeySet`. The code for this approach is actually not that different from the one of the listener we use for Yan LR. The only problematic part was translating code that added a `Key` for a YAML mapping that contains no value to a `KeySet`. For Yan LR we just check, if the mapping parser rule matched a value. This data is already available when we enter the rule, since the listener operates on a completed parse tree. In the corresponding grammar rule of the Bison parser this information is not available yet. To solve this problem we added an additional action to the Bison grammar and moved the code that adds an empty key into a later phase of the conversion process.

3.3.4 Earley Parser

For the Earley parser plugin we used a parsing library called Yet Another Earley Parser (YAEP). We used this library, since it uses C and C++ as programming language, which improves the comparability with the other parsing tools we used. We also found another C Earley Parser implementation by Amirouche Boubekki. However, his parser looked incomplete and unmaintained.

Unlike ANTLR and Bison, YAEP does not generate parsing code, but uses a function to parse input according to a user specified grammar. The output of YAEP is a heterogenous Abstract Syntax Tree (AST). To construct the tree the user annotates the grammar with tree construction rules (marked with `#`). For example, the first line in the code:

```
elements : element          # 0
         | elements element # elements(0 1)
         ;
```

states that the rule `elements` should return the same node `element` produced (`0`), if it matches a single `element`. If the rule `elements` matched a rule `elements` followed by the rule `element` (second line), then YAEP creates a new node with the name `elements` that contains the nodes produced by the rule `elements` as first child (`0`) and the node produced by `element` as second child (`1`).

Since the output of our storage plugin YAwn should be a `KeySet` and not an AST, we created a function that traverses (“walks”) the AST. This function calls certain methods of an auxiliary class `Listener` that creates the `KeySet`. We already used this pattern for the ANTLR plugin Yan LR. The most significant difference to the approach used in Yan LR is that we had to write the tree walking code ourselves.

3.3.5 PEG Parser

All of the previously described parser generators and libraries divide the parsing process into two distinct phases:

1. lexing (generating a stream of tokens from the textual input), and
2. parsing (forming a data structure from the stream of tokens emitted by the lexer).

Figure 3.30 shows a graphical example of this process.

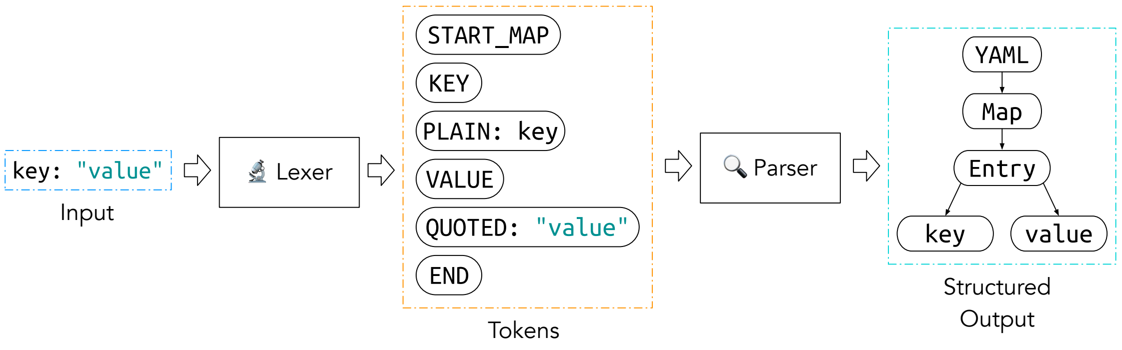


Figure 3.30: A lot of parser engines use two distinct phases (lexing and parsing) to process input.

Unlike ANTLR, Bison and YAEP, the library PEGTL does not use a separate lexing phase. Instead the Parsing Expression Grammar Template Library (PEGTL) parses the input in one sweep using C++ templates to combine simple matching functions into more elaborate parsers. The process of combining parsers this way is also known under the name “parser combinators” (see also Section “Parsing”). Besides the possibility to create *custom matching function*, the library also supports:

- *custom actions* to react to matched input, and
- *custom state* to store contextual data.

Using the three features above we were able to create a PEG parser plugin called YAy PEG that uses grammar rules that are similar to the ones described by the YAML specification.

The translation of some of the rules from the YAML specification to PEGTL rules was trivial. For example, the rule:

```
ns-plain-first(c) ::= ( ns-char - c-indicator )
                    | ( ( "?" | ":" | "-" )
                      /* Followed by an ns-plain-safe(c)) */ )
```

translates to the following code:

```
struct ns_plain_first : sor<seq<not_at<c_indicator>, ns_char>,
                            seq<one<'?', ':', '-'>,
                                at<ns_plain_safe>>> {};
```

The rule `sor` represents ordered choice, meaning that it tries to match it's template arguments:

1. `seq<not_at<c_indicator>, ns_char`, and
2. `seq<one<'?', ':', '-'>, at<ns_plain_safe>>`

in order. It succeeds if one of the argument matches, or fails if all of them fail. The rule `seq` on the other hand tries to match all its template arguments in sequence and either succeeds, if all of them succeed, or fails, if one of them fails. The rules `at` and `not_at` represents the PEG predicates `&` and `!`. The predicate rule `at` (`&`) succeeds if the current input matches the given input and fails if it does not. The predicate rule `not_at` behaves exactly opposite. Neither of these predicates consumes any input. The only remaining rule is `one`, which tries to match any of the given characters, `'?'`, `':'` or `'-'` in our example.

While translating the rule `ns-plain-first(c)` was not that hard, other rules that contained nested contextual data such as:

```
l+block-sequence(n) ::= ( s-indent(n+m) c-l-block-seq-entry(n+m) )+
                        /* For some fixed auto-detected m > 0 */
```

proved more difficult to express in the PEGTL. To translate these kind of rules we added a custom state that contains a stack for the indentation (the values `n` and `n+m` above). In the example above a metarule puts the value `n+m` on the stack before the parser tries to match `s-indent(n+m)` and `c-l-block-seq-entry(n+m)` multiple times. Afterwards the metarule removes the last value from the stack leaving the previous value `n`. We called the general metarule that makes this possible `with_updated_state`:

```
template <typename UpdateStateRule,
          typename RevertStateRule,
          typename... Rules>
struct with_updated_state :
seq<UpdateStateRule,
    sor<seq<Rules...>,
        seq<RevertStateRule, failure>>,
    RevertStateRule> {};
```

As we can see above `with_updated_state` first invokes the rule `UpdateStateRule` to update the state, then it tries to match a sequence of all rules stored in the template parameter pack `Rules`. Depending on the success of `seq<Rules>`,

1. the rule `with_updated_state` either applies `RevertStateRule` (third argument of outermost `seq`) and succeeds, if `seq<Rules...>` succeeds, or

2. it applies `RevertStateRule` and fails (second argument of `sor`) if `seq<Rules...>` failed.

We used `with_updated_state` to create rules that update nested versions of the indentation (`n` and `m` in the YAML spec) and context (`c` in the YAML spec). Using this approach we were able to translate all YAML rules for our subset.

For the conversion of the parsed data to a `KeySet` we used the parse tree facility provided by PEGTL. The whole parse tree contains many unnecessary nodes. Fortunately PEGTL supports parse tree selection and tree rewriting. These features allowed us to keep the parse tree simple. We then use custom tree walking code to walk this tree, invoking a listener at certain nodes, to create a `KeySet`. This approach is very similar to the one we used for the YAwn plugin (see Section "Earley Parser").

3.3.6 Parser Combinator

As described in the previous section "PEG Parser" we already used a parser combinator library to create a parser for our basic YAML subset. Initially we also wanted to use a second parser combinator library called mpc. However we decided against using mpc, since

- the parser engine only supports ASCII encoded data,
- requires manual memory management – mpc is written in C – and
- does not provide built-in support for advanced features such as tree selection.

In the end we did not think the effort to create yet another parser was worth the time, since at least in theory PEGTL seemed to be the better choice.

3.3.7 Augeas Lens

Augeas [Lut08] is a tool that uses so-called lenses to edit configuration data. The main advantage of lenses is that they handle both the parsing and writing process. Since Elektra already includes a plugin for Augeas [Ber16], it sounds like a YAML lens is the ideal tool to convert YAML data to a `KeySet`. In reality there are multiple problems with this approach. Besides the issues mentioned by Berlakovich in his bachelor thesis [Ber16], one of the main problems is that YAML is a context-sensitive language [Lut17], while Augeas offers only full support for regular languages. With this in mind we tested the official YAML lens with Elektra's `kdb` tool. Since even the conversion of a single single key-value pair failed, we used the tool `augcheck` to make sure the YAML lens is able to parse our example data. This tool showed us that the YAML lens currently supports nested mappings, such as

```yaml
root:
  key: value
```

but is unable to handle a non-nested mapping:

```yaml
key: value
```

and other simple data. With the current state of the YAML lens and the problems of the Augeas plugin in mind, we decided to not look any further into developing an Augeas lens for our YAML subset.

3.4 Additional Plugins

While most of the problems of adding a YAML storage plugin deal with the parsing process itself, there are other issues we handled using additional plugins. Elektra's plugin system allows us to use multiple plugins in conjunction as part of a so-called *backend* (see also section "Plugins").

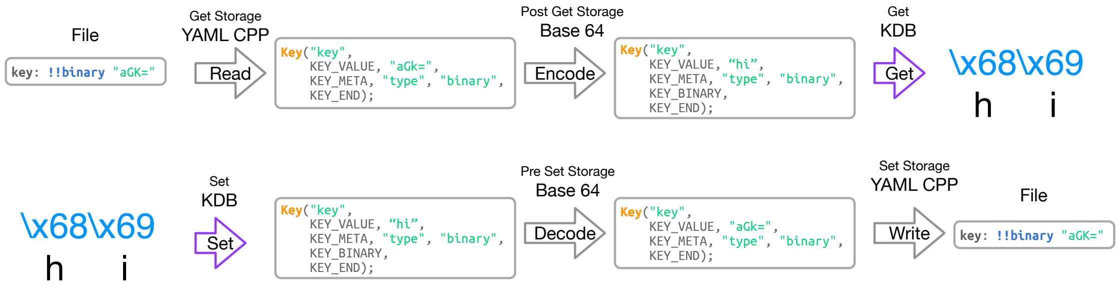
3.4.1 Base64

One of the first plugins we used to improve the YAML support of Elektra was the Base64 plugin of Peter Nirschl [Nir18]. The plugin en- and decodes binary values using the Base64 algorithm [Jos06].

Since Elektra supports values containing binary data, we can use the Base64 plugin to encode this data and store it using ASCII values in a YAML file. However, the plugin used a common prefix to mark base64-encoded data. For example, if we want to store the decimal numbers 104 (0x68) and 105 (0x69), then the plugin would encode these values as `aGk=` and add the prefix `@BASE64`. The resulting value would then be `"@BASE64aGk="`. In YAML a value should not contain a prefix. Instead YAML marks base64 encoded data with the tag (data type) `!!binary`. We therefore need to store the two values above as `!!binary "aGk="` in a YAML file. For this purpose we added a new mode to the Base64 plugin.

The new *meta mode* uses metadata to mark a key-value pair that contains a base64-encoded value. Instead of a prefix Base64 adds a meta-key `type` with the value `binary`. Figure 3.31 shows an example, where Elektra uses the Base64 plugin to encode and decode the bytes 0x68 and 0x69 (code points for the ASCII string `hi`).

We should mention here that we only added support for the Base64 encoded data to the YAML CPP plugin, since we decided to not support tags for our YAML subset (see Section "Decision").



```
key: !!binary "aGK="
```

```
Key("key",
    KEY_VALUE, "aGk=",
    KEY_META, "type", "binary",
    KEY_END);
```

```
Key("key",
    KEY_VALUE, "hi",
    KEY_META, "type", "binary",
    KEY_BINARY,
    KEY_END);
```

```
Key("key",
    KEY_VALUE, "hi",
    KEY_META, "type", "binary",
    KEY_BINARY,
    KEY_END);
```

```
Key("key",
    KEY_VALUE, "aGk=",
    KEY_META, "type", "binary",
    KEY_END);
```

```
key: !!binary "aGK="
```

Figure 3.31: The Base64 plugin decodes and encodes binary data.

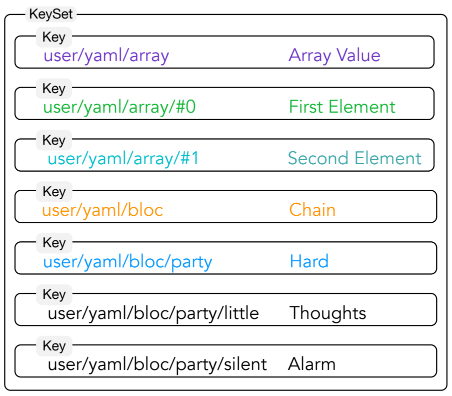
3.4.2 Directory Value

We already described the problem of storing a value in a non-leaf (directory) `Key` in the Section "Mapping Between Elektra's Data Types and YAML". Since the problem is independent of the parser engine and also relevant to other plugins, we implemented the functionality in a plugin named Directory Value.

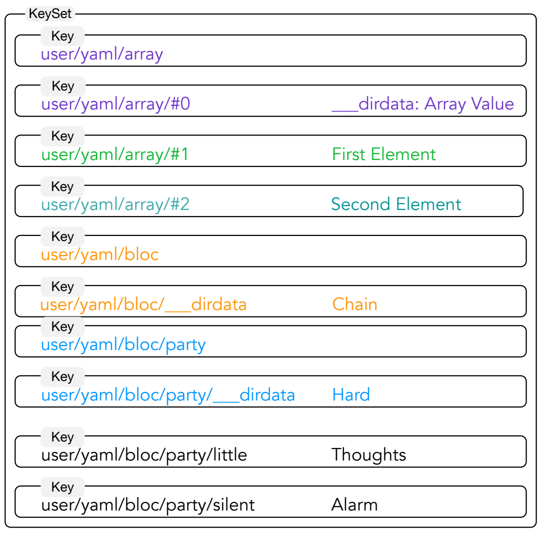
The Directory Value plugin adds an additional `Key` with the prefix `___dirdata` for every non-array `Key` that has children and contains a value in the `set` direction (position `preset`). For example, for the `KeySet` shown in Figure 3.32a, the plugin adds the `Key`

- `user/yaml/bloc/___dirdata` and
- `user/yaml/bloc/party/___dirdata`.

The plugin then moves the data stored in the parent `Key` to the newly created `Key`.


KeySet
Key
user/yaml/array Array Value
Key
user/yaml/array/#0 First Element
Key
user/yaml/array/#1 Second Element
Key
user/yaml/bloc Chain
Key
user/yaml/bloc/party Hard
Key
user/yaml/bloc/party/little Thoughts
Key
user/yaml/bloc/party/silent Alarm

(a) We use the `KeySet` above as input for the Directory Value plugin at the `preset` position.


KeySet
Key
user/yaml/array
Key
user/yaml/array/#0 ___dirdata: Array Value
Key
user/yaml/array/#1 First Element
Key
user/yaml/array/#2 Second Element
Key
user/yaml/bloc
Key
user/yaml/bloc/___dirdata Chain
Key
user/yaml/bloc/party
Key
user/yaml/bloc/party/___dirdata Hard
Key
user/yaml/bloc/party/little Thoughts
Key
user/yaml/bloc/party/silent Alarm

(b) The `KeySet` above shows the result of the conversion at the `preset` position.

Figure 3.32: The Directory Value plugin adds data at the position `preset` (3.32b) and then restores the original data (3.32a) at the position `postget`.

In addition, the plugin inserts a new `Key` for every array parent that stores a (non-binary) value at the first position of the array. In our example, the plugin adds a new `Key` with the value `___dirdata: Array Value` at the first position of `user/yaml/array` and increases the index of all other array elements by one.

Figure 3.32b shows the `KeySet` after the whole conversion at the position `preset`. This `KeySet` is also the input for the Directory Value plugin at the position `postget`.

3.4.3 YAML Smith

Elektra's storage plugins need to both:

1. convert a configuration file format to a `KeySet`, and
2. convert a `KeySet` to a configuration file format.

Since the second task is always the same, regardless of the parser library we use, we created a plugin called YAML Smith that takes care of this task.

This plugin first determines all leaf keys of a `KeySet`. After that it counts the levels of the parent key so it knows how many levels it has to skip for each leaf. The plugin then iterates over each leaf key.

For every leaf key the plugin iterates over each level of the name. First it skips all levels of the parent key. After that it skips all levels of the prefix it shares with the key that came before. In this process it adds a constant amount of spaces for each of the levels to an initially empty string and stores the result in a variable called `indent`. Now the plugin adds each remaining level of the key in its own line. To do that it first writes the content of `indent`, then adds the current part of the key, and after that the appropriate marker, either `-` for an sequence, or `:` for a map. For each written level of the key the plugin increases `indent` for the next key part. In the last step for a specific leaf key, the plugin adds another newline, the current indentation and finally writes the value of the key in double quotes.

