Enabling unbounded lookahead in LL(k) parsers using adaptive LL(*)

Performance analysis of the impact of adaptive LL(*) in the Chevrotain parsing library

Department of Computer Science

Master Thesis
to obtain the degree
Master of Science

presented by
Mark Sujew - 10189

Supervising Professor: Prof. Dr. José Baltasar Trancón Widemann
Second Examiner: Prof. Dr. Jan Haase
Submitted on: 21.04.2022

Contents

List of Figures

List of Tables

List of Abbreviations

ATN	Augmented Transitional Network
DFA	Deterministic Finite Automaton
DSL	Domain Specific Language
EBNF	Extended Backus-Naur-Form
ECMA5	ECMAScript 5
GC	Garbage Collection
JS	JavaScript
PEG	Parsing Expression Grammars

Listings

1 Introduction

Parsing, the recognition and analysis of input strings based on formal grammars, is a ubiquitous concept of software engineering and computer science. The matter is particularly important to computer programming itself, as virtually all programming languages are parsed from textual representations. Therefore, researchers and software engineers have developed numerous systems dedicated to parsing languages defined by formal grammars [1, 2]. Although extensive research has been done on parsing theory since the early 70s, as evident by the lasting success of Yacc based parsers [3] and the parsing techniques introduced in that period [1], parsing can hardly be called a solved problem. Recent additions to well-known parsing techniques like LL [4, 5] and the introduction of novel techniques like packrat parsing and the associated parsing expressions grammars (PEG) [6, 7] have shown that there is still potential for improvement on both parsing performance and cardinality of supported grammars.

1.1 Topic and context

Another effect of the widespread adoption of the Yacc technology was demonstrating the feasibility of using parser generators to produce parsers for widely used systems. While handcrafted parsers often provide increased flexibility and better error handling, later parser generators started incorporating these improvements to allow convenient and fast parser construction [8]. The ANTLR parser generator [9, 10] is arguably the best-known example of this. Starting as a simple parser generator for predicated LL(k) grammars, ANTLR has undergone a large number of improvements to its underlying parsing algorithm and thus has contributed to research in this field [4, 8].

Particularly exciting about ANTLR are its advances in arbitrary, unbounded LL lookahead [4, 5]. Their aptly called *Adaptive LL(*)* – or *ALL(*)* – strategy combines the unbounded lookahead mechanism known from other parsing techniques such as GLR [11] and PEG [7] with the strengths of LL parsers such

as its error handling and ease of debugging [4]. Although ANTLR has seen significant success in the commercial software industry, its advances in parsing research remain unused by other parsing libraries. A brief online search reveals that the current iteration of the ANTLR project, ANTLR4, is the only LL parser with unbounded lookahead [12]. While not providing an exhaustive listing, it demonstrates the prevalence of LL(1) and LL(k) parsers.

1.2 Questions and objectives

When presenting their findings, the authors claim that it outperforms any other parser generator while being only about 20% slower than the hand-built JavaC parser [5]. Due to its lookahead memorisation, ANTLR4 even outperforms JavaC when reparsing the test corpus. This performance assessment might be correct for Java-based parsers, but since JS-based languages and runtimes have become more critical in the software industry [13], JS parser libraries and parser generators with JS targets have become increasingly relevant. In this space, the Chevrotain parsing library [14] – an LL(k) interpreting parser – asserts the spot for the fastest parser, outperforming both ANTLR4 and hand-built parsers by a factor of two in an informal benchmark using the JSON grammar. Chevrotain achieves its incredible performance by heavily optimizing for the V8 engine, which is used in Chromium and the Node.js runtime [15–17].

Several questions arise from this claim. Most decisions of the JSON language can be parsed using an unpredicated LL(1) parser. As a consequence, both Chevrotain and ANTLR4 use an optimized LL(1) lookahead instead of their usual LL(k) and *ALL(*)* strategies. Therefore, this benchmark does not allow for comparison between the two lookahead algorithms. This begs the question whether Chevrotain's parsing algorithm is faster than ANTLR4's generated parsers on more complex grammars. Additionally, can it be further enhanced by using the *ALL(*)* strategy even if it is faster? Answering both of these questions is the objective of this thesis.

1.3 Overview of the structure

In chapter 2 we first present the basics of formal languages before moving on to explain the various techniques for parsing them. Additionally, an in-depth look at Chevrotain's architecture will be performed, highlighting its unique features compared to other parser libraries and parser generators. Furthermore, we discuss the current state-of-the-art parsing approaches in chapter 3.

Chapter 4 contains a description of the *ALL(*)* strategy implementation for Chevrotain. In chapter 5, we analyze how this new version of Chevrotain performs compared to other parsing libraries. We then discuss these benchmark results in chapter 6. Finally, we summarise the findings and propose open issues for future work.

2 Background

In this chapter, we give an overview of formal languages and outline well-known parsing techniques and their different lookahead algorithms. Furthermore, we explain how the Chevrotain parser library works in detail and how it differentiates itself from other parser libraries and parser generators.

2.1 Formal Languages

In the narrower sense of computer science, a formal language is a mathematical structure, defined on top of an *alphabet*, by the rules of a formal grammar [2, 18]. They are primarily concerned with the syntax of a sentence and allow to determine whether a specific string belongs to it or not [18]. The formal grammar allows to "generate" the formal language by using rules and elements of our alphabet to describe it [2].

As an example, let's take the alphabet $\Sigma = \{\varepsilon, a, b, c\}$. Using it, we can construct words such as a, abc and ε, the *empty* word. Now, languages using this alphabet might not contain every word built by this alphabet, but only a subset:

$$L_1 = \{ab, ac, abc\}$$
$$L_2 = \{a, aa, aaa, ..., a^n\}$$
$$L_3 = \{\varepsilon\}$$

2.2 Formal Grammars

Formal grammars enable us to generate languages more generically than simply writing them down, especially if the language can grow infinitely large, as is the case for L_2. For parsing, we will limit ourselves to the class of context-free grammars [2]. A context-free grammar is composed of a set of nonterminals N, a set of terminals T, a set of rules P, and a start symbol S which is a nonterminal. The union $N \cup T$ describes the set of *productions*. We write rules

in the form of $A \rightarrow \alpha$, where A is a nonterminal and α is a composition of an arbitrary amount of nonterminals and terminals. Rules with the same name can be grouped as $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_p$, where each α_k is called an *alternative* of A. [2, 19]

In the following, we will use the Extended Backus-Naur-Form (EBNF) to describe grammars. The EBNF extends *alternatives* to support *optionals* and *repetitions*. These are purely practical considerations, as both of these grammar features can be represented using empty alternatives or recursive rule invocations. A grammar rule such as $P \rightarrow \alpha \mid \alpha P$ can be replaced with $P \rightarrow \alpha+$ using the EBNF notation.

2.3 Parsing Techniques

Language construction using grammars is only the first step towards parsing. In practice, parsing does not only determine whether a string belongs to a language but also which productions constructed the string given a grammar [2]. Identifying productions is necessary since our languages contain embedded semantics. To understand the correct semantic meaning behind a sentence, we need the exact productions used to generate the string. The encountered productions form the *parse tree* [2, 20]. Reconstructing the parse tree is the main subject of parsing [2].

In the following, we explain the basics of commonly used parsing techniques, emphasising their respective lookahead mechanisms. The lookahead is a means to decide which alternatives the parser should choose by peeking into its unconsumed input. We start with bottom-up parsers and move on to top-down parsers.

2.3.1 Bottom-Up Parsers

Formally, bottom-up parsers derive the parse tree by searching for a series of leftmost reductions. They correspond to the rightmost productions that produced the input text [2]. Figure 2.1 presents a parse tree for a mathematical expression language. A bottom-up parser would first find the production matching the leftmost bottom element of the parse tree. The parser will then

move incrementally upwards and rightwards [21]. Figure 2.2 illustrates this behaviour. It portraits the same parse tree as figure 2.1 but replaces the labels of the input text and production names with the order in which the parser consumes them.

The search for productions can be realised using depth-first or breadth-first search algorithms. Both operate on the entire tree and are exponential in size. A depth-first search operates using backtracking [2]. While the breadth-first search is conceptually more straighforward, it has far larger memory requirements [2]. Unoptimised, both methods show exponential run time behaviour, rendering them unfit for real-world parsing tasks [2].