CHAPTER 4

Evaluation

In the evaluation phase of the work we compare our parser plugins. For that purpose we first describe our comparison criteria in the first section of this chapter. After that we measure and analyze the plugins according to each criteria in the following sections. At the end we determine the plugins that best fit our criteria.

4.1 Goals

The goal of this evaluation is to find one or more parser plugins that

- are fast,
- have low resource usage,
- use code that is
  - both maintainable and
  - easily extendible, and
- has good error reporting capabilities.

To make sure that the plugins are reasonably fast we compare their execution time using runtime benchmarks and answer RQ 1.

? **RQ 1.** *How does the theoretic runtime complexity of the parsing methods compare to the actual measured runtime of the parsing code?*

For the resource usage we analyze the heap memory consumption and answer RQ 2.

? RQ 2. *How does the peak memory usage of the algorithms compare to each other? Do some of the algorithms show nonlinear memory usage?*

To make sure that the plugins are maintainable we analyze code sizes and take a look at the cyclomatic complexity. Using the information gained in this task we answer RQ 3.

? RQ 3. *How much work does it require to implement the plugins, i.e. how many lines of code do we have to write to support our YAML subset for each parsing engine? How do the amounts of handwritten code for the plugins compare to each other?*

We also examine the extensibility and composability of the plugins looking at code changes for specific bug fixes and feature additions. This step helps us to answer RQ 4.

? RQ 4. *Which parsing technique allows us to stay closest to the definition of the configuration language? Does staying close to the given definition allow us to extend and improve the parser and its support code more easily?*

To make sure the plugins produce good error messages we also compare these messages for specific input files in a detailed analysis and answer RQ 5.

? RQ 5. *What are the error handling capabilities of the parsing engines? How well can they handle multiple syntax errors? How do the generated error messages compare to each other?*

4.2 Evaluated Plugins

The mapping between the parsing plugin names and the parser libraries/generators might be not apparent to everybody. Since we mainly use the plugin name in the evaluation we provide an overview of the mapping in Table 4.1. The colored text shows how you can memorize the mapping more easily. Apart from YAwn all plugins share at least two consecutive letters with the used parsing library. The name of YAwn is based on the very bad that "pun" that you might have to "yawn" if you get up earl(e)y.

Table 4.1: Plugin Overview

Plugin	Library/Generator	Parsing Technique
● YAML CPP	yaml-cpp	Handwritten Recursive Descent Parser
● Yan LR	ANTLR	ALL(*)
● YAMBi	Bison	Look-Ahead LR (LALR)
● YAwn	YAEP	Earley Parser
● YAy PEG	PEGTL	PEG Parser

4.3 Performance Analysis

In the following section we analyze the runtime and memory usage of our plugins for certain input files. We first describe the overall methodical steps we took for both the runtime and memory benchmarks. Then we describe, measure and analyze both of these criteria in their own sections.

4.3.1 Method

To make all benchmarks reproducible we start by detailing the whole setup including used hardware, software, build configuration options and how we generated the input files.

4.3.1.1 Hardware

For all of the tests we used the hardware described in Table 4.2.

Table 4.2: Hardware Setup

MacBook Pro (Retina, 15-inch, Late 2013)	
CPU	i7-4960HQ 2.6 GHz 6 MB L3 Cache 128 MB L4 Cache
RAM	16 GB 1600 MHz DDR3
HD	Apple SSD SM1024F 1 TB

4.3.1.2 Software

Table 4.3 shows the overall software setup for the benchmarks. We tested the performance both on macOS and Linux. For the Linux setup we used the Mac version of Docker. The basis of the runtime benchmark is commit 54a4c019 of Elektra's code base, while we measured the memory usage using commit ea418f17. We used two different commits, because we measured the memory usage some weeks after the run time. No code of any of the tested plugins changed between the two commits.

Table 4.3: Software Setup

(a) Mac Setup

OS	macOS 10.14.5
Compiler	Clang 8.0.0
Generators	ANTLR 4.7.2
	Bison 3.4.1
Libraries	yaml-cpp 0.6.2
	YAEP 550de4cc
	PEGTL 2.8.0
Other Software	CMake 3.14.4
	Ninja 1.9.0
	hyperfine 1.5.0
	cloc 1.82

(b) Linux Setup

Docker	18.09.2, build 6247962
Base Image	Debian sid (sid-20190506)
Compilers	Clang 6.0.1/GCC 8.3.0
Generators	ANTLR 4.7.2
	Bison 3.3.2
Libraries	yaml-cpp 0.6.2
	YAEP 550de4cc
	PEGTL 2.7.1
Other Software	CMake 3.13.4
	Ninja 1.8.2
	hyperfine 1.5.0

4.3.1.3 Build Setup

The following list shows the CMake options that we used for all benchmarks:

- `-GNinja`,
- `-DPLUGINS=ALL`,
- `-DCMAKE_BUILD_TYPE=Release`,
- `-DENABLE_LOGGER=OFF`, and
- `-DENABLE_DEBUG=OFF`.

4.3.1.4 Input

As first input for the benchmarks we used a JSON configuration file of the YAJL plugin that we converted to block syntax using the YAML CPP plugin. We then modified the exported data by removing all `!<!elektra/meta>` tags, which are not supported by the other YAML plugins. We call the resulting file `keyframes.yaml` in the remainder of the thesis. This file and all other data of the benchmarks is available here:

```
http://rawdata.libelektra.org/tree/master/YAML
```

For another input file called `combined.yaml` we copy and pasted parts of test data and various other YAML files in Elektra's repository into a single file. While the file content is nonsensical, it should at least contain a mix of YAML data that covers most of the code paths of the YAML plugins.

Since both of these files are relatively small, `keyframes.yaml` contains 218 lines, while `combined.yaml` contains 152 lines, we also generated data using a Python script that we called `generate-yaml`. This script generates YAML maps using Universally Unique Identifiers (UUIDs) as scalar keys and values. For the YAML scalars the script randomly selects one of the three flow scalar styles:

- single quoted scalar,
- double quoted scalar, or
- plain scalar.

This always works since UUIDs contain no character sequence that has special meaning according to the YAML specification. Using this method we generated two files composed of mappings and scalars.

- The file `generated.yaml` contains 10 000 lines and has a maximum nesting of 26 levels.
- The file `generated_100000.yaml` contains 100 000 lines and has a maximum nesting of 31 levels.

We also created another script called `cut_input` to generate additional smaller input files that contain the first 50000, 10000, 5000, 1000, 500, 100, 50, 10, 5 and 1 lines of `generated_100000.yaml`.

4.3.2 Runtime Performance

4.3.2.1 Method

To compare the runtime performance we used the C application `benchmark_plugingetset` that opens an Elektra plugin using a specific configuration file. For the whole benchmark process we created a Bash script, called `benchmark-yaml`, that uses the benchmarking tool `hyperfine` to call `benchmark_plugingetset` using different YAML plugins. Figure 4.1 shows a diagram of this setup.

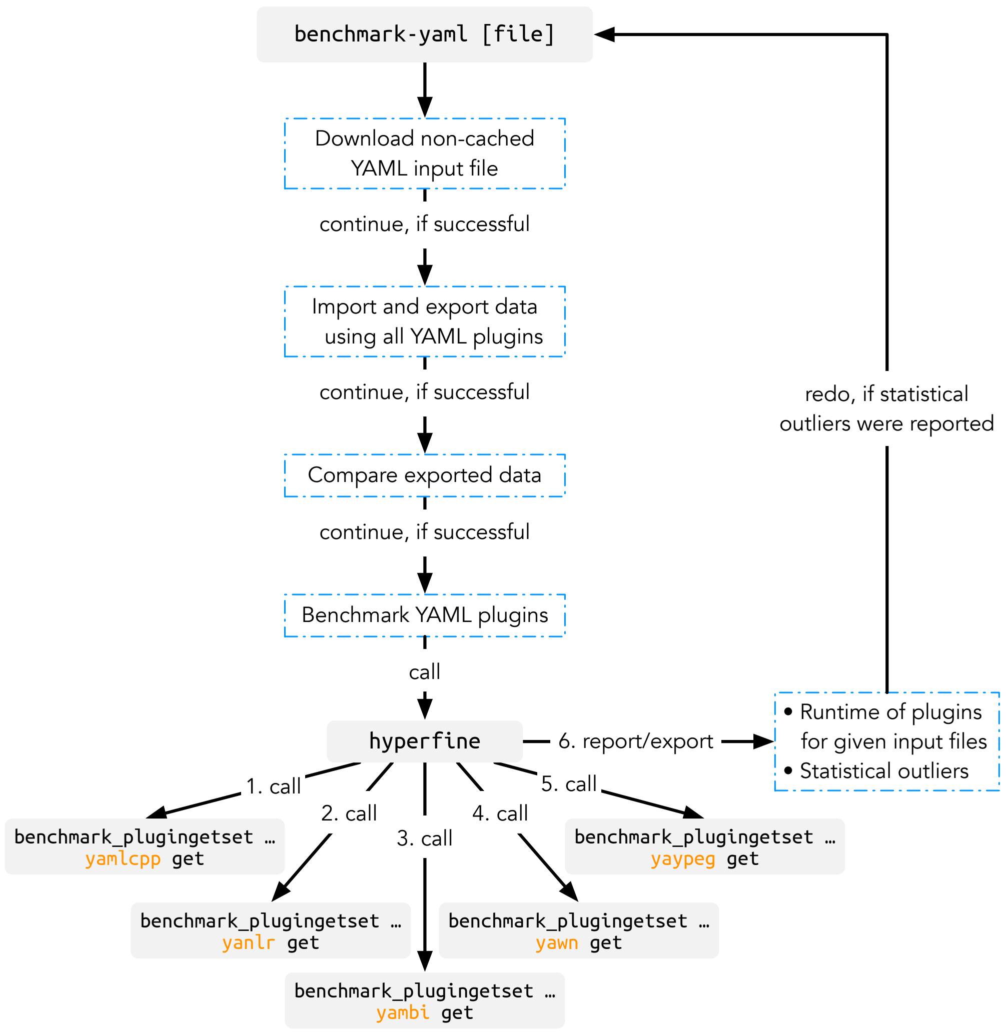

benchmark-yaml [file]
Download non-cached YAML input file
continue, if successful
Import and export data using all YAML plugins
continue, if successful
Compare exported data
continue, if successful
Benchmark YAML plugins
call
hyperfine
6. report/export
• Runtime of plugins for given input files
• Statistical outliers
redo, if statistical outliers were reported
1. call
2. call
3. call
4. call
5. call
benchmark_plugingetset … yamlcpp get
benchmark_plugingetset … yanlr get
benchmark_plugingetset … yambi get
benchmark_plugingetset … yawn get
benchmark_plugingetset … yaypeg get

Figure 4.1: The diagram above shows the basic sequence of steps to measure the runtime performance of the YAML plugins.

For the measurement we use `hyperfine`, since this tool

1. automatically determines how often it should call `benchmark_plugingetset` for meaningful measurement results,
2. shows us when it is time to redo the benchmark, by informing us about statistical outliers, and
3. prints important statistical data such as the mean runtime and the standard derivation.

To make sure we repeat the benchmark every time `hyperfine` reports statistical outliers we created the Shell script `benchmark-runtime`. This script calls `benchmark-yaml` for every input file and repeats a benchmark, until `hyperfine` does not report any warnings. The script `benchmark-runtime` *does* ignore warnings about runtimes under 5 milliseconds though, since `benchmark_plugingetset` might take less execution time for small YAML files.

4.3.2.2 Results

The graphs in this section show the results of the benchmark for different input files.

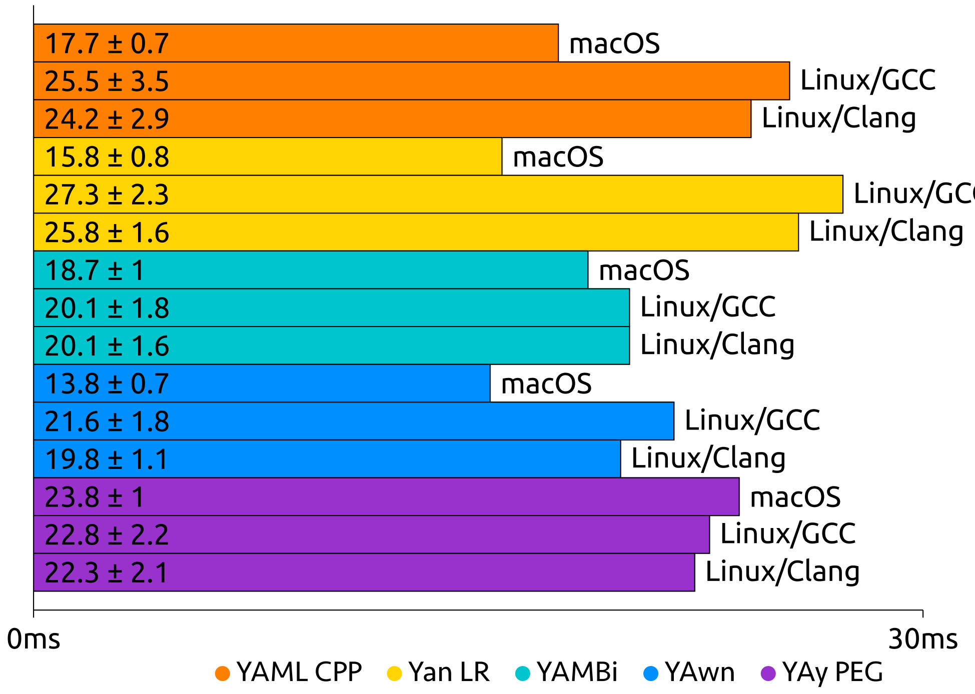


Figure 4.2: This bar chart shows the run time of the plugins for the input `keyframes.yaml`.

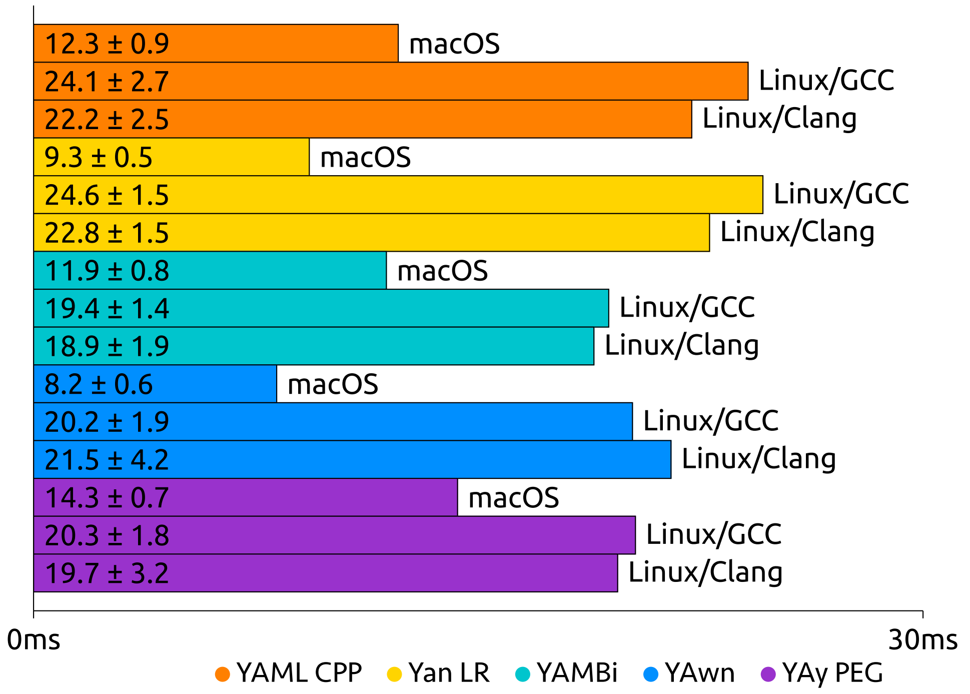


Figure 4.3: This bar chart shows the run time of the plugins for the input `combined.yaml`.