2.3.2 Earley

The Earley parser [22] is a breadth-first, bottom-up parser. Although described as a breadth-first top-down parser with bottom-up recognition by its author [22], its ability to handle direct left recursion aligns it closer to the class of bottom-up parsers. It improves the fan-out behaviour of naive breadth-first search by determining which reductions are incompatible with top-down pars-

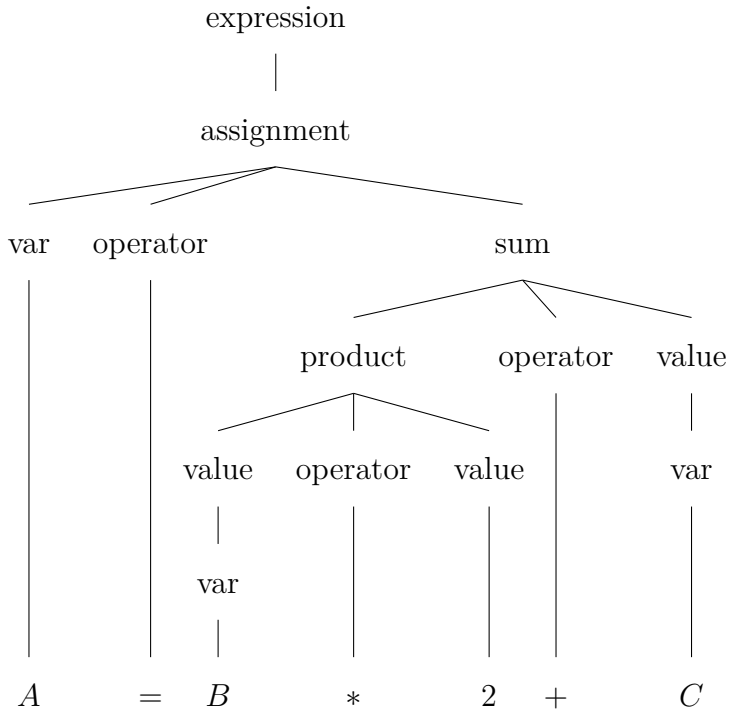

Figure 2.1: A parse tree for a mathematical expression language

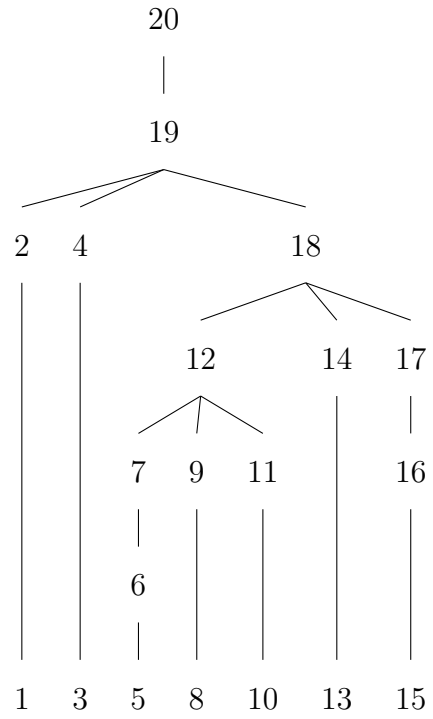

Figure 2.2: Order of bottom-up parsing steps

ing [2]. This reduces the algorithm to $O(n^3)$ from $O(C^n)$ for some constant C.

Adding a lookahead mechanism further improves the performance of the Earley parser. Such a lookahead computes the *FIRST* set of all nonterminals and their alternatives. The *FIRST* set of a production is the set of all terminals the production can start with [2]. While including a lookahead in an Earley parser makes it more efficient, as shown by Bouckaert, Pirotte and Snelling [23], other Earley parsing algorithms such as by McLean and Horspool [24] can improve parsing performance considerably without including a lookahead. The authors achieve this by building a hybrid between Earley and LR parsers [24].

2.3.3 LR

An LR parser (short for **L**eft-to-right, **R**ightmost derivation) works similar to an Earley parser. However, it uses a different approach to reduce the fan-out behaviour of the breadth-first search: By constructing a finite state automaton that starts with the start rule of the grammar and only considers right-hand

sides that could be derived from the start symbol [2, 25, 26]. The automaton is then converted into an *ACTION* and *GOTO* table, allowing the parser to quickly identify which reduction to choose. While this reduces the strength of the parser, it also improves the algorithm to $O(n)$ [2]. For example, a grammar that could never be deterministically parsed by an LR parser might look like this:

$$S \rightarrow aSa \mid a$$

The reason for this is simple. For every parsed aSa production, the LR parser would have to start its reduction from the middle of the input. However, no terminal indicates the middle of the input, rendering the LR parser unable to reduce the input to its intended productions [2]. This issue is not limited to LR parsers but also affects LL and PEG parsers presented in sections 2.3.5 and 2.3.6, respectively [7].

There are multiple methods to construct the automaton, with the most basic one being LR(0). Additional, more powerful methods take lookahead into account, like LR(1), LALR(1) or LR(k). These use a lookahead of 1 or k tokens, respectively. In practice, we rarely encounter LR(0) languages, while LR(1) grammars are ubiquitous [2]. We can prove that every LR(k) grammar with $k > 1$ can be transformed into an LR(1) grammar, but not LR(0) [27].

As the lookahead of LR grammars increases, the automaton and the associated tables become increasingly large [28]. For $k = 1$ in particular, better methods of deriving the tables have been discovered [2, 3], at the cost of its parsing power. The LALR(1) (short for Look-Ahead LR) method achieves this by first constructing all LR(0) automaton states and only deriving new states when necessary for parsing by adding a lookahead of 1, eliminating unnecessary automaton states (as constructed by LR(k)) in the process. While efficient LR(k) parser construction algorithms have been presented [29], $k = 1$ is usually enough for most real-world grammars [2].

2.3.4 Top-Down Parsers

As indicated by their name, top-down parsers process the input text by first predicting the top-most production, descending recursively until they have consumed all input. Figure 2.3 illustrates this behavior by replacing the labels from figure 2.1 to indicate the parsing step order. Top-down parsing can be accomplished using depth-first and breadth-first search algorithms, suffering from the same performance issues as naive bottom-up parsing [2, 26]. However, compared to bottom-up parsers, their prediction step for finding the top-most production does not have to rely on search and can be realized using other methods, drastically improving performance [2].

2.3.5 LL

The LL (**L**eft-to-right, **L**eftmost derivation) method replaces the search mentioned above with a simple table lookup. As previously with the Earley parser, we can use the *FIRST* set to provide the parser with lookahead information for each production [2]. The parser consults the lookup table each time it has to perform a prediction on the subsequent derivation – meaning when it en-

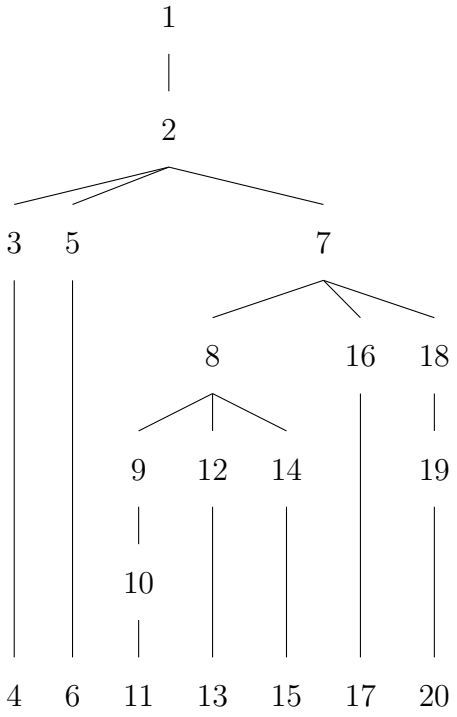

Figure 2.3: Order of top-down parsing steps

counters a production in the parsing process. The prediction algorithm peeks into the token stack, matches the tokens with entries in the lookup table and selects a single production to continue parsing [2]. In terms of performance, LL parsers take $O(n^4)$ units of time in the worst case but operate closer to $O(n)$ in practice, with n being the input length [2, 5].

Table 2.1 presents the lookup table for the grammar G_1. It shows which of the 3 productions to choose from depending on the start production and the received lookahead token. Since all predictions within G_1 can be achieved using a lookahead of a single token, it is an LL(1) grammar. Note that there is no prediction for production B and terminal a, indicating that such a configuration is illegal in the language produced by G_1.

$$
\begin{aligned}
G_1 : A &\to B \\
A &\to aA \\
B &\to b
\end{aligned}
$$

	a	b
A	2	1
B	—	3

Table 2.1: Lookup table for the simple LL(1) grammar G_1

Contrary to LR, a usable LL parser requires at least a lookahead of 1, as an LL(0) language would only consist of a single word [2]. However, the lookup table approach is easily extendable to construct an LL(k) language. Although such a lookup table could, in theory, experience exponential growth for $k > 1$, in practice, this growth is much closer to being linear [30]. The lookup table for LL(k) is built by extending the *FIRST* set to not only include a single terminal of each production but the first k terminals. This produces the $FIRST_k$ set, of which the already known *FIRST* set is just a special case for $k = 1$.

As with LR grammars, it is proven that any LL(k) grammar can be transformed into an LL(1) grammar [31]. However, doing so is not recommended,

as constructing an LL(k) parser is much less computationally expensive compared to an LR(k) parser.

LL(k) parsers cannot parse grammars that require an unbounded lookahead, such as G_2. The $FIRST_k$ set of both A_1 and A_2 will contain one path of terminals that looks like a^k. Consequently, the parser cannot decide which alternative to choose since, from the perspective of the LL(k) parser, G_2 is ambiguous [20].

$$
\begin{aligned}
G_2 : A_1 &\rightarrow Bb \\
A_2 &\rightarrow Bc \\
B &\rightarrow a*
\end{aligned}
$$

Such a grammar needs an unbounded LL parser. We will discuss approaches to LL parsing with unbounded lookahead in chapter 3. Additionally, regardless of the amount of lookahead, LL parsers are generally unable to handle left-recursion [2, 5].

2.3.6 PEG

Context-free grammars describe a *generative* approach to languages. For practical parsing use, context-free grammars are interpreted as being *recognition-based* [7]. Previous approaches to parsing such as LL, LR and Earley use context-free grammars. Ford introduces the Parsing Expression Grammars (PEG) system to describe languages that are meant to be parsed [7]. For this use case, an extension to the EBNF grammar syntax is presented, which aims to improve on ambiguous behaviour of the standard EBNF notation and introduces additional parsing features for practical use.

A context-free grammar rule such as $P \rightarrow a \mid ab$ poses an interesting problem to deterministic lookahead parsers. While it is clear which language the unordered alternative generates, it is left up for interpretation of the specific parser implementation for how to process this parsing "instruction". For example, one possible approach would always be to parse the longest alternative, meaning the matching alternative that consumes the most tokens. However,

this approach will fail to parse specific languages correctly, such as the one generated by G_3. The word ab belongs to the language $L_5 = \{ab, abb\}$, but when matching the longest alternative, the parser would reject this input since the token stack will be empty by the time it arrives at the second b terminal.

$$G_3 : A \to Bb$$
$$B \to a \mid ab$$

PEGs tackle this issue by dismissing the notion of *unordered* alternatives and introducing the "/" (*SLASH*) operator instead which indicates the order of alternatives [7]. This feature alone does not eliminate the issue as $a \ / \ ab$ would only ever match a and not ab, while $ab \ / \ a$ exhibits the same issue. For this, PEG additionally allows specifying syntactic prefix predicates within the grammar such as the & (*AND*) and ! (*NOT*) predicates, which indicate only to match the alternative if the lookahead for the prefixed production succeeds [7]. Note that such predicates also exist in common LL parser libraries and generators, although they are usually expressed in a Turing complete language instead of being a part of the grammar [5]. Using the & predicate we can modify B to $B \to ab\&b \ / \ a$, enabling successful parsing of L_5.

In addition to laying the groundwork for a different kind of grammar presentation with PEGs, Ford shows that it is possible to build linear time parsers for them, thereby introducing the *packrat* parser [6]. The packrat parser works similarly to backtracking parsers [1]. Using a memoisation table that stores previous parsing attempts for productions, the parser can improve the super-linear time of backtracking parsers. Furthermore, this approach allows for arbitrary, unbounded lookahead and the ability to parse left-recursion, making it stronger than both LL(k) and LR(k) parsers.

2.4 Chevrotain

Chevrotain is a parsing library developed by SAP SE for JS [14]. Since its inception in 2015, it has seen a steady rise in popularity, culminating in more than 8 million downloads from the package registry *npm* in 2021 and usage in academia [32–34]. In recent years, additional features and improvements have

made the library more approachable to developers, for example, extending its lookahead algorithm from LL(1) to LL(k). Nevertheless, its main selling point has always been the same: Chevrotain provides an embedded Domain Specific Language (DSL) to define its grammar while outperforming other parser libraries by several times. By embedding its grammar definition in code, it does not need to depend on generating additional code or parsing its grammar from an external source. Instead, its approach allows easy debugging and code bundling without complicated configuration.

2.4.1 Grammar Recognition

Unlike parser generators [3, 8, 11] or parser combinators [35], which explicitly define a grammar using a grammar definition file or a functional builder respectively, Chevrotain allows to define grammars using an object-oriented paradigm. Listing 1 displays a sample grammar for this. Using the `performSelfAnalysis()` method, the parser constructs an internal grammar representation. It accomplishes this by replacing the implementations of each parsing method (`CONSUME`, `SUBRULE`, `OPTION`, etc.) with a "recording" implementation. Afterwards, it invokes each parser rule exactly once without parsing any actual input, thereby transforming the parser rules into an abstract grammar representation with the help of the recording methods. Each time the embedded parsing DSL calls a recording method, the grammar recorder adds a corresponding production to the abstract grammar representation of the currently recorded rule. The parser shown in listing 1 results in the EBNF grammar seen in G_4:

$$
\begin{aligned}
G_4 : A &\rightarrow a\ (bc) * B? \\
B &\rightarrow d \mid e
\end{aligned}
$$

Additional computation is performed during the self-analysis phase to enable the parsing process. The most important aspect of this is the construction of the LL(k) lookup tables. Afterwards, an optional validation checks for any issues based on the recognized grammar, which would prevent successful parsing, such as left-recursion or ambiguous alternatives. This validation should be

disabled for productive usage since it introduces considerable computational overhead and provides no value after the initial development phase. Note that these computations would usually be performed during compile time of a parser generator tool and not run time of the parser itself, which is what is happening in the case of Chevrotain.

Furthermore, this grammar recognition approach lends to the simple addition of semantic actions, as the parsing instructions can be interspersed with other code without negatively impacting the parsing process. Semantic actions are wrapped in `ACTION` blocks which are skipped during grammar recognition.

2.4.2 LL(k) Lookahead

The LL(k) lookahead procedure in Chevrotain is split into two distinct phases. The first phase concerns itself with the construction of the lookup table during the self-analysis of the parser. Compared to the completely theoretical approach described in section 2.3.5, Chevrotain deviates slightly in its implementation: Instead of computing a single lookup table for the whole grammar, each alternation – which also includes optional elements and repetitions – receive a dedicated lookup table, consisting of a three-dimensional array of terminals. We use the term *alternation* here as an ordered list of alternatives. Building the lookup table resembles the design of the $FIRST_k$ set. Starting with a loop from $i = 1$ up to k, the table builder computes a $FIRST_i$ set for each alternative. Before increasing i, the parser performs a check for the uniqueness of each set. An additional byproduct of the $FIRST_k$ computation is the $FOLLOW_k$ set which describes the productions that follow the $FIRST_k$ set. If every $FIRST_i$ set identifies their respective alternative uniquely or every $FOLLOW_i$ set is empty, no further computation is necessary. This optimisation is performed based on the knowledge that not every prediction in an LL(k) grammar needs a lookahead of k. The lookup table for alternations can be sliced into two-dimensional arrays using the index of each alternative. This two-dimensional array is the practical representation of the $FIRST_k$ set, where every nested array is a possible token sequence that predicts the alternative. Finally, a validation is performed on the resulting lookup table, confirming that no ambiguous alternatives exist. The ambiguity detection for

LL(k) is quite simple: Chevrotain iterates over all alternatives within an alternation and determines whether any of its token sequences appear in another alternative. This indicates a lookahead ambiguity.

The second phase is executed during parsing each time the parser encounters an alternation. After retrieving the lookup table associated with the alternative, it performs the prediction and returns the index of the matched alternative or an indicator for showing that none of the alternatives could be matched (`undefined`). Listing 2 shows the algorithm performed on the lookup table for $k > 1$. The `LA(i)` function shown in the code returns the next unconsumed token at position `i`, thereby peeking into the token stack. As with PEGs, the algorithm always predicts the first matching alternative, ignoring every possible matching subsequent alternative. Chevrotain performs another optimization for the $k = 1$ case, where the three-dimensional lookup table is transformed into a simple dictionary, which maps each expected token to a predicted alternative. Listing 3 displays this optimization tactic.

The implementations shown in the listings were simplified and stripped of features such as semantic predicates. In addition, the prediction for alternatives is performed slightly differently compared to other productions such as optionals or repetitions, which only need to decide between two distinct alternatives, i.e. parsing the option/repetition or continuing with the other productions of the grammar rule. Due to this additional limitation, Chevrotain employs further optimizations for the lookahead of these productions.

3 Related Works

We have already discussed some relevant work regarding the theory of parsing systems and lookahead mechanisms in chapter 2. In this chapter, we will continue with different approaches to arbitrary lookahead parsing and focus on the work done by Parr, Harwell and Fisher [4, 5] on the *ALL(*)* parsing method.

3.1 Arbitrary Lookahead

The ideas behind arbitrary LL lookahead are rooted in the 1970s. Jarabek, Krawczyk [36] and Nijholt [37] introduced the notion of LL-regular grammars as an extension of LL(k) grammars. The LL-regular parsers presented by Nijholt [38] and Poplawski [39] are capable of linear two-pass parsing of the input. In the first pass, the parser reads the input right-to-left and enriches each token with a *right-context*. Afterwards, it performs a left-to-right pass in a similar fashion to LL(1) parsing. It can then correctly identify which production to predict based on the right-context. Research on LL-regular parsers has been mostly theoretical, and it is generally recommended to build an LR parser instead [2].

Another approach to arbitrary – but not unbounded – LL lookahead is presented by Belcak [40] as LL(*finite*). It aims to create LL(k) parsers without specifying k. More specifically, it calculates the exact k each alternative needs without calculating lookahead information beyond that point. Instead of building a lookup table for all productions, LL(*finite*) constructs a lookahead Deterministic Finite Automaton (DFA). The general structure of this DFA is similar to DFA for parsing type 3 languages with two key differences. First off, it does not allow cycles in the DFA since that would exceed the class of LL(k) grammars. Secondly, its accepting states indicate whether the unconsumed input is part of the lookahead and which alternative this state predicts. When arriving at an alternative, the associated DFA is *simulated*, effectively parsing the next k tokens as if they were part of an acyclic type 3 language.

Additionally, Belcak proposes modifications to the algorithms for optimization and enabling semantic predicates.

In the LR world, LR-regular grammars were designed by Cohen and Culik [41] as unbounded lookahead extensions to LR(k) grammars [25]. Like LL-regular parsers, they use a two-pass approach that computes right-context during the first pass to perform the correct reductions in the second pass. Bermudez and Schimpf [29] define LAR(m) grammars, which successively try to parse a production with LR(0) and LR(m) before falling back onto an arbitrary lookahead approach. Similar to Belcak [40], an ideal m is chosen where possible. However, the developer of the parser needs to "guess" the amount of left-context needed for the LR(0) parser [29].

Later, Tomita [42] developed generalized LR (GLR) parsers. By forking subparsers from the current state, the parser can explore all possible paths and choose the best fitting subparser for the given input. A graph-structured stack is employed to prevent parsing the same input twice in the same way. Tomita shows GLR parsers to be 5 to 10 times faster than corresponding Earley parsers. [42] Based on GLR, Scott and Johnstone [43] presented generalized LL (GLL) parsers, bringing the subparser and graph-structured stack approach to LL parsing. GLL is $O(n^3)$ and GLR is $O(n^{p+1})$ where p is the length of the longest grammar production.