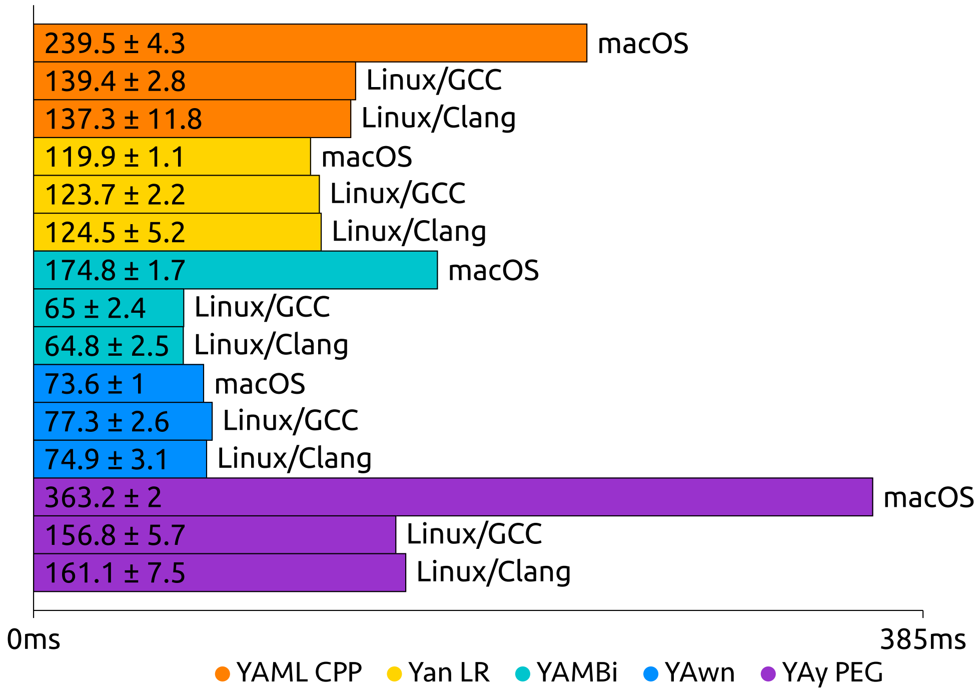


Figure 4.4: This bar chart shows the run time of the plugins for the input `generated.yaml`.

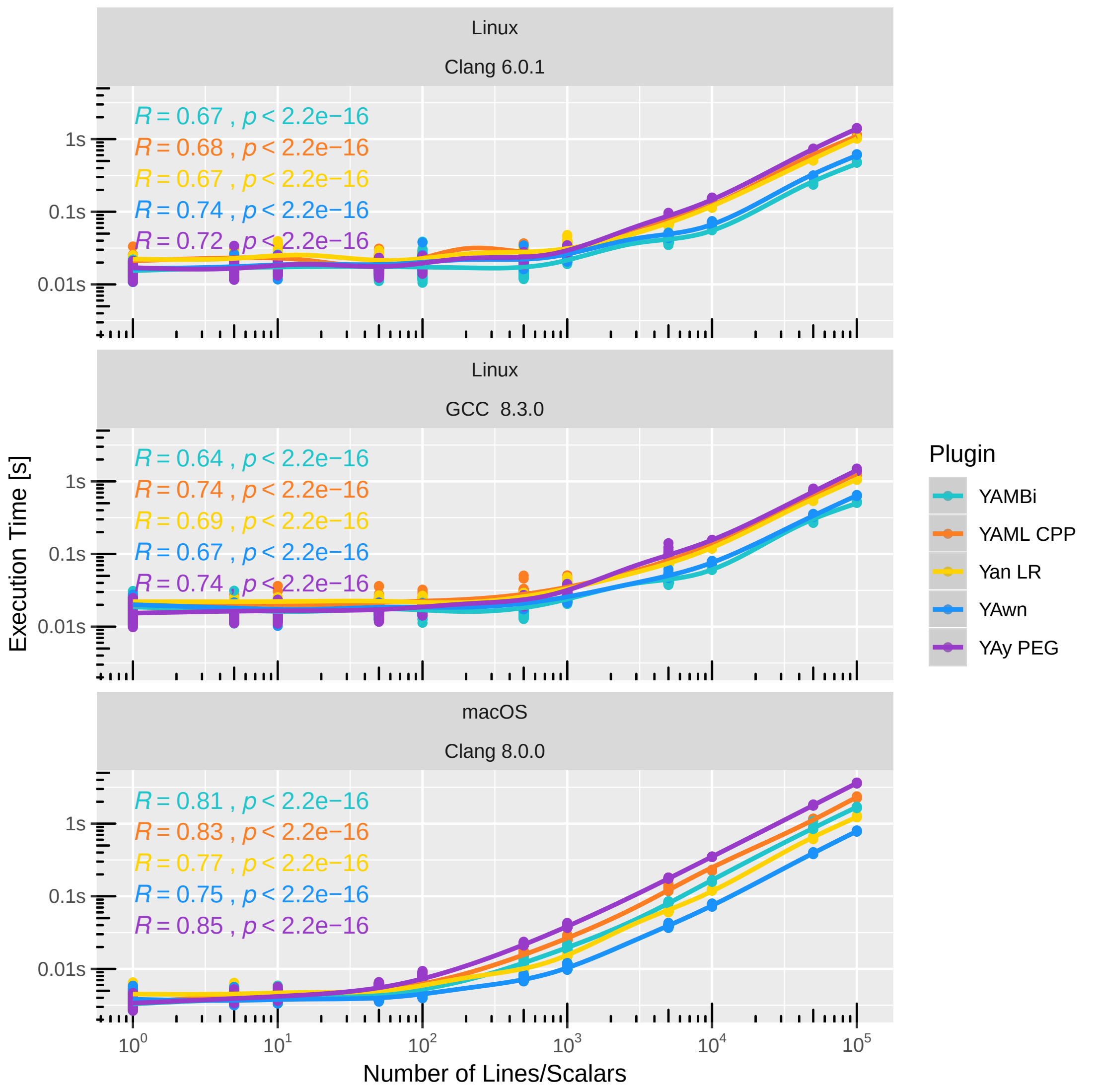


Figure 4.5: The diagrams above show the runtime of the plugins for the input file `generated_100000.yaml` and other files that contain only the first n number of lines of this file.

4.3.2.3 Analysis

If we look at Figure 4.5 we see that the runtime seems to grow linearly after a certain number of input lines. To verify this hypothesis we removed all samples with a line length smaller than 1000. Figure 4.6 shows the result.

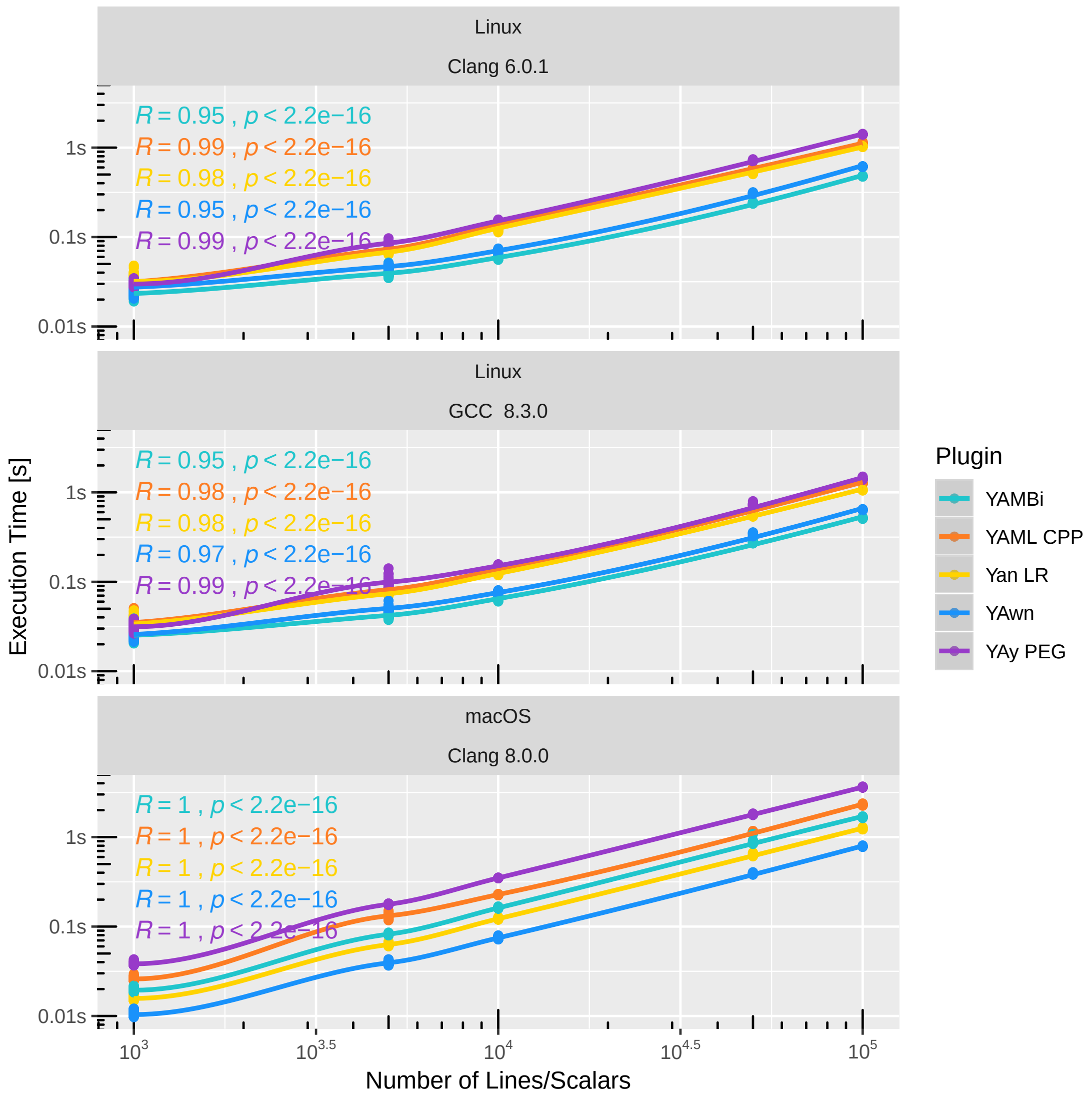


Figure 4.6: The diagrams above shows that the runtime for the first lines of the file `generated_100000.yaml` almost certainly grows linearly.

From the correlation coefficients (R) of 0.95 and higher we deduce that the runtime for all plugins almost certainly grows linearly. This means that the approximate runtime of the plugins should be the same. Now it is time to answer RQ 1.

? RQ 1. *How does the theoretic runtime complexity of the parsing methods compare to the actual measured runtime of the parsing code?*

The runtime of a non-backtracking recursive descent parser for an LL(1) grammar, as used by

YAML CPP, should be $O(n)$. According to the literature the upper boundary for the runtime of

- ALL(*), used by Yan LR, is $O(n^4)$, but the algorithm often performs linearly [PHF14, p. 1],
- LALR, used by YAMBi, is $O(n)$ [Bax17],
- an Earley parser, used by YAwn, is $O(n^2)$ for unambiguous grammars [HU69, p. 145],
- a general PEG parsers, as used by YAy PEG, is exponential [Mos14, p. 1] ($O(c^n)$) and for PEG parsers that use memoization is $O(n)$ [For02].

We now compare the theoretic runtimes with the measured runtimes shown in Figure 4.6. The text below lists some of our observations.

- The deterministic (aka non-backtracking) parsers (YAML CPP, YAMBi) show the expected linear behavior.
- Yan LR also executes in linear time for the input. This is probably the result of the relatively simple grammar used by the parser. At least for all our input and test files we also checked that the grammar works with the simpler, but faster Strong LL(*) (SLL(*)) strategy. This was indeed the case.
- YAwn also shows the linear behavior, even though YAEP does not implement Leo's optimization [Leo91] that makes sure that the algorithm runs in linear time for every LR(k) grammar.
- Even YAy PEG's backtracking parser without memoization shows linear behavior. As we already mentioned before in "State of the Art" it is generally not clear, if memoization provides runtime improvements for a certain grammar. This seems also one of the reasons, why PEGTL does not use memoization, as can be seen in a quote of Colin Hirsch [Hir16], one of the authors of PEGTL, below.

  > ...it would increase the complexity of the library beyond our design goals and only be useful in a very limited number of cases - in practice packrat parsers often perform worse than simple recursive descent parsers despite being in a better time complexity class.

Figure 4.4 shows that the constant factor between the linear runtimes can be relatively high. For example on macOS, the fastest plugin YAwn is nearly five times faster than the slowest plugin YAy PEG for the file `generated.yaml`:

$$\frac{363.2\text{ms}}{73.6\text{ms}} \cong 4.93 \tag{4.1}$$

Other interesting observations concerning the runtime are listed below.

1. The difference between the runtime of the fastest and slowest plugin for large files is nearly twice as large on macOS.
2. The OS seems to play a much more important role than the compiler for all of the used YAML libraries.
3. While Yan LR and YAwn perform similarly on macOS and Linux, the difference between YAML CPP, YAMBi and YAy PEG on both operating systems can be quite significant.

Since the runtime of the plugins is quite different on the two benchmarked operating systems we also determined mean of the mean values for the file `generated.yaml`. We use a weight of

- 0.5 for the combination macOS/Clang,
- 0.25 for the combination Linux/Clang, and
- 0.25 for the combination Linux/GCC

and obtain the formula:

$$\bar{t} = 0.5 \cdot \bar{t}_{\text{macOS/Clang}} + 0.25 \cdot \bar{t}_{\text{Linux/GCC}} + 0.25 \cdot \bar{t}_{\text{Linux/Clang}} \quad (4.2)$$

Figure 4.7 shows a bar graph with the result of this calculation.

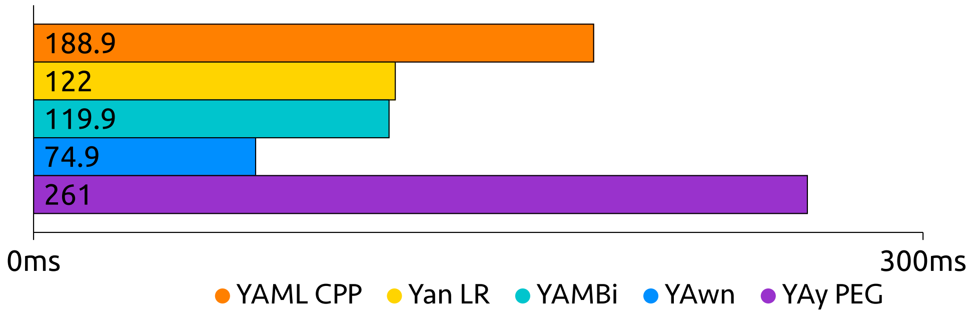


Figure 4.7: This bar chart shows the mean of the mean run times of the plugins according to Equation 4.2 for the input `generated.yaml`.

4.3.2.4 Conclusion

We determined with a high confidence that all of the YAML plugins show a linear runtime behavior (see Figure 4.6), at least for the file `generated_100000.yaml`. This puts all plugins, into the same computational complexity class. The constant factors between the runtimes can still be relatively high as we can see in Equation 4.1.

The fastest plugin according to the mean of the mean runtimes (see Figure 4.7) is YAwn if we weigh the results of the two tested operating systems equally. This is interesting, since Earley himself mentions in his dissertation [Ear70, p. 122] that his parsing technique was too slow for practical use at the time, as you can see in the quote below.

> First we ask, what impact will our algorithm have on the parsing done in production compilers for existing programming languages? The answer is, practically none. Production compilers require guessing time proportional to n with a fairly low coefficient of n.

Yan LR and YAMBi showed the second best runtimes, and were about 1.6 times slower than YAwn. YAML CPP, which was, according to the results of Figure 4.7, about 2.5 times slower than Yawn takes the second to last place. Yay PEG was the slowest plugin on both tested operating systems, and is about 3.5 times slower than the fastest plugin according to Figure 4.7.

4.3.3 Memory Usage

4.3.3.1 Method

We measured the heap memory usage with the heap profiler Massif. Most of the other setup is similar to the one we used for the runtime benchmark, described in the section "Runtime Performance". We still use the C application `benchmark_plugingetset` to execute the plugins. The input files are also the same as before.

This time we do not need to determine the mean value of the results. Massif always produces the same output on the same hardware/software combination, since it runs the instrumented program `benchmark_plugingetset` on a "synthetic CPU" [Val19].

To automate the process of measuring the memory usage for the different input files, we created a Shell script called `benchmark-memory`. We only benchmarked the memory usage on Linux, since Massif did not support macOS 10.14 at the time we executed the benchmark script.

4.3.3.2 Results

The graphs in this section show the results of the memory benchmark for different input files.

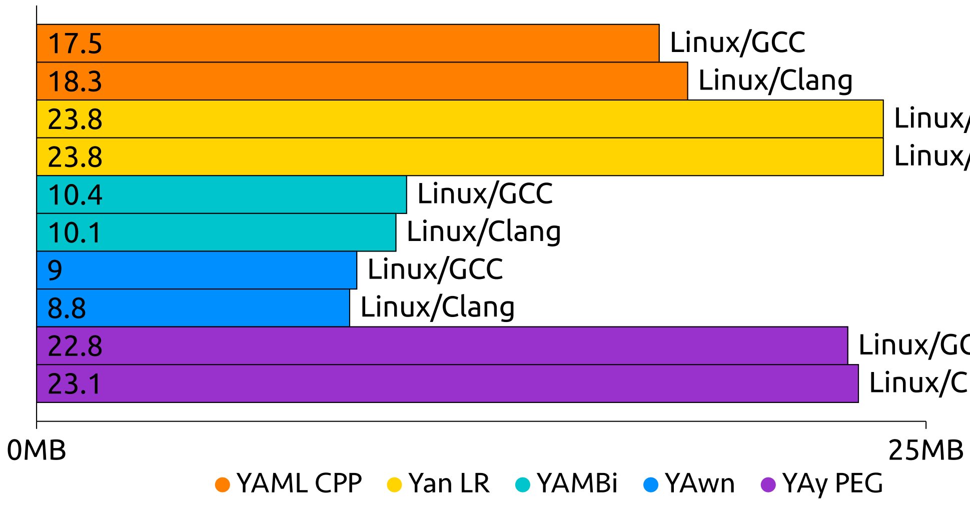


Figure 4.8: This bar chart shows the peak heap memory usage of the plugins for the input `keyframes.yaml`.