3.2 ANTLR3 LL(*)

Before Parr et al. developed *ALL(*)* for ANTLR4, ANTLR3 already employed a less capable arbitrary LL lookahead mechanism simply called LL(*) [44]. LL(*) lookahead utilizes DFAs during the static grammar analysis phase to generate switch-on-token prediction code. Contrary to Belcak's LL(*finite*), these DFAs are not limited to pure LL(k) decision, but allow cycles. Therefore, they are able to express unbounded lookahead scenarios that do not contain rule recursion. Trying to statically express rule recursion inside DFAs is not possible using ANTLR3's approach [44].

3.3 Adaptive LL(*)

Building on top of the research performed on LL-regular [36–39], GLL [43], PEGs [6, 7] and previous work on ANTLR3 [44], Parr, Harwell and Fisher [4, 5] present the *ALL(*)* parsing method. It heavily incorporates ideas from packrat parsing [6] by relying on dynamic analysis of the grammar and memorisation tactics to improve performance.

Before going deeper into the dynamic analysis part performed during the parser's run time, we first establish how the necessary static analysis operates. Using a predicated context-free grammar, *ALL(*)* constructs an ATN. ATNs resemble programming languages syntax documentation but provide additional information about predicates and make heavy use of ε-transitions. Figure 3.1 shows an ATN for a single nonterminal. For each nonterminal, the algorithm generates an ATN subgraph, starting with a *production start* state and iterating over all productions from the left-most edge of the nonterminal, including semantic predicates. All productions contribute at least one state and transition to the subgraph. The last state in the subgraph is the *production stop* state, doubling as the accepting state for each subgraph. While terminals and nonterminals in the grammar create simple transitions, other productions, in this context called *decision points*, lead to complex structures heavily utilizing ε-transitions.

Dynamic analysis of the ATN starts when encountering a decision point during the parsing process. It will begin the lookahead process by using the ATN state associated with the encountered decision point to create a lookahead DFA. At first, this DFA contains a single start state. Each DFA state consists of a set of ATN *configurations*, which describes a tuple made up of the

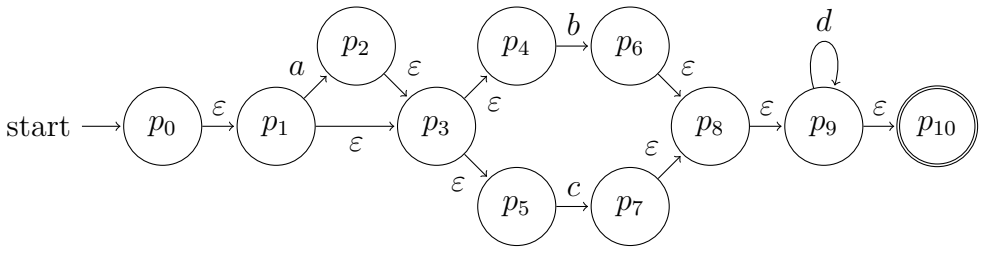

Figure 3.1: ATN for the production $P \rightarrow a?\ (b \mid c)\ d*$

ATN state p, the predicted alternative i, the ATN call stack γ and an optional predicate π. Except for its start state, each DFA state is produced by applying a modified *subset construction* algorithm on the ATN subgraph. This ATN simulation reads one token at a time from the stack and uses the subset construction to move through all ε-transitions. Simulation continues until it reaches a terminal matching the token, a nonterminal or a predicate. Predicates act as ε-transitions and are only evaluated at the end of the simulation. When encountering a nonterminal, the simulation uses the corresponding ATN subgraph to continue with the subset construction. The set of all reached ATN configurations is part of the next DFA state. After each new state, the simulation adds an edge from the previous state to the new state labelled with the read token. The simulation ends once all ATN configurations in the current DFA state point to the same alternative i and marks this state as being one *accepting* state of the DFA.

If the dynamic analysis reencounters the decision point further into the input, the algorithm can use the existing token labelled edges to move to any DFA state without recomputing it. Tokens that do not have a transition yet will restart the subset construction, enabling lazy memorisation similar to packrat parsing.

Contrary to the simulated lookahead DFAs used by LL(*finite*), *ALL(*)* is able to deal with cycles by utilizing the equality of DFA states. In particular, DFA states are equal if their set of ATN configurations are equal. When consuming a repetition, every iteration will result in equal ATN configurations, therefore creating a cycle in the DFA. These cycles greatly improve performance in unbounded lookahead scenarios.

Although semantic predicates should be used to remove grammar ambiguities, there might be cases where the dynamic analysis has to resolve the ambiguity during runtime. A practical example of this would be the ECMAScript 5 (ECMA5) grammar, which contains ambiguities by design [45]. For one, if all ATN configurations point towards *production stop* states and the ATN call stack is empty, ATN simulation cannot continue. Additionally, an *overflow* constant limits the total amount of lookahead. This limit guarantees that the prediction will not continue running until the end of the token stack. Further-

more, a heuristic is used to determine ambiguities even before reaching the production stop states, which is explained in more detail by Parr, Harwell and Fisher [5]. Once an ambiguity is detected, the prediction gracefully terminates in favour of the alternative with the lowest index.

ALL(*) has been successfully implemented in ANTLR4 [5]. Although the algorithm performs at $O(n^4)$ in theory, their experiments show that it operates linearly in practice. In particular, they show that the parse time increases linearly with the file size. Additionally, the authors were able to show that ANTLR4 performs better than every other Java or C++ parser generator for the Java grammar, often by multiple orders of magnitude.

4 Implementation

This chapter will concern itself with the implementation of the *ALL(*)* algorithm and the challenges that arise in the process of it. First, a naive approach to the problem will be implemented. It will take a shortcut to the solution by skipping the initial ATN construction, instead choosing to compute the prediction DFA using the production rules directly. This will allow for a first validation of the dynamic grammar analysis. Afterwards, we perform a full implementation of *ALL(*)* method.

4.1 ATN-less Implementation

In the *ALL(*)* method, an ATN is built during the analysis phase, which then derives the lookahead DFA during parse time. The ATNs main task is to provide a data source for the simulation to compute the next state of the lookahead DFA. However, this information can also be obtained by recursively calling the $FIRST_k/FOLLOW_k$ computation of Chevrotain introduced in section 2.4. Using the $FOLLOW$ set allows the next state to repeat this computation until either a single unique alternative has been found or no alternative matches the input. Since each encountered token produces a new $FOLLOW_k$ set, we always use $k = 1$ in this computation. Although the implementation of this approach is straightforward and yields positive results when testing for correctness on simple grammars, it has several drawbacks making it unfit for real-world grammars:

- Ambiguous alternatives are never resolved. If two or more alternatives with the same prefix are identified in LL(k), the algorithm will always match the first appearing alternative, behaving predictably and deterministically. In the ATN-less implementation, the automaton will continue iterating over the possible ambiguous alternatives until it finds a token that does not match any of the tracked alternatives since there is no configuration in which only a single unique alternative is actually

valid. As no alternative has been found that matches the token stack, a parsing error will occur.

- When encountering repetitions during the lookahead phase, a new DFA state will be constructed for each token instead of creating a cycle. These additional states decrease performance due to the $FIRST_k/FOLLOW_k$ computation performed for each encountered token.
- The approach reaches only sub-par performance. Even on simple grammars, using the ATN-less algorithm yields up to 50% decreased parsing performance compared to LL(k). Benchmarking was performed using Chevrotain's built-in performance benchmarking tool running tests for the JSON, CSS and ECMAScript 5 parser. This benchmark does not only include the lookahead computation but the actual parsing step as well, i.e. consuming input tokens and execution of semantic actions. While a certain degree of decreased performance can be attributed to benchmarking unoptimized code inside of a heavily optimized codebase, the magnitude of the regression implies that the ATN-less approach does not yield the same performance benefits presented by Parr et al. in their full implementation of the *ALL(*)* method.

We perform the full implementation next in the hope of improving both correctness of the lookahead and performance. It will later be evaluated in chapter 5.

4.2 *ALL(*)* in Chevrotain

The implementation of the complete *ALL(*)* method in Chevrotain aligns closely to the original algorithm presented by Parr and Fisher [4]. Additionally, to facilitate a more straightforward implementation of the method, the structure resembles the implementation of ANTLR 4.9. Nevertheless, due to architectural differences in Chevrotain and ANTLR4 and in an attempt to reduce the amount of breaking changes within the framework, some changes to the *ALL(*)* method have been performed.

4.2.1 Implementation differences

The first difference is the execution of semantic predicates. Similar to ANTLR4, Chevrotain does not differentiate between syntactic and semantic predicates. However, they differ drastically in their implementations. Whereas ANTLR4's syntactic predicates are expressed as parts of the grammar similar to PEGs and are transformed into semantic predicates during the static analysis phase, Chevrotain uses a Turing complete approach to syntactic predicates. Listing 4.1 shows an alternation in Chevrotain with syntactic predicates utilizing the `GATE` property and `LA(i)` method. This strategy, however, does not lend itself to embedding in the ATN. The reason for this can be explained using the grammar G_5. π_1 and π_2 are syntactic predicates that act as gates during ATN simulation. In Chevrotain, no tokens are consumed from the stack during lookahead. Hence, when the simulation starts with A and continues to B, all `LA(i)` method calls will result in an off-by-one error.

$$G_5 : A \rightarrow aB \mid aB$$
$$B \rightarrow \pi_1 b \mid \pi_2 b$$

```
1 P = this.RULE("P", () => {
2     this.OR([
3         {
4             GATE: () => this.LA(2).tokenType === b
5             ALT: () => this.CONSUME(a)
6         }
7     ])
8 })
```

Listing 4.1: An example of a syntactic predicates in Chevrotain

We resolve the predicate issue in the same manner as in the current version of Chevrotain. Instead of considering every predicate that could be encountered during ATN simulation, only the predicates of the alternation which started the prediction are taken into account. While implementing the same behaviour as proposed by Parr et al. [5] is possible, it would result in a breaking change in the behaviour of Chevrotain and will therefore not be performed.

Additionally, this change enables another optimization for predicated alternations. ATNLR's LL(*) algorithm performs predicate evaluation at the end of the lookahead. Consequently, predicates that would have resolved ambiguities already at the start are evaluated at the end. While this behaviour guarantees the stability of the constructed DFA [5] regarding predicates, it also increases the amount of lookahead needed for predicated alternations. In our implementation, predicates are evaluated at the start of the lookahead, with each configuration of predicate results receiving a dedicated lookahead DFA instance. This change resolves the stability issue experienced when using only a single DFA and potentially reduces the amount of lookahead needed for predicated alternations at the cost of having to construct 2^n DFAs with n being the number of predicates within an alternation.

Another difference is the addition of separated repetitions in Chevrotain. This feature simplifies repetitions by introducing a separating terminal. For example, a list of function arguments in a programming language usually has to be expressed using a rule such as $P \rightarrow (ID\ (',' ID)*)?$. Using separated repetitions, developers can skip the optional production and write this rule with a single repetition, as shown in listing 4.2. This feature has to be considered when predicting an alternative and, therefore, also in the ATN construction. Separators introduce new states and transitions, which must be added to the construction algorithm.

```
1 Args = this.RULE("Args", () => {
2     this.MANY({
3         SEP: Comma,
4         DEF: () => this.CONSUME(ID)
5     })
6 })
```

Listing 4.2: Argument list using a separated repetition

4.2.2 Ambiguity Detection

For LL(*) grammars, ambiguity detection cannot be performed by simply comparing token sequences across alternatives, since they could grow infinitely large. Instead, a more sophisticated grammar analysis is necessary. While

ANTLR4 postpones this ambiguity detection to the dynamic analysis phase [4], being able to provide meaningful error messages on ambiguous behaviour during static analysis is of great benefit to users of the parsing library.

In the context of Chevrotain, detecting ambiguities of an alternation is to determine whether at least two alternatives can predict the same sentence. In other words, the intersection of the language L_a (produced by one alternative) and L_b (produced by another alternative) is not empty. Confirming this is comparatively simple for type 3 languages, where we could construct an intersection automaton and determine whether it has any non-ε-transitions. However, for context-free grammars, the issue is more complex. It has been proven that whether a context-free grammar is (un)ambiguous is an undecidable problem [21].

Since this issue is a practical consideration for Chevrotain and is not part of any of the research objectives, this thesis does not concern itself any further with this question. Instead, the dynamic analysis is used to report on ambiguities during run time. The inclined reader will find more information on this topic in the works of Pandey [46] and Basten [47]. The authors present, test and rank state-of-the-art methods for ambiguity detection of context-free grammars.

5 Analysis

In section 1.2, we raised the questions of whether Chevrotain is actually faster than ANTLR4 for complex grammars and whether this performance can be enhanced even further by implementing the *ALL(*)* strategy in Chevrotain. In the following, we will explain how we ensure the correctness of our implementation and how we plan to answer our research questions. To differentiate better between the versions of Chevrotain, we will refer to them as Chevrotain(k) and Chevrotain(*) for the respective lookahead length.

5.1 Correctness

Chevrotain features multiple testing suites to ensure the functional correctness of the library. The most basic test suite performs unit tests by testing parsing functions in isolation. Additionally, a large set of integration test cases ensures that these functionalities correctly work in conjunction. In these tests, a custom parser is usually built to test specific features of the library, such as its lookahead or error recovery functionality. The Chevrotain project requires any contributions to maintain a code-coverage of 100%. Although achieving code-coverage is not a good predictor for correct functionality [48], it is easily enforceable and provides some degree of certainty in both functionality and edge case behaviour, assuming the tests are written sensibly. Furthermore, Chevrotain features example implementations of commonly used languages in another test suite, ranging from simple descriptive languages like CSV and JSON to more complex languages like GraphQL and ECMAScript 5. The ability to correctly parse these languages provides an additional level of confidence in the implementation.

Before adding any tests that explicitly test the behaviour of the ATN construction or simulation, the test suites contain 1213 test cases. Running the test suites resulted in 5 failures, leaving 1208 passing tests. All testing failures concern themselves with the unique lookahead behaviour of certain ECMAScript features, which is accomplished by overriding lookahead related parser meth-

ods. These are now deprecated, which results in unexpected lookahead behaviour. All other integration tests pass.

Using the existing tests, test coverage of 96% and 97% has been achieved for the ATN construction and simulation, respectively. This leaves 16 lines of untested code, indicating either a comparatively small amount of edge cases or high quality of the integration tests. In any case, after adding further tests for the new lookahead strategy, our confidence in the correctness of the implementation of the *ALL(*)* method is high. The tests are responsible for testing new behaviour, such as always predicting the longest alternative or correctly resolving ambiguities during dynamic analysis.

5.2 Benchmarking

We aim to answer both research questions by performing benchmarks on Chevrotain(k), Chevrotain(*) and ANTLR 4.9. The source of the first claim – to recall, being unsure whether Chevrotain performs faster in its benchmark due to the simplicity of the parsed language in question or employing superior optimization tactics – can be validated by performing another benchmark using more complex languages. For this, we will use the existing benchmarking infrastructure of Chevrotain. To facilitate easier development and maintenance, Chevrotain allows comparing the parsing performance of its last release with the current in-development version using a simple script. Since performance is a feature of Chevrotain, regressions are treated the same as functional regression. Therefore, this benchmark script can be seen as an additional test suite. It compares the performance of parsers implemented for the JSON, CSV and ECMA5 languages to the current baseline.

In particular, ECMA5 can be considered a complex language and exhaustive concerning the parsing features needed to to parse the language successfully. For example, it contains ambiguities, employs rule recursion and requires semantic predicates. Note that ECMA5 is an LL(2) language when respecting semantic predicates. In addition to the different versions of Chevrotain, we will include ANTLR4 for the upcoming benchmark. For this purpose, we will use the reference implementations for the respective grammars from the ANTLR4

`grammars-4` repository [49]. These grammars align with the official specifications of their respective languages. While this usually decreases performance, it increases the likelihood of correctness. The Chevrotain parser for any given language will also align to the specification to prevent grammar specific performance optimizations, which might impact the comparison negatively. An ANTLR4 based JS parser for each language will be generated and incorporated into the benchmarking infrastructure.

In addition, we also include parsers for the Java programming language in the benchmark. Contrary to ECMA5, it does not feature ambiguities but instead covers the *unbounded* part of the lookahead algorithm. In many parts of the grammar, such as lambda expressions, typecasting or package declarations, Java requires unbounded lookahead to align closely with the official specification. In Chevrotain(k), these productions are parsed using backtracking, which incurs a heavy performance penalty. Chevrotain(*) should be able to parse Java code without backtracking, thereby improving performance. Note that the specification uses left-recursion and therefore needs to be transformed into an LL conform grammar without left-recusion.

The Chevrotain benchmark performs multiple parses over the same input to gain statistically significant results. This type of benchmark automatically biases the results towards a parsing technique with memorisation since reparsing a document does not require computing new DFA states. Nevertheless, such a scenario is valuable for several real-world use cases, such as editors, which reparse input several times with minimal modification. Hence, we also perform a second, more exhaustive benchmark. We will parse a larger corpus of files once, which aligns more with how command-line tools operate.

Furthermore, we include the Acorn parser [50] for the ECMA5 benchmark. It serves as a "gold-standard", being a handwritten parser for JS using JS itself. Usually, we would consider using parser implementation from a JS runtime. However, due to it being an interpreted language, these parsers are compiled to machine code, rendering comparison with them misleading.

All benchmarks will be conducted on an AMD Ryzen 7 3700X 8-Core CPU with 2133 MHz RAM running Windows 10. The benchmark excutes the full

parser flow, including lexical analysis of the input. The input files are stored in memory before the benchmark begins to minimise the impact of disk-read-related bottlenecks. The benchmarking framework used is “Benchmark.js” [51].

5.3 Results

Table 5.1 shows the performance of different parsing libraries for selected languages. The results are measured in operations per second, meaning that a higher numerical value of a parsing library compared to other results of the same row indicates higher parsing performance for that specific language. The benchmarking framework repeated parsing a hundred times to gain statistically significant results, making the results prone to memorisation bias.

In general, Chevrotain(*) shows slightly performance across LL(2) languages compared to Chevrotain(k). Using the LL(*) algorithm has decreased parsing performance by 1%, 2% and 5% on JSON, CSS and ECMA5 respectively. Additionally, it is 24% slower than our set “gold-standard” for ECMA5, Acorn. Both Chevrotain versions outperform ANTLR4 by multiple factors on JSON and by multiple orders of magnitude on CSS and ECMA5. For Java, Chevrotain(*) outperforms Chevrotain(k) and ANTLR4 by a factor of 10 and 16, respectively.