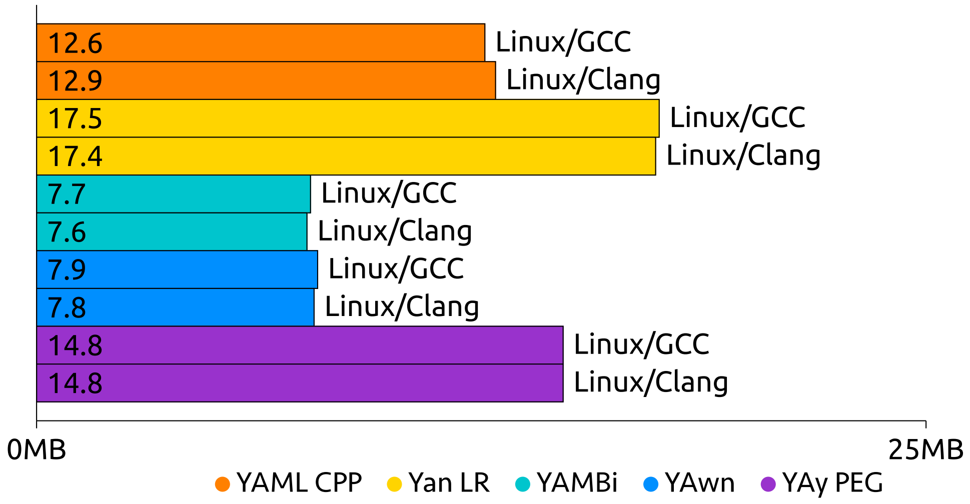


Figure 4.9: This bar chart shows the peak heap memory usage of the plugins for the input `combined.yaml`.

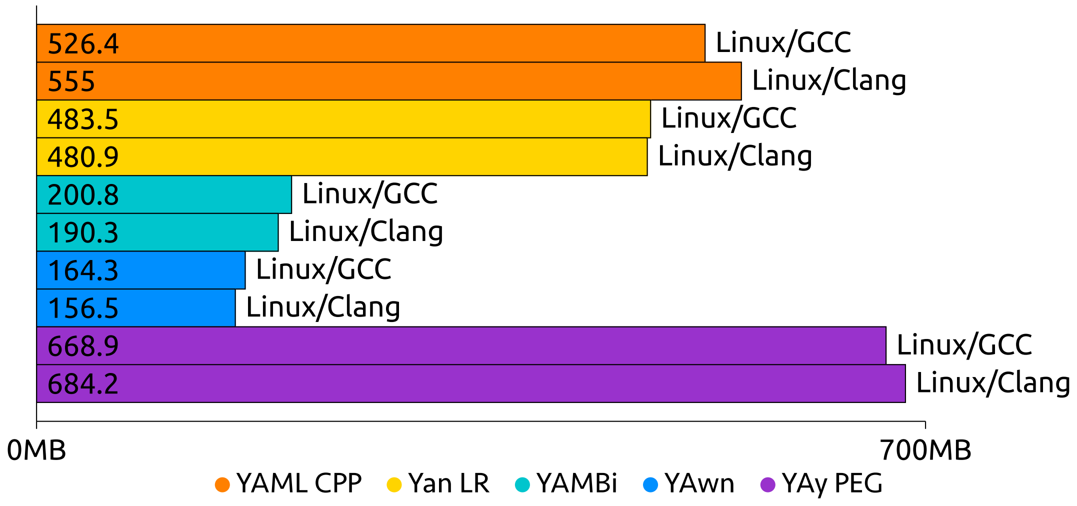


Figure 4.10: This bar chart shows the peak heap memory usage of the plugins for the input `generated.yaml`.

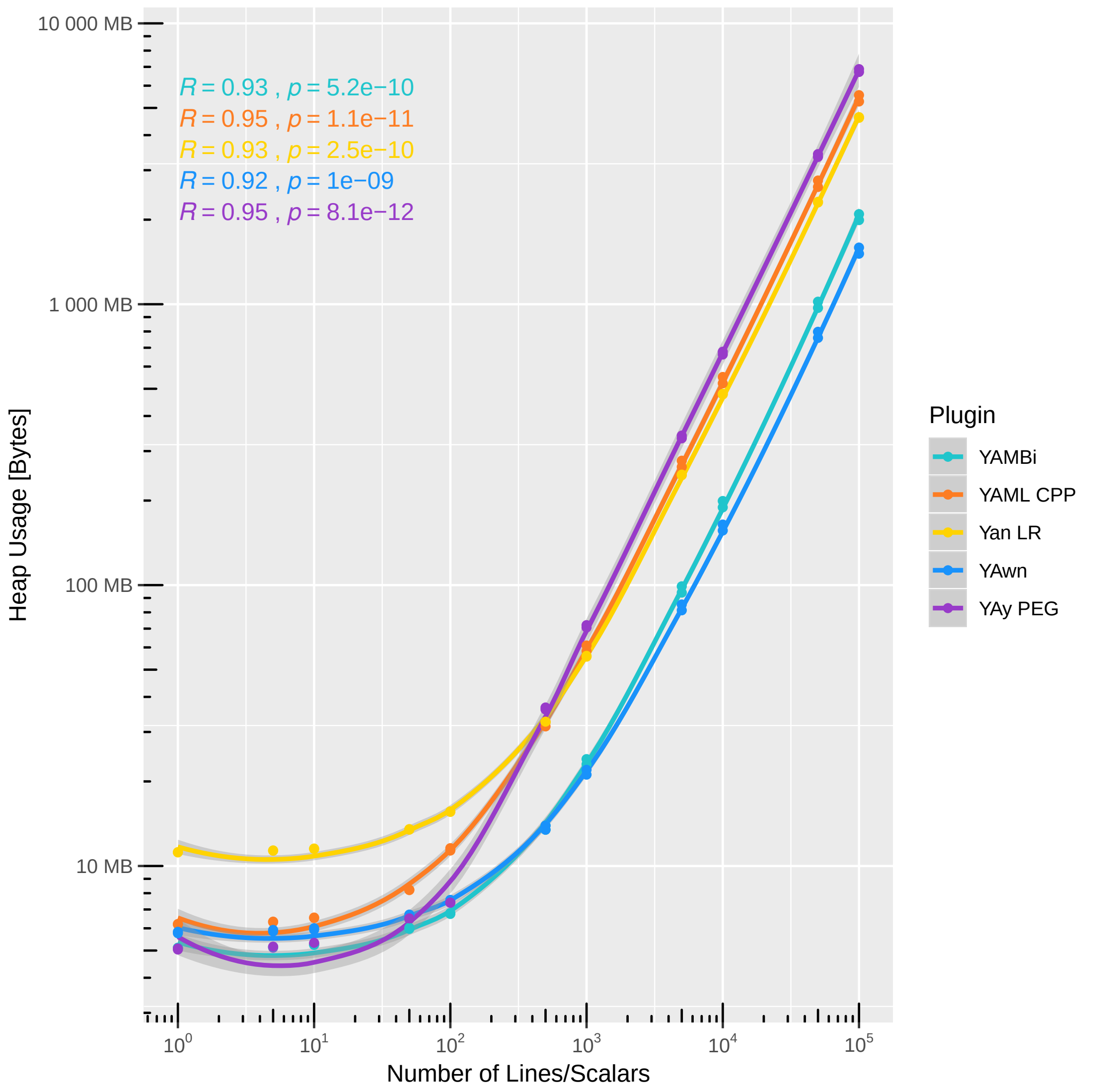


Figure 4.11: The diagrams above show the peak heap memory usage of the plugins for the input file `generated_100000.yaml` and other files that contain only the first n number of lines of this file.

4.3.3.3 Analysis

Figure 4.11 shows that the memory usage seems to grow linearly for large line numbers. To analyze this behavior further, we limited the data for the graph to observation where the line number is 1000 or higher. Figure 4.12 shows the graph after this modification. From the correlation coefficients (R) of 1 and the low probabilities of the null hypothesis being wrong (p) we conclude that the memory consumption almost certainly grows linearly for all plugins.

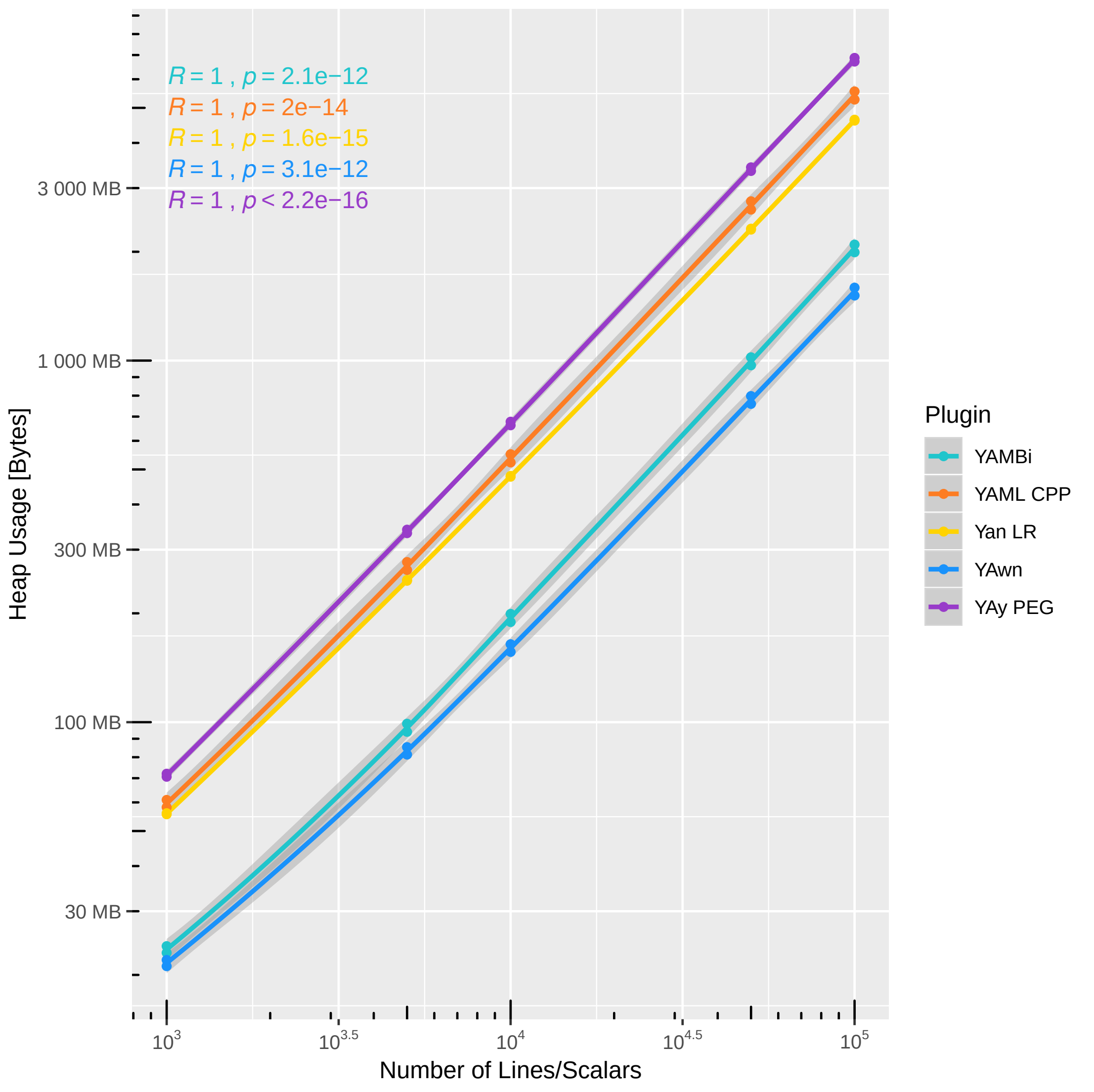


Figure 4.12: This diagram above shows that the memory usage of the YAML plugins almost certainly grows linearly for the first n lines of the file `generated_100000.yaml`

4.3.3.4 Conclusion

We conclude the subsection about the memory evaluation, by answering RQ 2.

? **RQ 2.** *How does the peak memory usage of the algorithms compare to each other? Do some of the algorithms show nonlinear memory usage?*

While the asymptotic memory usage of all plugins grows linearly according to our measurements, the factors between the memory usages can be quite high. If we look at the data for the file `generated.yaml` (see Figure 4.10), then YAwn performs best, while YAMBi needs about 20% more heap memory. The memory usage of the other plugins is much worse. Yan LR allocates about three times the heap memory of YAwn, YAML CPP takes about 3.4 times the memory amount of YAwn, and YAy PEG needs even more than 4 times the heap memory of YAwn.

4.4 Code Size

4.4.1 Method

We created a Shell script called `count-lines` that counts the code lines of the YAML plugins using the tool cloc. Since cloc does not support Bison grammar files, we wrote code that removes comments and empty lines from a Bison grammar, and afterwards counts the remaining lines using the Unix tool `wc`.

4.4.2 Results

Figure 4.13 shows the results reported by `count-lines`. As you can see, we did not have to write a grammar for the library based YAML CPP plugin. For the other plugins, we either created a grammar (Yan LR, YAMBi, YAwn), or specified the grammar as handwritten code (Yay PEG). Only the ANTLR (Yan LR) and Bison (YAMBi) based plugins use generated code. YAwn, which is based on YAEP, uses the grammar directly.

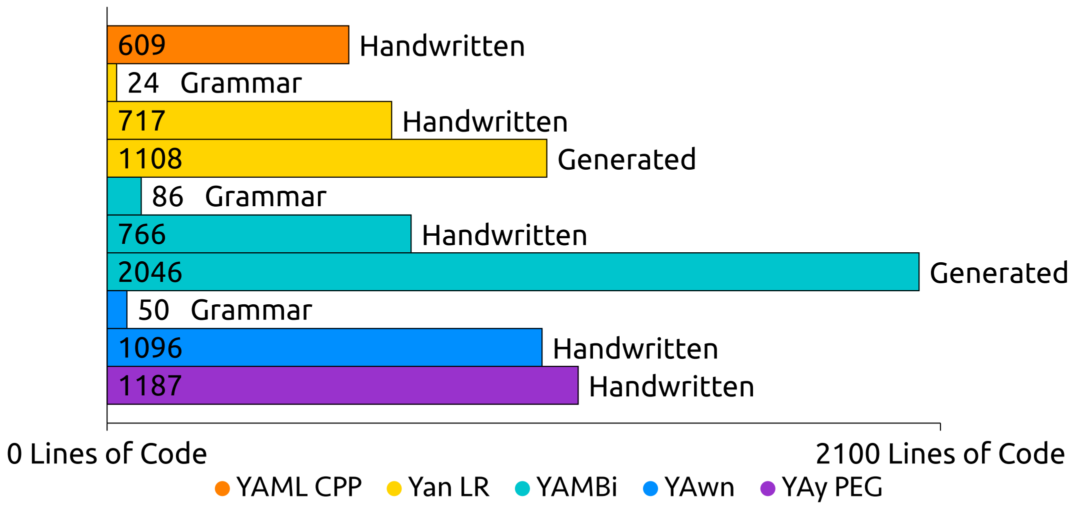


Figure 4.13: This bar chart shows the line counts of of the YAML plugins.

4.4.3 Analysis

Figure 4.13 shows the individual line counts and therefore answers the first part of RQ 3.

? **RQ 3.** *How much work does it require to implement the plugins, i.e. how many lines of code do we have to write to support our YAML subset for each parsing engine? How do the amounts of handwritten code for the plugins compare to each other?*

To answer the second part of the question we have to analyze the individual code sizes further.

Figure 4.13 show that YAML CPP requires the least amount of code. This results is not surprising, considering that the plugin uses yaml-cpp, a library that already represents the YAML file using high-level abstract data structures. We can convert these data structures into Elektra's `KeySet` structure relatively easily, keeping the code size of the plugin small.

For a fair size comparison we also have to take the library code itself into consideration. We use the command

```
cloc --include-lang='C++,C/C++ Header' src include
```

to determine the code size of version 0.6.2 of the yaml-cpp library. The command reports 8413 lines of code, showing us that parsing the full YAML standard requires a relatively large amount of code.

If we look at the other YAML plugins, we had to write the least code for Yan LR, followed by YAMBi, YAwn, and YAy PEG. The higher line counts of YAMBi and YAwn compared to Yan LR are not that surprising, considering that we implemented some functionality already provided by ANTLR, for those plugins ourselves.

For YAMBi we wrote two classes,

- one that represents the input for the lexer (63 code lines), and
- another class that stores data about a lexer symbol (53 code lines).

The sum of the lines of code ($63 + 53 = 116$) explains the difference between the amount of handwritten code between Yan LR and YAMBi ($717 - 609 = 108$) pretty well.

For YAwn we also added a symbol (69 code lines) and input class (101 code lines) as we did for YAMBi. Additionally we added

- tree walking code (124 code lines),
- a listener (126 code lines),
- an error listener (94 code lines), and

- classes storing positional information (38 code lines).

The 552 code lines mentioned above are responsible for about half of the code of the whole plugin ($\frac{1096}{2} = 548$).

YAy PEG, with its scanner-less parsing engine based on C++ templates, uses the most handwritten code. Just like for YAwn we wrote

- tree walking code (128 lines of code), and
- a listener (122 lines of code)

for the plugin. The other handwritten code take care of parsing (822 code lines) and the communication between the plugin and Elektra (115 code lines). The amount of parsing code of YAy PEG is quite large, compared to the grammar code of the other plugins. However, we have to consider

- that YAy PEG's parsing code supports a subset of YAML that is a little bit larger, than the one of the lexer based parsing plugins (Yan LR, YAMBi, YAwn), and
- that the parsing code also takes care of the work usually done by a lexer.

If we look at the lexer based plugins, they use between 352 (YAMBi) and 416 (Yan LR) code lines for the lexer.

Handwritten code requires manual work, and is therefore the most interesting criteria when we compare the code size of the plugins. However, the difference between the amount of generated code between the ANTLR based Yan LR plugin (1108 code lines), and the Bison based YAMBi plugin (2046 code lines) is also interesting. One reason for the big difference might be that ANTLR also requires a runtime library, while Bison generates all code needed for the parser. Both of these approaches have advantages and disadvantages. While Bisons' approach means no additional dependencies, ANTLR's runtime library provides space advantages. If multiple programs on the same machine use an ANTLR based parser, they can use the same compiled code, which only has to be stored in memory and on disk once.

4.4.4 Conclusion

If we take the amount of handwritten plugin code as main criteria for the code size comparison, then YAML CPP takes the lead, requiring the least amount of code. This is not surprising, considering that the parsing code of the plugin is part of the external yaml-cpp library, and not part of the plugin code itself.

The parser based plugin with the least amount of code is the ANTLR based Yan LR. The main reason for this is, that ANTLR already generates code for functionality that we had to create ourselves for the other plugins. While writing this support code is usually not that hard, it is certainly an advantage that ANTLR already provides support for common tasks, such as tree walking.

4.5 Code Complexity

4.5.1 Method

For the code complexity analysis we measured the Cyclomatic Complexity (CC) of

- the parsing libraries and generators,
- the generated code (Yan LR, YAMBi), and the
- handwritten plugin code

with the analyzer lizard. Since we had to measure the complexity of many different code parts and we wanted to improve the reproducibility of the measurement we created a script for this task called `measure-complexity`.

4.5.2 Results

Table 4.4: The table below shows the measurement results of the script `measure-complexity`.

Plugin	Part	NLOC	Average CC	Warnings	Function RT	NLOC RT
YAML CPP	yaml-cpp	7841	2.5	7	0.01	0.09
	Plugin	556	4.6	0	0	0
Yan LR	ANTLR C++ Runtime	15760	2.3	18	0.01	0.15
	Plugin	610	2.1	0	0	0
	Generated Code	1105	1.4	0	0	0
YAMBi	Bison	12677	4.8	22	0.04	0.28
	Plugin	667	2.3	0	0	0
	Generated Code	1526	2.9	8	0.07	0.42
YAwn	YAEP	5944	3.9	10	0.04	0.28
	Plugin	990	2.5	0	0	0
YAy PEG	PEGTL	8858	1.6	4	0.01	0.05
	Plugin	1094	2.9	0	0	0

CC...Cyclomatic Complexity NLOC...Noncommented Lines Of Code

$$\text{Function RT} = \frac{\text{Warnings}}{\text{Number of Functions}} \qquad \text{NLOC RT} = \frac{\text{NLOC with Warnings}}{\text{NLOC inside Functions}}$$

4.5.3 Analysis

The most interesting parts of Table 4.4 are the last two columns that show the relative amount of code that has a higher cyclomatic complexity than 15. None of the handwritten plugin code contains any function with a CC over this threshold. This is the result of using the static code analyzer OCLint to check the code while we developed the plugins. Other than that, only the code generated by ANTLR contains no code with a cyclomatic complexity over 15. The cyclomatic complexity of the generated code by Bison is higher, which does not seem that surprising considering that LR parsers, contrary to LL parser, are almost never written by hand, because of their inherent complexity. If we look at the parser libraries and parser generators themselves, PEGTL is the library containing the least amount of code over the complexity threshold, followed by yaml-cpp, and ANTLR's C++ runtime. Bison and YAEP are the generator and library that contain the most code with a high cyclomatic complexity.

4.5.4 Conclusion

While cyclomatic complexity has "never been unambiguously correlated with defective or unmaintainable code" [Mar+17], the measurements in this section provide at least some indication about code that might be problematic due to high code complexity.

4.6 Ease of Extensibility and Composability

One advantage of parsing libraries and parser generators over handwritten parsers is that we can update the language grammar without having to rewrite parsing code. This way we can extend the parsed YAML subset easily without many manual code changes. In the first part of the next section we analyze how many code line changes it took to fix bugs in, and add minor features to, the YAML plugins. The amount of code changes provides a good metric on how much effort it takes to extend the parser plugins.

In the second part of this section we will take a look at how the composability of our parsing code influences the extensibility. One option to create an extensible system is to base it on components. We can reuse these components to keep the amount of code for a new feature or bug fix low. We can imagine that the rules of a grammar represent components in our parsing systems. Parsing systems without a separate lexing phase such as PEGTL take the composability idea one step further. In PEGTL we can compose the parser for the whole grammar out of smaller parser that build on each other. We will analyze if the component based parser of YAy PEG provides extensibility advantages over the other parsers. We will also answer RQ 4 in this part of the thesis.

? **RQ 4.** *Which parsing technique allows us to stay closest to the definition of the configuration language? Does staying close to the given definition allow us to extend and improve the parser and its support code more easily?*

4.6.1 Plugin Updates

In the next subsections we look at the effort it took to add certain features to, and fix certain bugs in, the YAML plugins.

4.6.1.1 Method

As measurement for the extendibility effort we use the amount of changed code lines. Since we want to keep the comparison fair, we only look at the code changes needed for a certain feature or bug fix, excluding additional test code and documentation updates.

4.6.1.2 Support for Elektra's Boolean Data Type

Elektra's `Key` data structure usually saves data as untyped character string. We can add type information for a certain key by adding a `type` meta key (see also Section "`KeySet`"). If we do that Elektra ensures that applications store and retrieve the right kind of data for that specific key. Elektra's C++ API offers a direct way to store and retrieve a typed value via templated functions. We used these functions to improve the support for boolean data in the YAML plugins.

YAML's JSON schema represents boolean data as scalar with the canonical value `false` or `true` [BEN09]. More advanced schemas, such as the core schema offer additional aliases for true and false values. For the YAML plugins in this thesis we only added support for the JSON schema though. For this to work we had to translate the YAML values `false` and `true` to Elektra's boolean values `0` and `1`.

Just like Elektra, yaml-cpp, the library used by the YAML CPP plugin, also offers templated functions to retrieve and set boolean values. One problem of yaml-cpp's API is that there seems to be no way to check for the type of a YAML node, without the possibility of throwing an exception [Bed13]. This can be problematic for the runtime efficiency, since YAML CPP might trigger multiple exceptions before it converts a YAML node to a correctly typed Elektra key. To improve the runtime performance of YAML CPP we checked the textual value of a YAML node before we converted it. The implementation of this more complicated approach modified 16 code lines (15 additions, 1 deletion), while the implementation of the direct approach – that might cause more exceptions for data without many boolean values – modified only 9 additional code lines (8 additions, 1 deletion).

Adding support for boolean values modified 10 lines in Yan LR's code base (9 additions, 1 deletion), and 9 lines (8 additions, 1 deletion) in each of the other plugins (YAMBI, YAwn, YAy PEG). The similar line counts are a direct result of all of the plugins using a listener interface to convert parsed YAML data. To add boolean support we only had to change code in one of the functions of the listeners.

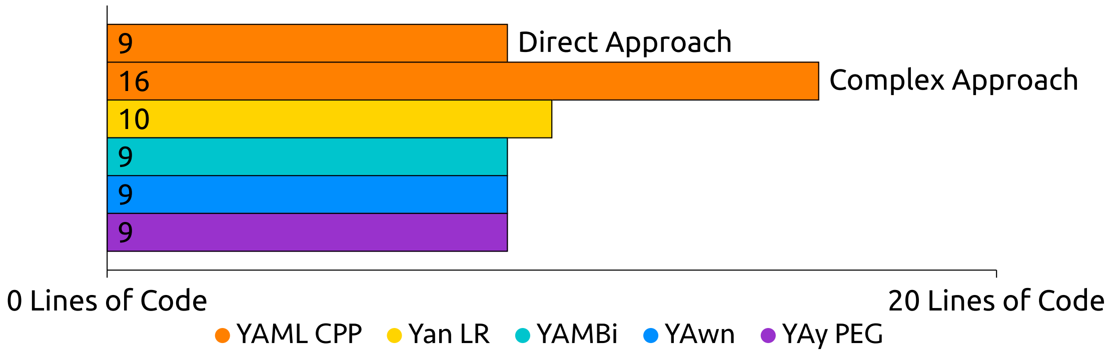


Figure 4.14: This bar chart shows the number of modified lines needed for adding better boolean support to the YAML plugins.

4.6.1.3 Conversion of Empty Values

Depending on the context, empty content (empty nodes) in a YAML stream might represent null values. Elektra should store these null values in a key with a zero length binary value. This was not the case in the first version of the parser based YAML plugins (Yan LR, YAMBi, YAwn, YAy PEG). Instead the plugins would incorrectly convert these null values into empty strings.

Another similar problem was that the lexer based plugins would not convert empty content null values right before the end of a file (`EOF`). The result of this bug was that the lexer would not terminate for a stream such as the one shown in Figure 4.15.

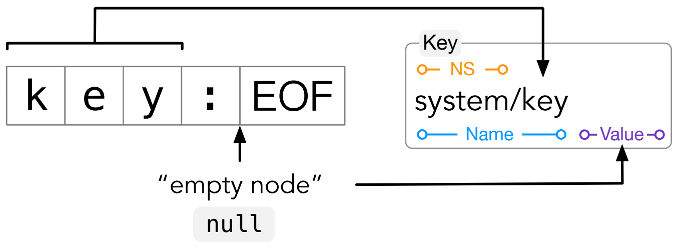


Figure 4.15: The YAML data on the left represents a map that contains one key-value pair with the name `key` that stores a null value. The `Key` structure on the right shows the converted YAML data, if we store it directly below the Namespace (NS) `system`.

Figure 4.16 shows the amount of changed lines for the Yan LR, YAMBi, YAwn, and YAy PEG plugin. The code changes for all the lexer based plugins (Yan LR, YAMBi, YAwn) are almost identical, since we had to fix the problems in the lexer and listener code, which is quite similar for all of those plugins.

For YAy PEG we only had to fix the support for empty nodes in the middle of a YAML stream, since YAy PEG's parsing code already handled empty nodes at the end of a file correctly. This is probably a direct result of using grammar code that is quite similar to the one from the YAML specification. The notable difference between the amount of code we added for the empty node fix (5 additions, 1 deletion) compared to the one of the lexer based plugins (1 addition) is also a result of YAy PEG's grammar. YAy PEG uses the same code path to add empty and non-empty mapping values. The lexer based plugins on the other hand check for an empty value inside the listener code for a key-value pair. We could also use this approach in the YAy PEG plugin, but that would possibly require substantial code changes to the tree walking code.

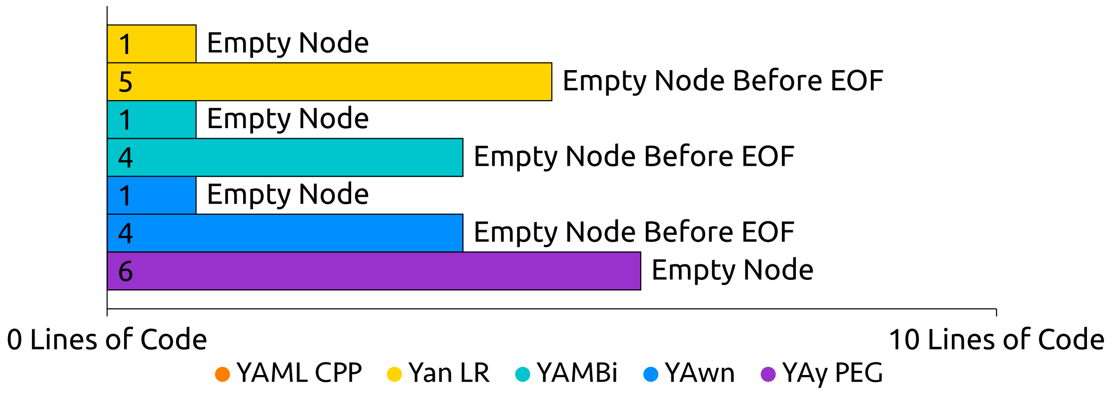


Figure 4.16: This bar chart shows the additional lines needed for fixing the null value support of the YAML plugins.

4.6.1.4 Conversion of Empty Documents

According to the YAML specification an empty file corresponds to a null value. We did not consider this in the initial versions of the YAML plugins, which meant the empty documents shown in Figure 4.17 were translated incorrectly. Figure 4.18 shows the amount of code lines we modified to fix this problem for the Yan LR, YAMBi, YAwn and YAy PEG plugin.

```yaml

```

```yaml
# Empty document
```

Figure 4.17: The examples above show two options on how to store “nothing” (`null`) using YAML.

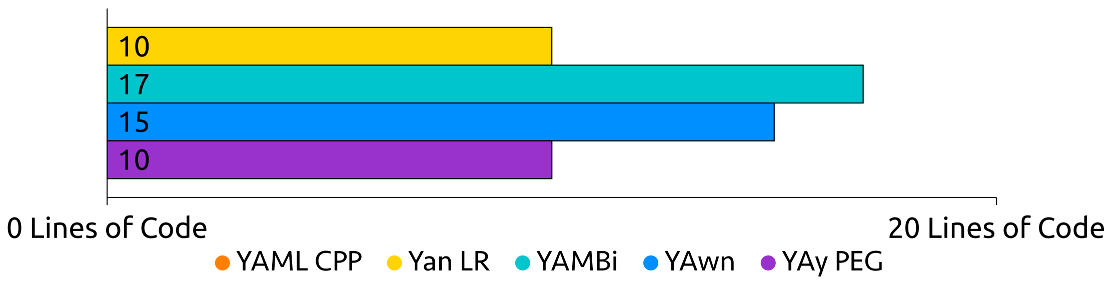


Figure 4.18: This bar chart shows the amount of code lines we modified to fix the conversion of empty documents.

When we implemented the fixes for the empty document conversion we noticed that we added nearly identical code to the listener (Yan LR, YAwn, YAy PEG) respectively driver (YAMBi) of the plugins. These modifications took 6 additions for Yan LR and 5 additions for the other plugins. We should mention here that we could also have avoided two additional lines for the Yan LR plugin, which consisted of a `using` statement and the inclusion of an optional header file.

The other code line differences were a result of grammar updates to

- Yan LR (3 additions, 1 deletion),
- YAMBi (7 additions, 5 deletion), and
- YAwn (5 additions, 1 deletion)

and updates to tree walking code of:

- YAwn (4 additions), and
- YAy PEG (5 additions).

The relatively high number of changes to the grammar of YAMBi is deceiving, since we also moved a 4 line grammar block (4 additions, 4 deletion) in the bug fix update for the plugin. Figure 4.19 takes this code block movement and the optional line changes for Yan LR into account to give a better overview of the code changes we needed to implement the bug fix.

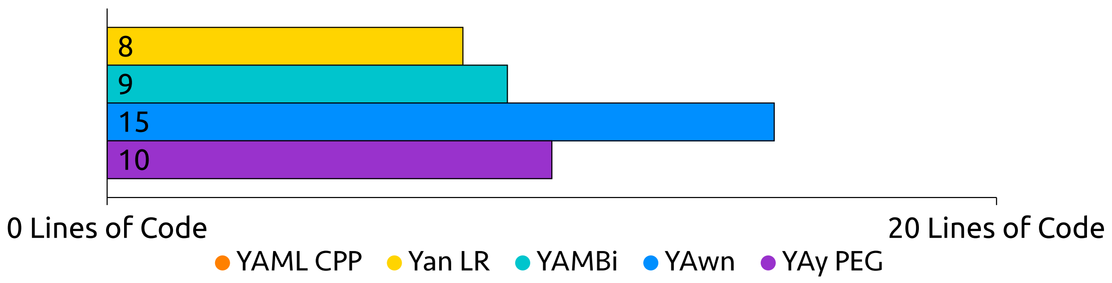


Figure 4.19: This bar chart shows the "minimal" code line modifications we needed to fix the conversion of empty documents.