While the results for the JSON language are consistent with the online benchmark provided by Chevrotain, ANTLR4 performs unexpectedly slowly for the CSS and ECMA5 language. When looking at the reference CSS grammar for ANTLR4, it can be quickly discerned that it does not follow the specification closely. It seems to contain additional productions to deal with malformed input and therefore spends more time during its lookahead prediction. Although this might not explain the full extent of the performance regression, it indicates that the Chevrotain and ANTLR4 results are not comparable due to differences in their concrete grammar. The differences in performance for the ECMA5 grammar are not as noticeable and will be explained later in this section.

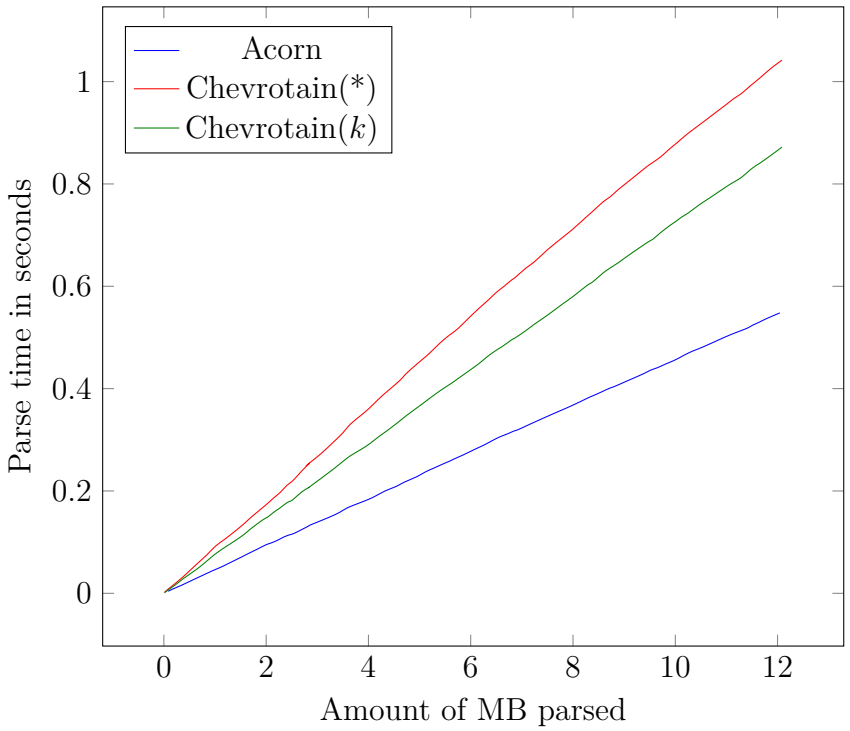

Figure 5.1: Parse time in seconds in relation to the parsed amount of text. Conducted on 13MB of ECMA5 code from the Theia-IDE framework

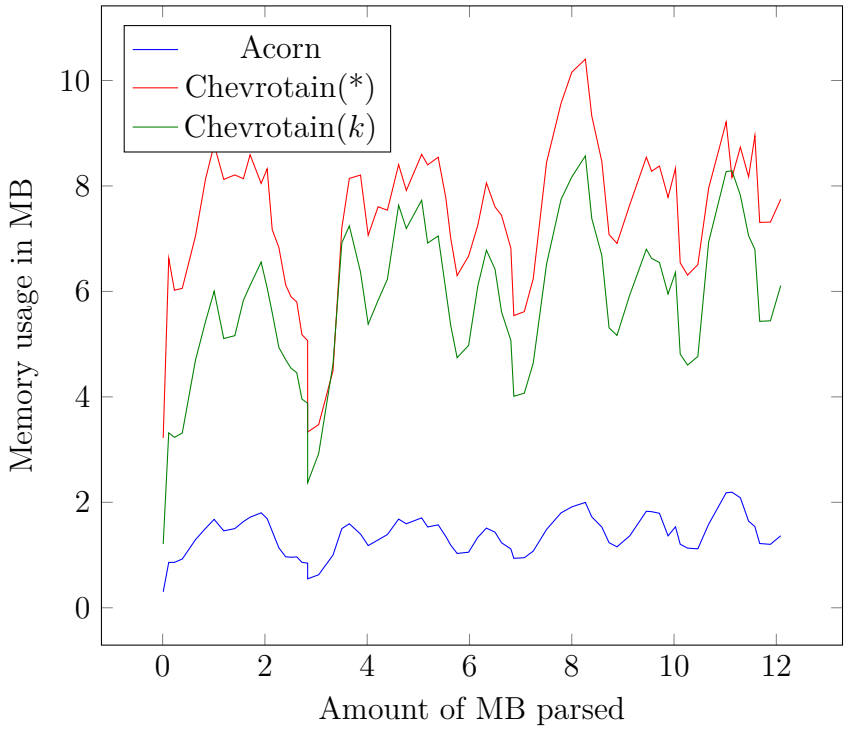

Figure 5.2: Memory consumption in MB in relation to the parsed amount of text. Conducted on 13MB of ECMA5 code from the Theia-IDE framework

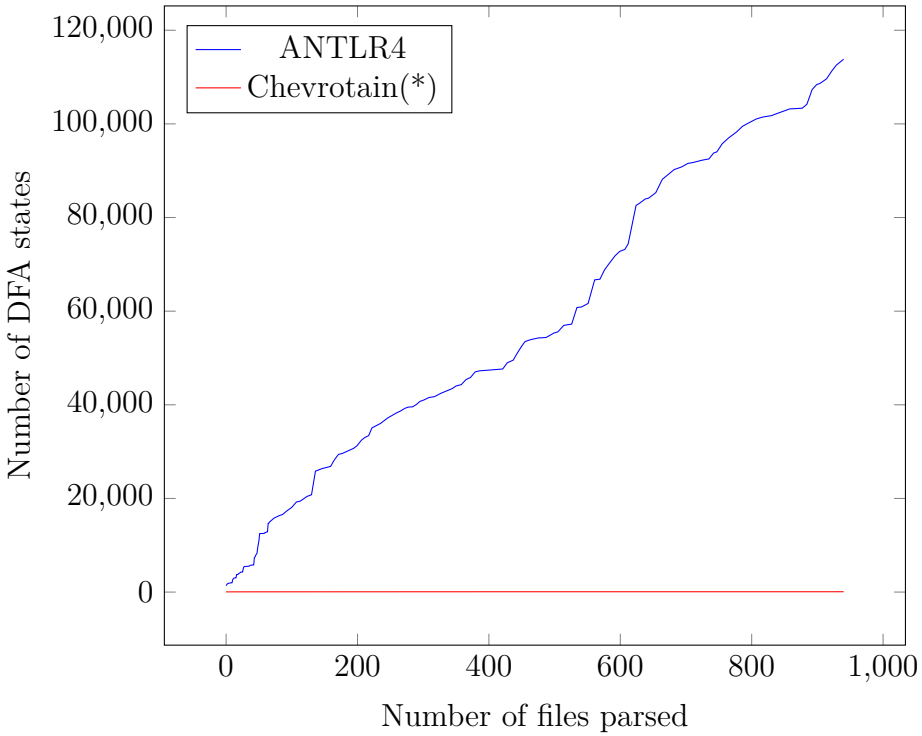

Figure 5.3: Number of DFA states in relation to the amount of files parsed. Conducted on 13MB of ECMA5 code from the Theia-IDE framework

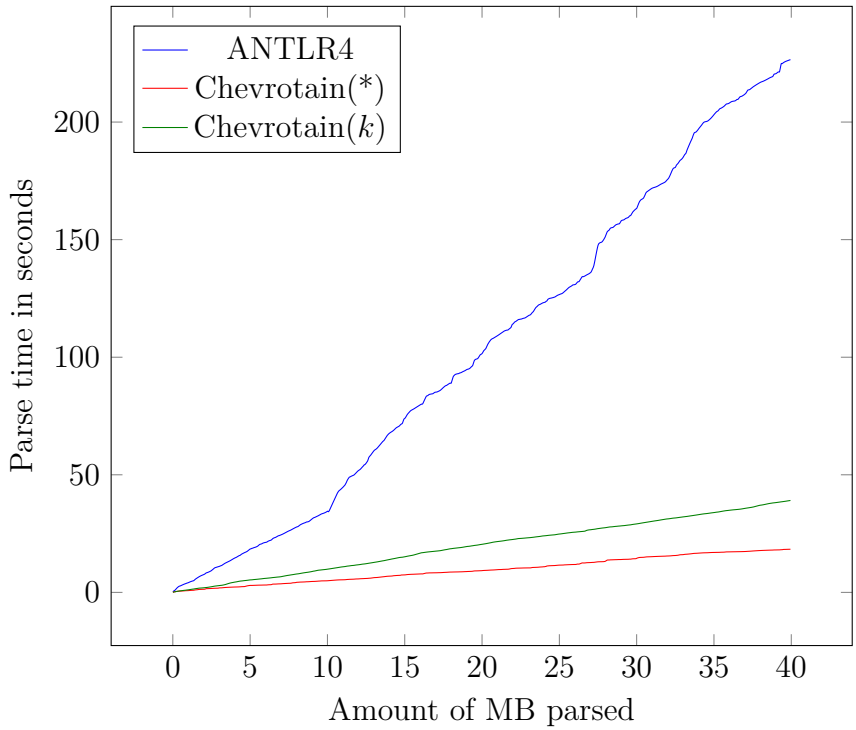

Figure 5.4: Parse time in seconds in relation to the parsed amount of text. Conducted on 40MB of Java code from the Spring framework