4.6.2 Component Based Grammars and Extensibility

The YAML specification includes a detailed grammar description of the language in a parameterized BNF like syntax. The grammar rules of the specification are very reminiscent of parser combinator functions [Hut92; HM96]. This is not that surprising considering that YAML's reference parser is also based on parser combinators. A large part of the reference parser actually uses slightly modified versions of all of the rules of the YAML spec. Since "the order of alternatives in the grammar is significant" [BEN09] (ordered choice) and some of the parsing rules also uses positive and negative lookahead we can also categorize YAML's reference grammar as an extended version of a PEG.

In the section "PEG Parser" we already mentioned that the grammar description of the YAy PEG plugin resembles the YAML specification grammar quite closely. It is time to analyze, if this close resemblance provides advantages over the other YAML plugins, which use a description that is quite different to the one of the YAML specification.

Table 4.5: The table below lists some advantage and disadvantages of PEG based (YAy PEG) and lexer based (Yan LR, YAMBi, YAwn) plugins regarding extensibility.

● YAy PEG	●●● Lexer Based Plugins
+ Grammar Extension via "Copy & Paste"	+ Simple Grammar
− Complicated Grammar	− Handwritten Lexer
− Debugging	

Using the PEG library certainly offers advantages considering the extensibility of the grammar, since we can more or less copy the grammar rules from the specification or reference parser and modify them slightly for PEGTL. Fortunately the reference parser already contains a relatively large test suite, which means the chance of errors in the reference grammar is quite low. This is helpful, since the specification grammar consists of 211 rules, which can be quite complicated. This is also one of the disadvantages of using PEGTL, since the grammar of YAy PEG is more complicated than the one of the lexer based plugins. The complexity is a result of using a single pass to parse YAML data instead of using a separate lexer and parser phase. The single parsing phase also makes debugging harder since we cannot debug the lexer and parsing code separately.

Keeping the information above in mind we can now answer RQ 4.

? RQ 4. *Which parsing technique allows us to stay closest to the definition of the configuration language? Does staying close to the given definition allow us to extend and improve the parser and its support code more easily?*

The parser of YAy PEG certainly stays closest to the grammar definition of the YAML specification. This closeness is helpful, if we look at the extension of the grammar. However, since YAy PEG's grammar code also handles low-level details of the parsing process we have to consider these details later in the parsing process, which reduces the extensibility of the parsing support code compared to the one of the lexer based plugins.

4.6.3 Conclusion

The examples at the start of this section show that the extensibility of the different YAML plugins depend on the specific bug we want to fix or the feature we like to add. Sometimes the code changes can be quite similar, at least for all the lexer based plugins (Yan LR, YAMBi, YAwn). Other times we need to change code in nearly all parts of a plugin. In these cases plugins that use generators and libraries that provide more built-in code support are easier to extend. If we take the built-in support code into account, then the order of ease of extensibility for the lexer based plugins is roughly Yan LR, followed by YAMBi and then YAwn.

YAML CPP's extensibility depends on the specific part we need to extend. While we can modify the conversion code of the plugin easily, fixing bugs in the lexer or adding features, such as comment preservation would require us to change the library code of yaml-cpp. This would take more effort, than it would for the other YAML plugins for a similar feature, since we would have to update yaml-cpp's handwritten parser code instead of the code of a more compact grammar file.

YAy PEG's advantage considering grammar extensibility is that the plugin uses parsing code that is very similar to the one of the YAML specification. This allows us to extend the grammar relatively easily by taking rules from the YAML specification and modifying them slightly. The similarity of the grammar code can also be a disadvantage though, since the support code of the plugin needs to consider the many rules of the specification grammar compared to the relatively simple grammar rules of the lexer based plugins.

4.7 Error Reporting

While there exist techniques to enhance error reporting, by using external tools or modifying a parser engine (see Section "Error Handling"), we will only consider built-in solutions or slight modifications to a grammar. We do this, since extending a parser engine is out of scope of the thesis and elaborate extensions would also make the comparison concerning error reporting unfair.

4.7.1 Initial Erroneous Input

Listing 4.1 shows the erroneous YAML data we used initially to compare the error reporting capabilities. Listing 4.2 and 4.3 show two solutions to fix the problematic part of the YAML document.

```yaml
key:
  - element 1
- element 2
```

Listing 4.1: The indentation of the sequence item `- element 2` is incorrect in the code above. One of the most obvious solutions to fix the syntax error would be to add a single space character right before `- element 2` (see Listing 4.2). Another solution is to remove `- element 2` altogether (see Listing 4.3).

```yaml
key:
  - element 1
  - element 2
```

Listing 4.2: Usually a person would fix the error shown in Listing 4.1 by adding an indentation character before the sequence item `- element 2`.

```yaml
key:
  - element 1
```

Listing 4.3: One of the easiest solutions to fix the code in Listing 4.1 for a computer program is to remove `- element 2`.

4.7.2 Basic Error Messages

We started the comparison by listing the basic error messages for the YAML plugins. These messages contain the error location and the auto-generated error text by the parsing engines. For the sake of brevity we removed data that is identical for all plugins, such as the filename of the parsed file.

Table 4.6: Basic error messages

Plugin	Parser	Message
YAML CPP	yaml-cpp	yaml-cpp: error at line 3, column 1: end of map not found
Yan LR	ANTLR	3:1: mismatched input '- ' expecting BLOCK_END
YAMBi	Bison	3:1: syntax error, unexpected ELEMENT, expecting KEY or BLOCK_END
YAwn	YAEP	3:1: Syntax error on token number 9: “<Token, ELEMENT, -, 3:1–3:2>”
YAy PEG	PEGTL	3:0(18): parse error matching tao::yaypeg::eof

4.7.2.1 Interpretation

As we can see in Table 4.6 all of the parsing engines report the error location for the code from Listing 4.1 correctly. The error messages also shows that the question whether the first position after a newline is at column 0 or 1, is still open for debate. We also see that YAML CPP, Yan LR, and YAMBi show information about the expected element at the error position (end of a block collection is missing). YAML CPP provides the best error message, since the plugin also shows which type of end element is missing (end of map). This type of information can also be determined easily in all of the lexer-based parsing engine plugins (Yan LR, YAMBi, YAwn). We modified them accordingly. Table 4.7 shows the slightly improved error messages, highlighting the updated part of the text.

Table 4.7: Slightly improved error messages

Plugin	Parser	Message
YAML CPP	yaml-cpp	yaml-cpp: error at line 3, column 1: end of map not found
Yan LR	ANTLR	3:1: mismatched input '- ' expecting **MAP_END**
YAMBi	Bison	3:1: syntax error, unexpected ELEMENT, expecting **MAP_END** or KEY
YAwn	YAEP	3:1: Syntax error on token number 9: “<Token, ELEMENT, -, 3:1–3:2>”
YAy PEG	PEGTL	3:0(18): parse error matching tao::yaypeg::eof

After the slight modifications to the YAML parser plugins we decided to take a closer look at the error handling capabilities of each of the parsing engines on their own in the next subsections.

4.7.3 ANTLR

ANTLR uses an error listener class that provides a callback method that includes access to

- the location,
- the offending symbol,
- the used recognizer class,
- the thrown exception, and
- the default error message

for each detected error. As we already saw in Table 4.7, the default error message provided by ANTLR usually describes an error already well. For the initial version of the Yan LR plugin, we only stored the last error message reported by ANTLR. Since ANTLR uses methods such as token deletion and insertion to keep parsing a file, even if it contains multiple errors [Par13b], the last error message usually will not provide the most obvious information on how to fix an error.

```
key: - element 1
  - element 2 # Incorrect Indentation!
```

Listing 4.4: The indentation of the sequence element `- element 2` is incorrect in the code above.

For example, for the input shown in Listing 4.4 the parser produced the following error output:

```
2:37: extraneous input 'MAP END' expecting {STREAM_END, COMMENT}
```

To fix this defect in the Yan LR plugin we stored all error messages, which resulted in the better error report:

```
2:1: mismatched input '- ' expecting MAP_END
2:37: extraneous input 'MAP END' expecting STREAM_END
```

You might also notice that in the error report `COMMENT` is missing from the list of expected tokens. This difference is the result of an ambiguity in the ANTLR grammar we fixed.

One of the more recent improvements in error messages of modern compilers such as Clang and GCC is the ability to highlight erroneous input. We also implemented this error reporting mechanism based on the Java code in *The Definitive ANTLR 4 Reference*, page 158 [Par13b]. The text:

```
2:1: mismatched input '- ' expecting MAP_END
     - element 2 # Incorrect Indentation!
     ^^
2:37: extraneous input 'MAP END' expecting STREAM_END
      - element 2 # Incorrect Indentation!
                                        ^
```

shows the improved error message for Listing 4.4. One thing that this error report still lacks is a more human friendly representation of the tokens. Someone with limited knowledge of the YAML specification and Yan LR's lexer code will probably not know what `MAP_END`, `MAP END` and `STREAM_END` mean. One option to improve this situation is to replace the text used by the lexer (`MAP END`) and the parser (`MAP_END`, `STREAM_END`). The update of the relevant lexer code is trivial, since we can create tokens containing arbitrary text. For the parser code generated by ANLTR, we used a script that uses regular expressions to replace the relevant strings such as `"MAP_END"` and `"STREAM_END"`. After this update the error report for the YAML data in Listing 4.4 looks like this:

```
2:1: mismatched input '- ' expecting end of map
     - element 2 # Incorrect Indentation!
     ^^
2:37: extraneous input 'end of map' expecting end of document
      - element 2 # Incorrect Indentation!
                                          ^
```

4.7.4 Bison

In the first improvement step for the error messages of the Bison parser, we defined alternative names for tokens, just as we did for Yan LR. Bison supports this feature directly, which means we did not have to write a script to replace the symbols in the generated parser code. After this update the error message from Table 4.7 changed from:

```
3:1: syntax error, unexpected ELEMENT, expecting MAP_END or KEY
```

to

```
3:1: syntax error, unexpected element, expecting end of map or key
```

We then looked into the error recovery capabilities of Bison. Unlike ANTLR the generated parser does not do error recovery by default, but rather exits on the first error. To improve the error behavior, Bison offers the possibility to add the predefined `error` token to a grammar. Every time the Bison parser encounters an error it will produce this token [DS19]. We modified the grammar to allow errors inside YAML maps and sequences:

```
pairs : pair
      | pairs pair
      | pairs error /* Allow errors after key-value pairs */
      ;

elements : element
         | elements element
         /* Allow errors after elements of a sequence */
         | elements error
         ;
```

This way the parser is able to report multiple syntax errors.

```
key 1: - element 1
 - element 2
key 2: scalar
      - element 3
```

Listing 4.5: The indentation of the sequence item `- element 2` is incorrect in the code above. Another error is that the value of `key 2` can not be both a scalar (`scalar`) and a sequence (containing `- element 3`).

After the update the parser produces an error message that looks like this:

```
2:2: unexpected start of sequence, expecting end of map or key
      - element 2
      ^
4:8: unexpected start of sequence, expecting end of map or key
            - element 3
            ^
```

for the input shown in Listing 4.5. As you can see above, we also added the erroneous input to the error message, just as we did in the Yan LR plugin.

4.7.5 YAEP

Just as Bison, YAEP also requires that we add error tokens to the grammar to specify locations for error recovery. We therefore defined the same error recovery locations inside sequences and maps, as we did for Bison. The other updates were quite similar too: We improved the name of tokens inside error messages and added the erroneous input to the error message.

After all these changes the output for the YAML data from Listing 4.5 looks very similar to the one produced by YAMBi:

```
2:2: Syntax error on input “start of sequence”
      - element 2
      ^
4:8: Syntax error on input “start of sequence”
            - element 3
            ^
```

The only thing missing is the information about the expected type of token.

4.7.6 PEGTL

We already talked about general error strategies for PEG parsers in the Section “Error Handling”. PEGTL does neither implement the error handling strategy described by Ford [For02], nor labelled failures [Mai+16]. Instead the library offers a grammar rule called `must`, which states that a certain rule, specified as template argument, has to match at a given position or an error will be raised. We can customize the code executed for a given `must` rule according to this template argument. Effectively this strategy allows us to specify different error messages for each expected but unmatched rule.

As we described in the section “PEG Parser”, we tried to keep the grammar of our PEG parser plugin YAy PEG close to the grammar of the YAML specification [BEN09]. This also meant that the grammar contained only a single `must` rule that makes sure that the grammar matched the whole input:

```
struct yaml : if_must<l_yaml_stream, eof> {};
```

The code above also explains the initial version of the error message shown in Table 4.7:

```
3:0(18): parse error matching tao::yaypeg::eof
```

which tells us that the parser was unable to match the expected “end of file” in line 3 of the input. We customized the error message above to show a more user friendly text:

```
3:0: Incomplete document, expected “end of file”
     - element 2
     ^
```

As you can see we also added the erroneous input to the message, just as we did for the other parsing engines.

The same single error message, regardless of the error, is not helpful. For good error reporting we need to add other `must` rules. However, adding failure points (`must` rules) changes the behavior of the grammar and might even cause the parser to fail on valid input. To minimize the probability of incorrect grammar changes we only added a few rules for situations we were sure that the remainder of a grammar rule had to match. For example, when the parser reads an unescaped single or double quote character (outside of a block scalar) at the beginning of a line or after a white space character, it found a quoted flow scalar. Therefore

1. the text after the initial quote has to be followed by a (possibly empty) text containing only certain characters, and
2. the last character of the flow scalar has to be an unescaped quote.

If one of those two rules is not fulfilled, then the parser found a syntax error. After we updated the code accordingly the error message for the YAML data

```
"double quoted
```

looks like this:

```
1:14: Missing closing double quote or
      incorrect value inside flow scalar
      "double quoted
                    ^
```

As you may have noticed we included both error possibilities in the error message, since reacting to both errors independently would require fundamental changes to the grammar.

4.7.7 Final Error Messages

4.7.7.1 Element Outside of Sequence

Table 4.8 shows the final error messages for the code of Listing 4.1:

```
key:
 - element 1
- element 2
```

Table 4.8: Final error messages for the YAML code of Listing 4.1

Plugin	Error Messages
YAML CPP	`error at line 3, column 1: end of map not found.`

Yan LR	`3:1: mismatched input '- ' expecting end of map`
	`     - element 2`
	`     ^^`

YAMBi	`3:1: syntax error, unexpected element,`
	`     expecting end of map or key`
	`     - element 2`
	`     ^^`

YAwn	`3:1: Syntax error on input “-”`
	`     - element 2`
	`     ^^`

YAy PEG	`3:0: Incomplete document, expected “end of file”`
	`     - element 2`
	`     ^`

Interpretation

- All the parsing engines report the correct error location.
- YAy PEG and YAwn do not tell us in which YAML node the error occurs. All the other plugins report that a map ended prematurely.
- The error message of YAMBi reports an additional option – besides deleting the input – to fix the error: adding a key (in the line between `- element 1` and `- element 2`).

The list below shows a ranking of the plugins according to the interpretation of the error messages above.

1. YAMBi
2. YAML CPP, Yan LR
3. YAy PEG, YAwn

4.7.7.2 YAML Data Containing Multiple Errors

The YAML data from Listing 4.1 only contains a single syntax error. To compare the error recovery capabilities of the parsing libraries, we used YAML data that contains multiple syntax errors as input (see Listing 4.6).

```yaml
key 1:
  key 2:
    - element 1
    - element 2
    element 3 # Missing `- `
key 3: "double quoted scalar"
key 4:
  key 5:
    - element 4
    - element 5
     key 6: # Not on same level as key 5
      - element 6
key 7: 'single quoted scalar'
key 8:
  - element 7
scalar # Not a key
```

Listing 4.6: The YAML data above contains three syntax errors that we directly describe in the comments right next to the error positions.

Table 4.9 shows the error messages of the different storage plugins for the YAML input of Listing 4.6. As we can see the error output is quite different.

Table 4.9: Error messages for the YAML code of Listing 4.6

Plugin	Error Messages
YAML CPP	error at line 5, column 5: end of map not found.
Yan LR	5:5: mismatched input 'element 3' expecting end of sequence element 3 # Missing `- ` ^^^^^^^^^ 6:1: extraneous input 'end of sequence' expecting end of map key 3: "double quoted scalar" ^ 11:4: mismatched input 'start of map' expecting end of map key 6: # Not on same level as key 5 ^ 13:1: mismatched input 'end of map' expecting end of document key 7: 'single quoted scalar' ^
YAMBi	5:5: syntax error, unexpected plain scalar, expecting end of sequence or element element 3 # Missing `- ` ^^^^^^^^^ 11:4: syntax error, unexpected start of map, expecting end of map or key key 6: # Not on same level as key 5 ^ 13:1: syntax error, unexpected key, expecting end of document key 7: 'single quoted scalar' ^
YAwn	5:5: Syntax error on input “element 3” element 3 # Missing `- ` ^^^^^^^^^ 11:4: Syntax error on input “start of map” key 6: # Not on same level as key 5 ^ 13:1: Syntax error on input “key” key 7: 'single quoted scalar' ^ 16:1: Syntax error on input “scalar” scalar # Not a key ^^^^^^
YAy PEG	5:0: Incomplete document, expected “end of file” element 3 # Missing `- ` ^

Interpretation

- *YAML CPP* and *YAy PEG* do not provide any error recovery.
- *YAy PEG* shows the correct line number for the first error, but not the correct column number. The plugin also only provides a very generic error message.
- All of the plugins that use error recovery (Yan LR, YAMBi, YAwn) print a spurious error messages about an error at line 13.
- *Yan LR* shows two error messages for the first syntax error, and one for the second syntax error. Error messages one and three describe the problematic part of the YAML data reasonably well.
- Compared to Yan LR, *YAMBi's* (non-spurious) error messages also describe a second option to fix the erroneous input. However, while the first error message provides a useful suggestion on how to fix the error (insertion of a sequence element), the second option in the second error messages (insertion of a key), will probably confuse anyone that does not know how YAMBi's lexer works.
- *YAwn* prints the same error messages as YAMBi, without the crucial information about the expected element. In addition YAwn prints a fourth error message that addresses the third syntax error.

According to the interpretation we concluded that all of the plugins with error recovery provide about the same level of useful error information. YAML CPP describes the first error reasonably well, while the error message from YAy PEG is not that useful. This leaves us with the following ranking of the error capabilities of the plugins based on the input of Listing 4.6:

1. Yan LR, YAMBi, Yawn
2. YAML CPP
3. YAy PEG

4.7.8 Conclusion

We conclude this section by answering RQ 5.

? **RQ 5.** *What are the error handling capabilities of the parsing engines? How well can they handle multiple syntax errors? How do the generated error messages compare to each other?*

While the parsing libraries do not produce particularly great error messages, at least the ANTLR (Yan LR) and Bison (YAMBi) plugin, provide error messages that are comparable in quality to the ones of the handwritten parsing engine (YAML CPP).

One advantage of Yan LR, YAMBi, and YAwn is that their parsers offer error recovery. They are therefore able to report multiple errors in a file. This is something that YAML CPP is currently not able to do. ANTLR offers error recovery for free, while Bison and YAEP require the addition of error tokens to the grammar. This can be problematic, since these error tokens can produce conflicts in the case of Bison, and ambiguous parsing results in the case of YAEP.

The parsing plugin that showed the least useful error messages is YAy PEG. While the PEGTL offers basic error handling facilities that are able to provide good error messages for character level errors, producing good error messages for "high-level" errors would require a substantial amount of work.

4.8 Most Promising Plugin

Using the information we collected in this chapter it is time to determine the most promising YAML plugin. Let us start by saying that we think all of the parser based YAML plugins (Yan LR, YAMBi, YAwn, YAy PEG) could be extended to parse a more complete subset of YAML. However, some of them fit all the requirements we have for a more complete YAML plugin better than others. Since the whole evaluation is quite extensive we summarize general information about the used parser libraries in Table 4.10 and the concrete parsing plugins in Table 4.11.

Table 4.10: Overview of the used parsing libraries and their characteristics

	yaml-cpp	ANTLR	Bison	YAEP	PEGTL
Parser Techniques	• Recursive Descent	• SLL(*) • ALL(*)	• LALR • IELR • LR • GLR	• Earley Parser	• PEG Parser
Lexer Support	Handwritten	Integrated (Regex)	External	External	Integrated (PEG)
Grammar	• C++ Code	• Standalone • Annotated with Code	• Annotated with Code	• Annotated with AST Rewriting Rules	• Templated C++ Code
Conversion Interface	• Node Class	• Parser Actions • AST • Tree Walking (Listener & Visitor)	• Parser Actions	• AST	• Parser Actions[A] • AST
Input API (Encoding)	–	✅	❌	❌	✅
Token Handling	–	✅	✅	❌	✅
AST Support	–	✅	❌	✅	✅
Tree Walking	–	✅	❌	❌	❌
Tree Rewriting	–	❌	❌	✅	✅
Error Listener	–	✅	❌	❌	❌
Error Recovery	❌	✅	✅[E]	✅[E]	❌
Language Support	• C++	• C++ • C# • Go • Java • JavaScript • PHP • Python • Swift	• C • C++ • Java	• C • C++	• C++

[A] Since PEGs use backtracking the action code has to take “accidentally” taken actions into consideration.
[E] Error recovery support requires non-trivial changes to the grammar.

Table 4.11: Overview of the YAML parsing plugins and their characteristics

	YAML CPP	Yan LR	YAMBi	YAwn	YAy PEG
Parser Technique	Recursive Descent	ALL(*)	LALR(1)	Earley Parser	PEG Parser
Lexer Used	Existing Lexer	Custom C++ Lexer	Custom C++ Lexer	Custom C++ Lexer	Templated C++ Code
Handwritten Code	• Converter Functions[T]	• Lexer • Grammar • Listener [E] • Error Listener[E]	• Input Support • Token Support[E] • Lexer • Grammar • Driver[D]	• Input Support • Token Support • Location Info • Lexer • Grammar • Listener • Error Listener • Tree Walking	• Grammar • State • Listener • Error Listener • Tree Walking
Run Time[R]	188.9 ms	122 ms	119.9 ms	74.9 ms	261 ms
Memory Usage[M]	540.7 MB	482.2 MB	195.6 MB	160.4 MB	676.6 MB
Code Lines[L]	609	741	852	1146	1187
Extensibility & Maintainability[X]	★☆☆☆☆	★★★★☆	★★★☆☆	★★☆☆☆	★★☆☆☆
Error Recovery	❌	✅	✅	✅	❌
Error Messages[X]	★★☆☆☆	★★★☆☆	★★★☆☆	★★★☆☆	★☆☆☆☆
Main Development Time Frame[F]	6.8.2017 – 10.10.2017	6.2.2018 – 27.7.2018	25.8.2018 – 10.9.2018	11.9.2018 – 9.10.2018	26.9.2018 – 20.1.2019

[E] Here we extended generated code.
[F] The data in this row specifies the date from the initial commit of the respective plugin up to the first merged version of the plugin. The time frames do not include all the time necessary for research.
[D] The driver contains similar code that we implemented in the "Listener" and "Error Listener" parts of the other plugins.
[L] This number includes the line number of the grammar, but not any generated code, according to the data shows in Figure 4.13.
[M] This row shows the (rounded average) peak heap memory usage in MB of the plugins for the input `generated.yaml` according to the data shown in Figure 4.10.
[R] This row contains the mean of the execution times in milliseconds for the file `generated.yaml` show in Figure 4.7.
[T] We translated the node based structured output produced by yaml-cpp into a `KeySet`.
[X] This line shows an overall rating according to the conclusion section about the respective feature.

4.8.1 Requirements for an Extended YAML Plugin

Issue #2330 of Elektra's issue tracker specifies some of the desirable features for an extended YAML plugin. We list them below in their order of importance according to Elektra's maintainer.

1. **Round Trip:** Assume a plugin
   a) converts some YAML data to a `KeySet`, then
   b) writes back the converted `KeySet` to a YAML file, and afterwards
   c) reads the new YAML file again.

   In this scenario the `KeySet` read in step a) and step c) should be identical.
2. **Auxiliary Data:** Additional file data such as comments and ordering of keys should be kept, when the plugin adds new data to a YAML file.
3. **Maintainability & Modularity:** The code of the plugin should be readable and extendable. Specific tasks should be handled by specific parts of a plugin or other specialized plugins.
4. **Error Messages:** The plugin should provide good error messages for common mistakes.
5. **Line Information:** The plugin should store line information to provide this data to other plugins and users.

4.8.2 Elimination Process

To minimize the list of candidates for extension we first eliminate some of them according to the requirements listed in the previous section.

YAML CPP The library used by the plugin (yaml-cpp) does currently not store comments (2. requirement) or line information (5. requirement). Adding this kind of information should be possible, but would require large modifications of yaml-cpp's code base.

YAy PEG The error message information provided by the plugin is quite bad and does therefore not meet requirement 4.

After the elimination three possible candidates are left. It is now time to answer our main research question.

❓ **Main Research Question.** *Which parsing systems allows us to create a configuration parsing plugin that is easily extendable, has low maintainability cost and provides good error messages, while offering decent runtime performance and low memory overhead.*

Yan LR Compared to YAwn, the plugin performed worse in the runtime benchmark. It also requires more memory to parse our example data than YAMBi and YAwn. On the other hand, ANTLR provides a lot of support code we had to write for the other plugins ourselves.

YAMBi The plugin performed well in the runtime performance test on Linux and – compared to Yan LR – only required a relatively small amount of memory for our example data. The plugin does require more support code than Yan LR though. A problem considering the maintainability of the plugin might be the LR parsing algorithm. If there are any problems like shift/reduce conflicts in the grammar, then a developer usually needs at least some information on how the LR parsing algorithm works to fix the problem. Bison also offers considerable less support code for common parsing task than ANTLR (see Table 4.10), which can be a problem, e.g. if we are not able to only use parser actions for an extension of the YAMBi plugin.

YAwn While the plugin showed good runtime performance and the lowest memory usage, we found two disadvantages, compared to YAMBi and Yan LR.

1. YAEP is more or less the work of a single author: Vladimir N. Makarov. Recently another author, Alexander Klauer, fixed some of the problems of the project and modernized parts of the code base. However, compared to ANTLR and Bison, the community behind the parsing library is rather small.
2. YAEP requires more support code than Yan LR and even YAMBi. While this support code is not that complicated, it would still be something we need to maintain and extend for a more complete support of YAML and the requirements listed in "Requirements for an Extended YAML Plugin".

With all the information above in mind, we decided that the best candidate for extension is Yan LR. We think ANTLR's advantages, such as

- providing the most complete set of support code of all of the tested libraries and generators, and
- producing good error messages without any changes to the grammar,

make up for the worse runtime performance compared to YAEP, and the relatively large heap memory usage compared to Bison and YAEP.

CHAPTER 5

Conclusion & Future Work

5.1 Conclusion

In the thesis we compared different parsing techniques using the `KeySet` data structure of the configuration framework Elektra. The aim of this work was to find the most promising parsing technique for configuration files using a subset of the language YAML as example.

In a detailed evaluation we determined the answer to our auxiliary research questions.

❓ **RQ 1.** *How does the theoretic runtime complexity of the parsing methods compare to the actual measured runtime of the parsing code?*

The benchmarks showed a big difference between the runtime of the parsing plugins especially for large files. However, at least for our example data, all the plugins showed linear runtime behavior. Even the PEG parser library PEGTL that has a theoretical exponential runtime in the worst case showed this linear behavior. Interestingly, YAEP, the library that uses one of the most powerful parsing techniques tested (Earley parsing), showed the best runtime performance on macOS. On Linux the library was only slightly slower than the fastest parser, based on Bisons' LALR code.

❓ **RQ 2.** *How does the peak memory usage of the algorithms compare to each other? Do some of the algorithms show nonlinear memory usage?*

All of the parser plugins showed a linear peak memory usage increase for a linear increase of the size of the input. The difference between the peak memory usage was substantial, though. The YAEP parser needed the least amount of memory, while the Bison parser required about 20% more memory. The other parsers required about three, up to more than four times more peak heap memory.

❓ **RQ 3.** *How much work does it require to implement the plugins, i.e. how many lines of code do we have to write to support our YAML subset for each parsing engine? How do the amounts of handwritten code for the plugins compare to each other?*

The parser based on the YAML library yaml-cpp required the least amount of handwritten code. This was not surprising, since we can translate the high level output of the library relatively easily. If we only consider the plugins for which we wrote or generated the parsing code ourselves, then the ANTLR based plugin takes the lead, followed by the plugins based on Bison, YAEP and PEGTL.

❓ **RQ 4.** *Which parsing technique allows us to stay closest to the definition of the configuration language? Does staying close to the given definition allow us to extend and improve the parser and its support code more easily?*

We showed that the PEG library PEGTL allowed us to stay much closer to the representation of the specification of our example language YAML. This closeness provides utility, when we compare the ease of extensibility of the language grammar. However, in the case of YAML, the language specification is rather low-level. This means the extension of the support code that converts the data in Elektra's data structures takes considerable more effort than for the lexer based parsers.

❓ **RQ 5.** *What are the error handling capabilities of the parsing engines? How well can they handle multiple syntax errors? How do the generated error messages compare to each other?*

Only ANTLR offers multi-error message support (error recovery) without requiring any grammar changes. The Bison and YAEP parsers on the other hand need manual grammar updates to add error recovery points, which can cause conflicts (Bison) or ambiguous output (YAEP). Overall the error messages produced by the ANTLR, Bison and YAEP parsers are not great, but comparable to the ones produced by the handwritten recursive descent parser of yaml-cpp. Since yaml-cpp's parser does not support error recovery it is only able to report the first syntax error. The library with the most limited built-in error handling capabilities was PEGTL. The parser plugin based on this library only shows a single very limited error messages that might also not report the correct error location.

The answer to the research question above helped us to answer our main research question.

❓ **Main Research Question.** *Which parsing systems allows us to create a configuration parsing plugin that is easily extendable, has low maintainability cost and provides good error messages, while offering decent runtime performance and low memory overhead.*

In the end, ANTLR, the parsing engine

- that provides the most complete support code,

- that produces good error messages, and
- offers error recovery without any changes to the grammar

showed the best overall results according to our evaluation. Bison and YAEP also showed promising results, while yaml-cpp and PEGTL did not fit all requirements for an extended YAML storage plugin well. In the beginning of the thesis we also considered using the bidirectional programming library Augeas and the parser combinator library mpc. However, in the implementation phase we found that both of these libraries are unsatisfactory for our needs. Augeas is not able to process the context sensitive language YAML, and mpc does not seem to offer any significant advantage over the similar library PEGTL.

Overall this thesis contributes a thorough comparison of state of the art implementations of parsing techniques in the context of configuration data. While the current literature mostly compares different parsers that produce different output, we verify that our parsers produce the same data, providing a fair comparison of the given parser engines. Unlike other research we do not only compare the execution time and memory usage of the parser plugins, but also provide a detailed analysis of other important criteria, such as the error handling capabilities of the evaluated parsers.

5.2 Future Work

While we think that the comparison presented in this theses is thorough, we found some limitations future research could take into consideration.

5.2.1 Additional Data Formats

The YAML file format we used as example is quite complicated. We therefore used a custom lexer instead of a standard tool, like ANTLR's lexer or flex, to make the parsing process of the white space rules of the language easier. It would make sense to also write, generate, and compare parsing code for simpler data configuration formats, such as JSON, TOML or INI. These formats should make it easier to use a standard lexing tool, and allow us to determine how much influence a lexing tool has on the overall parsing process.

5.2.2 Type Support

We did not consider proper type support in the thesis. While we added support for binary data to the YAML CPP plugin, most of the other code we wrote does not support types properly. This can lead to problems, such as unwanted conversions from boolean data to integer values.

5.2.3 Lexer Level Error Messages

The custom lexer code written in this thesis does not detect or report errors at the token level. Adding support for this feature should be relatively easy and allow us to compare error messages for common low-level mistakes. This is especially interesting, since we would be able to assess, if the PEG library PEGTL is able to provide the same error message quality as handwritten custom code for low-level errors.

5.2.4 Additional Parser Engines/Generators

To improve the comparability of the runtime and memory benchmarks we only considered tools written in C or C++ in this thesis. However, some of the most interesting parsing research focuses on tools written in other programming languages.

LPegLabel (Lua) This library supports some of the recent interesting features for error handling in PEGs, such as labeled failures [Mai+16] and syntax error recovery [MM18].

Marpa (Perl) Marpa is a parsing library based on Earley's parsing algorithm. The library implements improvements to the algorithm from Leo [Leo91] and Aycock and Horspool [AH02]. According to the author [Keg19]: "Marpa is intended to replace, and to go well beyond, recursive descent and the yacc family of parsers".

Menhir (OCaml) Menhir is an LR parser generator that provides support for "example based error reporting" [Jef03; Käs+18; PR19]. In theory we should be able to generate parsers with Menhir that produce error messages comparable to the ones of handwritten recursive descent parsers, used in tools such as Clang or GCC [Käs+18, p. 2].