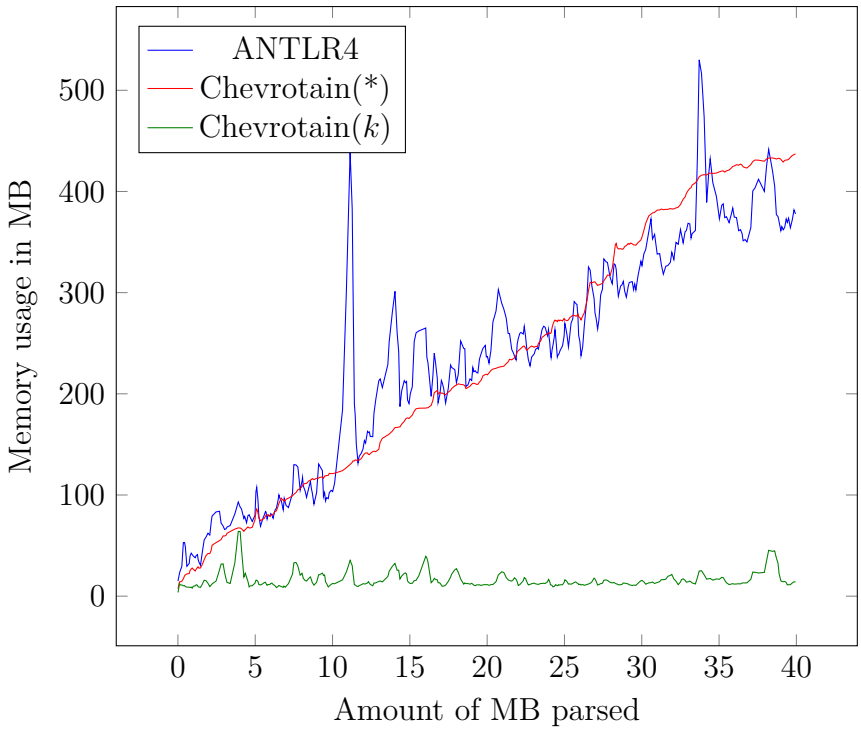

Figure 5.5: Memory consumption in MB in relation to the parsed amount of text. Conducted on 40MB of Java code from the Spring framework

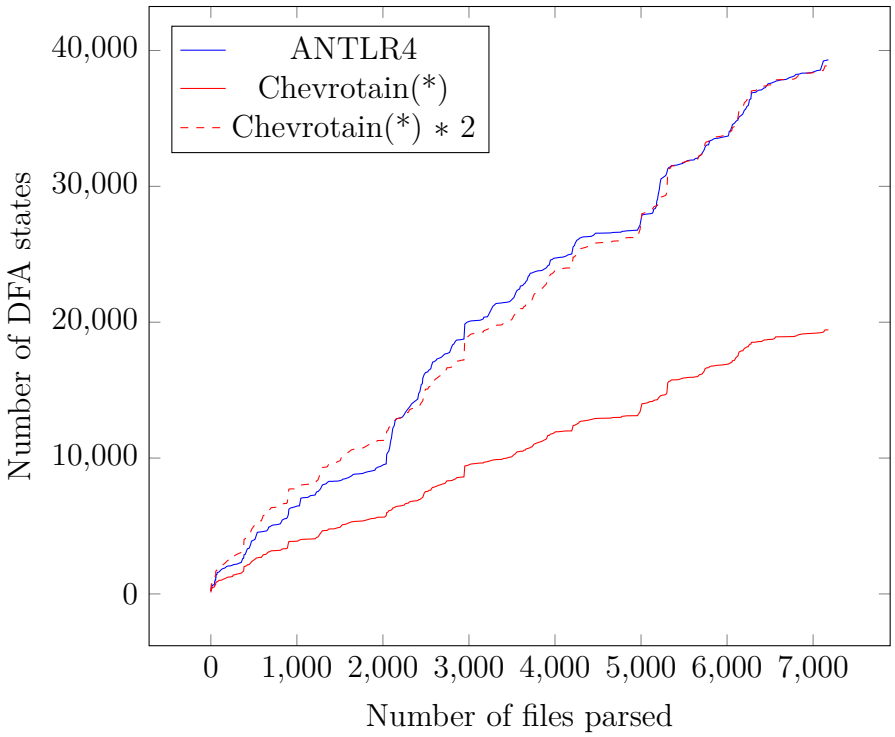

Figure 5.6: Number of DFA states in relation to the amount of files parsed. Conducted on 40MB of Java code from the Spring framework

Table 5.1: Single file parsing performance of different parser libraries measured in operations per second.

	Chevrotain(*)	Chevrotain(k)	ANTLR4	Acorn
JSON	9184	9527	3175	—
CSS	3092	3127	8.22	—
ECMA5	471	496	4.65	621
Java	20.05	2.03	1.22	—

Figures 5.1 and 5.2 display performance metrics of Chevrotain ECMA5 parsers and the Acorn library using a corpus of 13MB of code taken from the Theia-IDE framework. They show the parse time and memory consumption throughout the full parsing process. All parsers perform in linear time with results that are consistent with the data shown previously in table 5.1. Note that Chevrotain(*) performs linearly, although its memorisation approach makes it susceptible to non-linear behaviour due to the initial warm-up phase of the ATN simulation.

As shown in figure 5.2, memory consumption for Chevrotain(*) stays close to the original Chevrotain(k) parser. The increased memory consumption can be attributed to the creation of DFAs states which are not disposed of during garbage collection. Additionally, it can be observed that DFA computation leads to higher memory consumption at the start of the parsing process.

Furthermore, memory consumption is also the reason why ANTLR4 does not appear in figures 5.1 and 5.2. After parsing less than 10% of the test corpus, the benchmark crashes with an out-of-memory error. Figure 5.3 demonstrates this behaviour by showing the amount of DFA states constructed during parsing. After less than a thousand files, ANTLR4 has already constructed more than 110.000 DFA states using more than 4GB of memory, thereby crashing the Node.js runtime. Given that ECMA5 is a predicated LL(2) language, it seems unreasonable to create such a large amount of DFA states. Chevrotain(*) also indicates this, which only creates 72 DFA states for parsing the whole corpus. The source of this issue is quickly identified and is inherent to ANTLR4's employed ambiguity resolving strategy. ECMA5 contains an ambiguity for function declarations, which a syntactic predicate can easily resolve. Although

the ANTLR4 grammar used in this benchmark contains such a predicate, it is only evaluated at the end of the ambiguity. Therefore, ANTLR4 needs to parse the whole function declaration before deciding on the correct alternative. Since function declarations are often long and filled with different token types, the lookahead automaton for this alternation quickly increases in size, resulting in the out-of-memory error.

Contrary to ANTLR4, Chevrotain(*) evaluates predicates already at the start of the alternation, therefore excluding impossible alternatives from the DFA state computation completely. This approach automatically resolves the predicated ambiguity problem experienced by ANTLR4 based parsers and decreases demand for DFA computation drastically, as shown in figure 5.3. For the single file performance benchmark of table 5.1, ANTLR4 needs to process multiple hundred tokens to come to a lookahead prediction. Even with memorisation, this incurs a heavy performance penalty and explains the magnitude of performance difference compared to Chevrotain.

Moving on to Java parsers, we can see in figure 5.4 that both versions of Chevrotain perform in linear time, with Chevrotain(*) being twice as fast as Chevrotain(k). The Chevrotain(k) parser uses backtracking to accomplish unbounded lookahead required by parts of the Java grammar specification. Replacing backtracking in favour of actual unbounded lookahead has a highly positive impact on parsing performance. Nevertheless, the performance increase is not as significant as indicated by the data displayed in table 5.1. This behaviour can be explained by taking the data shown in figure 5.6 into account. Throughout the whole parsing process, the adaptive predict algorithm continues to construct 2.7 new DFA states per parsed document on average. Consequently, Chevrotain(*) spends some time on the subset construction algorithm for each encountered file. However, when reparsing the same document repeatedly, no new ATN configurations can be encountered, thereby skipping the subset construction and improving parsing performance.

ANTLR4 displays irregular behaviour when parsing the large test corpus compared to Chevrotain. Although all three parsers show linear parsing performance up to 10MB, ANTLR4 behaves non-linearly afterwards. The start of this behaviour coincides with a sharp increase in memory consumption seen in

figure 5.5. While Chevrotain's memory usage remains relatively stable across the test corpus, the ANTLR4 parser experiences large spikes. Note that during the memory benchmark, the GC is invoked manually in a regular interval. Consequently, large spikes indicate increased memory usage over short periods. The observed non-linear behaviour of ANTLR4's parsing performance could be related to the additional time it takes for the runtime to perform its GC. This would explain the linear behaviour seen for the first 10MB and the irregular behaviour after that. Although figure 5.6 shows an increase of DFA states after 10MB as well, it is no clear explanation for the memory spikes, considering that Chevrotain(*) does not experience this abnormal behaviour.

Instead, we argue that the algorithms and data structures ANTLR4 employs to perform the subset construction are not suitable for the V8 runtime engine. This is a consequence of ANTLR4's main target platform being the Java runtime and the fact that the implementation of the ANTLR4 runtime on all other target platforms is closely aligned to the Java implementation. A prominent example of this is the custom dictionary implementations used for ANTLR4 in JS. To replicate the exact behaviour of the Java implementation, ANTLR4 cannot use natively implemented dictionaries. Consequently, every algorithm built on top of these custom implementations operates on slower data structures that are prone to higher memory consumption. While this is a plausible explanation for the increased memory usage, more investigation is needed to confirm this suspicion.