List of Figures

List of Tables

List of Listings

Acronyms

ABNF Augmented Backus-Naur Form. 43

ALL(*) Adaptive LL(*). 12, 17, 58, 67, 97, 98

ANTLR Another Tool for Language Recognition. 5, 6, 14, 17, 23, 46–50, 60, 74–78, 86–88, 96, 100, 102

API Application Programming Interface. 12, 79, 105

AST Abstract Syntax Tree. 49, 97

BNF Backus-Naur Form. 30, 46, 83

CC Cyclomatic Complexity. 14, 77, 78

EOF End Of File. 80

FLOSS Free/Libre and Open-Source Software. 15

GLR Generalized LR. 97

IELR Inadequacy Elimination LR. 97

JSON JavaScript Object Notation. 15, 21, 28, 36–39, 79, 103

KDB Key Database. 11

LALR Look-Ahead LR. 58, 97, 98

LL Left to right/Leftmost derivation. 16–19, 24, 46, 66, 78

LR Left to right/Rightmost derivation in reverse. 18, 23, 24, 67, 78, 97, 104

NLOC Noncommented Lines Of Code. 77

NS Namespace. 24, 25, 80, 105, 107

PEG Parsing Expression Grammar. 12, 19, 24, 50, 51, 83, 90, 97, 102, 104

PEGTL Parsing Expression Grammar Template Library. 4, 14, 50–52, 67, 78, 83, 86, 96, 101–104

SIMD Single Instruction/Multiple Data. 28

SLL(*) Strong LL(*). 67, 97

UUID Universally Unique Identifier. 61

XML eXtensible Markup Language. 21

Yacc Yet Another Compiler Compiler. 23, 29

YAEP Yet Another Earley Parser. 4, 14, 49, 58, 67, 74, 78, 86, 89, 100–103

YAML YAML Ain't Markup Language. 8, 9, 12–14, 21, 24, 27, 28, 30, 31, 34, 38–50, 52, 53, 55, 60–63, 68, 73–76, 78–81, 83–86, 88, 89, 91–96, 99–101, 103, 106–109

References

[ASU06] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.

[AH02] John Aycock and R Nigel Horspool. “Practical Earley Parsing”. In: *The Computer Journal* 45.6 (2002), pp. 620–630. URL: `http://staff.icar.cnr.it/ruffolo/progetti/projects/10.Parsing%20Earley/2002-Practical%20Earley%20Parsing-10.1.1.12.4254.pdf`.

[BS08] Ralph Becket and Zoltan Somogyi. “DCGs + Memoing = Packrat Parsing but Is It Worth It?” In: *Practical Aspects of Declarative Languages*. Ed. by Paul Hudak and David S. Warren. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 182–196. ISBN: 978-3-540-77442-6.

[Ber16] Felix Berlakovich. “A Universal Storage Plugin for Elektra”. B.S. Thesis. Technische Universität Wien, 2016. URL: `http://www.libelektra.org/ftp/elektra/berlakovich2016universal.pdf`.

[BPV06] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. “Relational Lenses: A Language for Updatable Views”. In: *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’06. Chicago, IL, USA: ACM, 2006, pp. 338–347. ISBN: 1-59593-318-2. DOI: `10.1145/1142351.1142399`. URL: `http://doi.acm.org/10.1145/1142351.1142399`.

[Boh+08] Aaron Bohannon et al. “Boomerang: Resourceful Lenses for String Data”. In: *ACM SIGPLAN Notices*. Vol. 43. 1. ACM. 2008, pp. 407–419. URL: `http://www.cis.upenn.edu/~bcpierce/papers/boomerang.pdf`.

[CP11] Xin Chen and David Pager. “Full LR(1) Parser Generator Hyacc And Study On The Performance of LR(1) Algorithms”. In: *Proceedings of The Fourth International C* Conference on Computer Science and Software Engineering*. ACM. 2011, pp. 83–92. URL: `http://hyacc.sourceforge.net/files/Xchen_c3s2e11.pdf`.

[Cho59] Noam Chomsky. “On Certain Formal Properties of Grammars”. In: *Information and control* 2.2 (1959), pp. 137–167.

[DM08] Joel E Denny and Brian A Malloy. “IELR(1): Practical LR(1) Parser Tables for Non-LR(1) Grammars with Conflict Resolution”. In: *Proceedings of the 2008 ACM symposium on Applied computing*. ACM. 2008, pp. 240–245. URL: https://people.cs.clemson.edu/~malloy/publications/papers/sac08/paper.pdf.

[DeR69] Franklin L DeRemer. “Practical Translators for LR (k) Languages”. PhD thesis. MIT Cambridge, Mass., 1969.

[Ear70] Jay Earley. “An Efficient Context-free Parsing Algorithm”. In: *Commun. ACM* 13.2 (Feb. 1970), pp. 94–102. ISSN: 0001-0782. DOI: 10.1145/362007.362035. URL: http://doi.acm.org/10.1145/362007.362035.

[Flo14] Daniel Flodin. “A Comparison Between Packrat Parsing and Conventional Shift-Reduce Parsing on Real-World Grammars and Inputs”. MA thesis. Uppsala University, Department of Information Technology, 2014. URL: http://uu.diva-portal.org/smash/record.jsf?pid=diva2%3A752340.

[For02] Bryan Ford. “Packrat Parsing: Simple, Powerful, Lazy, Linear Time”. In: *ACM SIGPLAN Notices*. Vol. 37. 9. ACM. 2002, pp. 36–47.

[For04] Bryan Ford. “Parsing Expression Grammars: A Recognition-Based Syntactic Foundation”. In: *ACM SIGPLAN Notices*. Vol. 39. 1. ACM. 2004, pp. 111–122.

[Fos+05] J Nathan Foster et al. “Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View Update Problem”. In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 233–246. URL: https://www.cs.cornell.edu/%7Ejnfoster/papers/lenses.pdf.

[Fro92] Richard A. Frost. “Constructing Programs as Executable Attribute Grammars”. In: *The Computer Journal* 35.4 (1992), pp. 376–389.

[FHC07] Richard A Frost, Rahmatullah Hafiz, and Paul C Callaghan. “Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars”. In: *Proceedings of the 10th International Conference on Parsing Technologies*. Association for Computational Linguistics. 2007, pp. 109–120.

[GJ08] Dick Grune and Ceriel JH Jacobs. “Parsing Techniques”. In: *Monographs in Computer Science. Springer, Second Edition. ISBN 978-1-4419-1901-4.* (2008).

[HU69] John E Hopcroft and Jeffrey D Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., 1969. URL: https://archive.org/details/HopcroftUllman_cinderellabook/page/n153.

[Hut92] Graham Hutton. “Higher-Order Functions for Parsing”. In: *Journal of Functional Programming* 2.03 (1992), pp. 323–343.

[HM96] Graham Hutton and Erik Meijer. “Monadic Parser Combinators”. In: (1996). URL: http://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf.

[Jef03] Clinton L Jeffery. “Generating LR Syntax Error Messages from Examples”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25.5 (2003), pp. 631–640.

[Jos06] Simon Josefsson. “The base16, base32, and base64 data encodings”. In: (2006). URL: https://tools.ietf.org/pdf/rfc4648.

[Käs+18] Daniel Kästner et al. “CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler”. In: *ERTS 2018: Embedded Real Time Software and Systems*. SEE, Jan. 2018. URL: http://xavierleroy.org/publi/erts2018_compcert.pdf.

[Knu65] Donald E Knuth. “On the Translation of Languages From Left to Right”. In: *Information and control* 8.6 (1965), pp. 607–639. URL: https://doi.org/10.1016/S0019-9958(65)90426-2.

[KZH16] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. “BiGUL: A Formally Verified Core Language for Putback-Based Bidirectional Programming”. In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM. 2016, pp. 61–72. URL: http://research.nii.ac.jp/~hu/pub/pepm16.pdf.

[Lan74] Bernard Lang. “Deterministic Techniques for Efficient Non-Deterministic Parsers”. In: *Automata, Languages and Programming*. Springer, 1974, pp. 255–269. URL: https://www.researchgate.net/publication/220898271_Deterministic_Techniques_for_Efficient_Non-Deterministic_Parser.

[LL19] Geoff Langdale and Daniel Lemire. “Parsing Gigabytes of JSON per Second”. In: *CoRR* abs/1902.08318 (2019). arXiv: 1902.08318. URL: http://arxiv.org/abs/1902.08318.

[Leo91] Joop MIM Leo. “A General Context-Free Parsing Algorithm Running in Linear Time on Every LR (K) Grammar Without Using Lookahead”. In: *Theoretical computer science* 82.1 (1991), pp. 165–176.

[Lut08] David Lutterkort. “Augeas – A Configuration API”. In: *Linux Symposium, Ottawa, ON*. 2008, pp. 47–56. URL: https://ols.fedoraproject.org/OLS/Reprints-2008/lutterkort-reprint.pdf.

[Mai+16] André Murbach Maidl et al. “Error Reporting in Parsing Expression Grammars”. In: *Science of Computer Programming* 132 (2016). Selected and extended papers from SBLP 2013, pp. 129–140. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2016.08.004. URL: https://arxiv.org/abs/1405.6646v3.

[Mar+17] Stewart Martin-Haugh et al. “C++ software quality in the ATLAS experiment: tools and experience”. In: *Journal of Physics: Conference Series*. Vol. 898. Institute of Physics Publishing Ltd. 2017. URL: http://iopscience.iop.org/article/10.1088/1742-6596/898/7/072011.

[MM18] Sérgio Medeiros and Fabio Mascarenhas. “Syntax Error Recovery in Parsing Expression Grammars”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing - SAC '18* (2018). DOI: 10.1145/3167132.3167261. URL: https://arxiv.org/abs/1806.11150.

[Mos14] Aaron Moss. "Derivatives of Parsing Expression Grammars". In: *arXiv preprint arXiv:1405.4841* (2014). URL: https://arxiv.org/pdf/1405.4841.pdf.

[Nir18] Peter Nirschl. "Cryptographic Methods For Elektra". B.S. Thesis. Technische Universität Wien, 2018. URL: https://www.libelektra.org/ftp/elektra/publications/nirschl2018cryptographic.pdf.

[Pac+15] Francesca Pacini et al. "Performance Analysis of Data Serialization Formats in M2M Wireless Sensor Networks". In: *ewsn 2015* (2015), pp. 7–8.

[Par09] Terence Parr. *Language implementation patterns: create your own domain-specific and general programming languages*. Pragmatic Bookshelf, 2009.

[Par13b] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013. URL: http://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference.

[PQ95] Terence J. Parr and Russell W. Quong. "ANTLR: A Predicated-LL (k) Parser Generator". In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810. URL: https://www.antlr3.org/article/1055550346383/antlr.pdf.

[PF11] Terence Parr and Kathleen Fisher. "LL (*): The Foundation of the ANTLR Parser Generator". In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 425–436.

[PHF14] Terence Parr, Sam Harwell, and Kathleen Fisher. "Adaptive LL (*) Parsing: The Power of Dynamic Analysis". In: *ACM SIGPLAN Notices*. Vol. 49. 10. ACM. 2014, pp. 579–598. URL: https://www.antlr.org/papers/allstar-techreport.pdf.

[Pot16] François Pottier. "Reachability and Error Diagnosis in LR (1) Parsers". In: *Proceedings of the 25th International Conference on Compiler Construction*. ACM. 2016, pp. 88–98. URL: http://gallium.inria.fr/~fpottier/publis/fpottier-reachability-cc2016.pdf.

[Raa10] Markus Raab. "A Modular Approach to Configuration Storage". MA thesis. 2010. URL: http://www.libelektra.org/ftp/elektra/thesis.pdf.

[Raa16] Markus Raab. "Improving System Integration Using a Modular Configuration Specification Language". In: *Companion Proceedings of the 15th International Conference on Modularity*. ACM. 2016, pp. 152–157. URL: https://www.libelektra.org/ftp/elektra/publications/raab2016improving.pdf.

[Raa17] Markus Raab. "Context-aware configuration". PhD thesis. TU Vienna, Dec. 2017. URL: http://book.libelektra.org.

[RB17] Ashley Robinson and Christopher Bates. "APRT – Another Pattern Recognition Tool". In: *GSTF Journal on Computing* 5.2 (Jan. 2017). URL: http://shura.shu.ac.uk/14312/.

[RS69] Daniel J Rosenkrantz and Richard Edwin Stearns. "Properties of Deterministic Top Down Grammars". In: *Proceedings of the first annual ACM symposium on Theory of computing*. ACM. 1969, pp. 165–180. URL: http://dl.acm.org/citation.cfm?id=805431.

[Rüf16] Michael Rüfenacht. "Error Handling in PEG Parsers". MA thesis. University of Berne, Switzerland, 2016. URL: `http://scg.unibe.ch/archive/masters/Ruef16a.pdf`.

[SS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory: Volume II LR (k) and LL (k) Parsing*. Vol. 20. Springer Science & Business Media, 1990.

[SM12] Audie Sumaray and S. Kami Makki. "A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform". In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '12. Kuala Lumpur, Malaysia: ACM, 2012, 48:1–48:6. ISBN: 978-1-4503-1172-4. DOI: `10.1145/2184751.2184810`. URL: `http://doi.acm.org/10.1145/2184751.2184810`.

[Tom85] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1985. ISBN: 0898382025.

[UA77] Jeffrey D Ullman and Alfred V Aho. *Principles of Compiler Design*. Addison-Wesley, 1977.

[WDG16] Alessandro Warth, Patrick Dubroy, and Tony Garnock-Jones. "Modular Semantic Actions". In: *Proceedings of the 12th Symposium on Dynamic Languages*. DLS 2016. Amsterdam, Netherlands: ACM, 2016, pp. 108–119. ISBN: 978-1-4503-4445-6. DOI: `10.1145/2989225.2989231`. URL: `http://doi.acm.org/10.1145/2989225.2989231`.

[Xu+13] Tianyin Xu et al. "Do Not Blame Users for Misconfigurations". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 244–259. URL: `https://cseweb.ucsd.edu/~tixu/papers/sosp13.pdf`.

Online Resources

[Bal13] Marc Balmer. *Lua as a Configuration And Data Exchange Language*. Jan. 2013. URL: https://www.netbsd.org/~mbalmer/lua/lua_config.pdf.

[Bax17] Ira Baxter. *What is the time and space complexity of a shift-reduce parser?* 2017. URL: https://www.quora.com/What-is-the-time-and-space-complexity-of-a-shift-reduce-parser (visited on 04/21/2017).

[Bed13] Jesse Beder. *Obtain Type of Value Stored in YAML::Node for yaml-cpp*. 2013. URL: https://stackoverflow.com/questions/19994312/obtain-type-of-value-stored-in-yamlnode-for-yaml-cpp/19995193#19995193 (visited on 11/15/2013).

[BEN09] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain't Markup Language (YAML™) Version 1.2*. Oct. 2009. URL: https://yaml.org/spec/1.2/spec.html.

[Ben12] Eli Bendersky. *How Clang Handles the Type / Variable Name Ambiguity of C/C++*. June 2012. URL: http://eli.thegreenplace.net/2012/07/05/how-clang-handles-the-type-variable-name-ambiguity-of-cc.

[Cox10] Russ Cox. *Generating Good Syntax Errors*. Jan. 2010. URL: https://research.swtch.com/yyerror.

[DS19] Charles Donnelly and Richard Stallman. *Bison - The Yacc-compatible Parser Generator*. May 2019. URL: https://www.gnu.org/software/bison/manual.

[Hab13] Josh Haberman. *LL and LR Parsing Demystified*. June 2013. URL: http://blog.reverberate.org/2013/07/ll-and-lr-parsing-demystified.html (visited on 10/09/2016).

[Hir16] Colin Hirsch. *RFE: Memoization*. 2016. URL: https://github.com/taocpp/PEGTL/issues/35 (visited on 11/24/2016).

[Keg11] Jeffrey Kegler. *What is the Marpa algorithm?* Nov. 2011. URL: http://blogs.perl.org/users/jeffrey_kegler/2011/11/what-is-the-marpa-algorithm.html (visited on 11/10/2016).

[Keg19] Jeffrey Kegler. *The Marpa parser*. 2019. URL: https://jeffreykegler.github.io/Marpa-web-site/.

[Lut17] David Lutterkort. *Support all YAML multiline modes*. Nov. 2017. URL: https://github.com/hercules-team/augeas/pull/524#issuecomment-343698357.

[Mye08] Joseph Myers. *New C Parser*. Jan. 2008. URL: https://gcc.gnu.org/wiki/New_C_Parser (visited on 01/10/2008).

[Par13a] Terence Parr. *Recursive Descent Parsing and ANTLR*. Dec. 2013. URL: http://stackoverflow.com/questions/20708126/recursive-descent-parsing-and-antlr/20709551#20709551 (visited on 11/02/2016).

[Pik16] Robert C. Pike. *[AMA] We are the Go contributors: ask us anything!* Feb. 2016. URL: https://www.reddit.com/r/golang/comments/46bd5h/ama_we_are_the_go_contributors_ask_us_anything/d03zx6f (visited on 02/17/2016).

[PR19] François Pottier and Yann Régis-Gianas. *Error handling: the new way*. June 2019. URL: http://pauillac.inria.fr/~fpottier/menhir/manual.html#sec67 (visited on 06/26/2019).

[Pro16] The Go Project. *Go 1.6 Release Notes*. Feb. 2016. URL: https://golang.org/doc/go1.6.

[Sim18] Kirill Simonov. *LibYAML*. June 2018. URL: https://pyyaml.org/wiki/LibYAML.

[Val19] Valgrind Developers. *What Valgrind does with your program*. 2019. URL: http://valgrind.org/docs/manual/manual-core.html (visited on 06/13/2019).