Figure 5.6, which displays the number of constructed DFA states throughout the benchmark for ANTLR4 and Chevrotain(*), is also showing unexpected behaviour. Given that ANTLR4 and Chevrotain both parse the same grammar, it is surprising that ANTLR4 constructs approximately twice the amount of DFA states at any point of the benchmark compared to the Chevrotain implementation. The scaled plot of Chevrotain(*) closely resembles the plot of ANTLR4, intuitively suggesting either an error in the implementation of the LL(*) algorithm or a difference in the number of alternatives in the grammar. However, the actual reason for this behaviour is the same as for the data shown in figure 5.3, although not as obvious. The Chevrotain(*) Java grammar contains 20 syntactic predicates used to speed up the lookahead algorithm without

influencing the actual parsing result. Although the parser correctly parses any input without these predicates, they are used to skip the prediction of specific alternatives, thereby reducing the total amount of lookahead necessary for each prediction as well as the total number of DFA states.

Nevertheless, this does not allow elimination of all LL(*) predictions. Consequently, Chevrotain(*) still experiences a steady increase of DFA states, although not as large as ANTLR4's. Adding these predicates to ANTLR4 as well does not improve parsing performance or reduce the amount of DFA states. On the contrary, it increases parse time due to the additional evaluation of predicates at the end of its lookahead. Since ANTLR4 continues with the ATN simulation until it can predict a single alternative even in the presence of predicates, its lookahead algorithm constructs the same amount of DFA states as if predicates were absent. Chevrotain(*) constructing roughly half the amount of DFA states can be therefore attributed to chance rather than errors in the Chevrotain(*) implementation or the grammars employed for the benchmark.

5.4 Summary

Having analyzed the runtime behaviour of Chevrotain(*), Chevrotain(k) and ANTLR4 in-depth, we can summarize our results as follows:

1. Chevrotain(*) displays almost equivalent runtime performance compared to Chevrotain(k) on all LL(k) grammars. In general, it performs slightly worse, with up to 5% less operations per second on the single file benchmark.

2. It performs better than Chevrotain(k) by multiple factors on LL(*) grammars. Chevrotain(k) requires to employ backtracking to parse them effectively.

3. Both implementations outperform ANTLR4 by multiple factors or even magnitudes. This effect becomes more exaggerated the more complex the grammar becomes.

4. Chevrotain(*) consumes approximately the same memory as ANTLR4.

In conclusion, we were able to show that the *ALL(*)* algorithm does not improve parsing performance for LL(k) grammars in Chevrotain. Instead, it enables the parser to handle LL(*) grammars without requiring backtracking, albeit at the cost of a minor performance regression. Additionally, we can observe that the implementation differences to ANTLR4 result in significantly increased parsing performance for predicated ambiguous alternatives.

6 Discussion

In this chapter, we discuss our findings and the conduct of this thesis. Firstly, we identify differences and overlap in our results and the work of Parr et al. [4, 5]. Secondly, we present advantages and potential disadvantages arising from differences in the LL(*) implementation. Finally, we analyze any threats to the validity of our work.

6.1 Comparison with ANTLR LL(*)

Figure 6.1 displays the benchmark results for parsing 123MB of Java 6 code as presented by Parr et al. [5]. It shows a clear improvement of ANTLR4 LL(*) compared to traditional LL(k) parser generators. These results cannot be reproduced using a Java 17 parser sourced from the `grammars-v4` repository with the V8 JS runtime. The results are not even reproducible in the Java runtime environment. We assume this is due to two factors: The Java 6 grammar contains no generics or lambda expressions introduced in Java 7 and 8, respectively. This lack of new features makes the language much easier to parse compared to later iterations of Java. Furthermore, we assume that a non-specification compliant Java grammar has been employed for the original ANTLR4 benchmark. This is indicated by using LL(k) parsers in the benchmark, which cannot parse a specification compliant Java grammar due to its LL(*) decisions.

Although we were unable to show in our experiments that Chevrotain(*) parsers are faster than equivalent, backtracking-less Chevrotain(k) parsers, we can support the claim made by Parr and Fisher [4] that the *ALL(*)* method provides an efficient lookahead mechanism on a par with commonly used LL(k) algorithms. We can also observe other similarities, such as the ever-increasing amount of DFA states on the Java grammar or the improved parse time when reparsing a test corpus. Nevertheless, our experiments do not indicate a clear superiority of *ALL(*)* compared to optimized LL(k) lookahead.

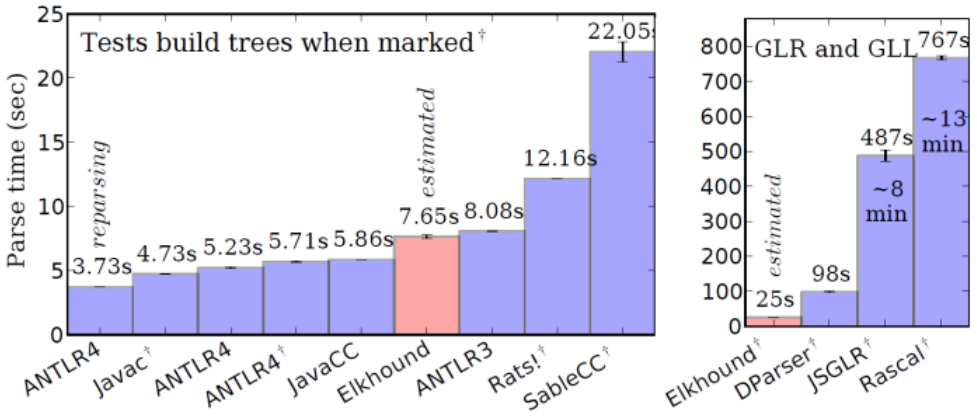

Figure 6.1: Comparing Java parse times on Java 6 Library and compiler source code [5]

6.2 Effect of Implementation Differences

The largest deviation in behaviour of Chevrotain(*) compared to ANTLR4 involves handling ambiguities in predicated grammars. To recall, our implementation only resolves the predicates of the starting alternation and creates a DFA for each unique predicate configuration. The driving force behind this decision was to enable faster ambiguity resolution for predicated alternations. Our experiments further enforce this decision by showing that this ambiguity resolving strategy successfully reduces the amount of DFA states needed to parse ambiguous grammars. This effect is shown heavily exaggerated in the ECMA5 grammar, where this strategy reduced a potentially expensive LL(*) decision into an LL(2) decision.

Nevertheless, there is an obvious disadvantage to this approach. Aside from the predicates of the alternation that started the prediction, no other predicates are taken into account during dynamic analysis. This could potentially lead to parsing errors of valid input on LL(k) languages. Take G_6 as an example, which contains a parameterized production B:

$$G_6 : A \rightarrow B\langle false\rangle \mid B\langle true\rangle$$
$$B\langle\pi\rangle \rightarrow \{\pi\}\ a \mid \{!\pi\}\ b$$

When receiving the terminal input a, our *ALL(*)* implementation will always choose the first alternative of A, as it is the alternative with the lowest index. However, when performing the lookahead afterwards inside of B, it will determine that a is not a valid option anymore since its associated predicate returns *false*. Consequently, although the language produced by G_6 contains a, the parser cannot parse the word and reaches an erroneous state. While ANTLR4 deals with this edge case correctly, such a grammar configuration is rare. Additionally, it can still be parsed correctly by using backtracking. Therefore, we believe that our early exit strategy is better suited for Chevrotain than the predicate resolving strategy employed by ANTLR4.

6.3 Threats to Validity

Empirical evidence is prone to many different kinds of biases, measurement errors, and other issues that might threaten the validity of our work. We now discuss the precautions we have taken to prevent these issues from invalidating our results.

When benchmarking code in interpreted languages, the cold-startup behaviour of the runtime has to be taken into account. Figure 6.2 shows the effect of benchmarking the parsers without an initial warm-up phase. Instead of the expected linear behaviour exhibited in figure 5.1, we observe seemingly logarithmic runtime behaviour. However, the additional time needed here can be attributed to the runtime engine optimizing code only after it has seen it multiple hundred times. In the meantime, the runtime executes non-optimized code, which incurs a performance penalty. The amount of yet-to-be optimized code decreases as the benchmark progresses, ultimately leading to linear behaviour. The data for figures 5.1 and 5.4 has been collected after each parser has seen the whole test corpus once. Note that the DFAs for Chevrotain(*) and ANTLR4 have been reset to provide accurate measurements of the subset-construction algorithm.

Additionally, measurements for parse time and memory consumption have been conducted independently from each other. To gain an accurate presentation of memory usage, we tracked the memory for each parsed file and

also regularly invoked the GC. We have done this to gain more comprehensible data over long-lived objects such as DFA states. However, it also incurs a performance penalty unrelated to the actual parsing performance. Therefore, the benchmarks shown in figures 5.1 and 5.4 have been conducted without memory measurements. Instead, figure 6.3 presents the impact of accurate memory measurements on parse time. It shows that parse time increases by a factor of 20 when also measuring memory usage.

It could be argued that manually invoking the GC falsifies the results. However, we could only barely verify the general upwards trend of LL(*) parsers in memory consumption without it. Figure 6.4 presents the memory consumption of the Java parser benchmark without manually invoked GC. While a faint trend for Chevrotain(*) and ANTLR4 is visible in the data, most of it is simply noise produced by short-lived objects which are deleted from memory during GC as indicated by the zig-zag pattern. Since we already discussed the performance impact of a large number of short-lived objects in section 5.3, we are confident that analyzing the data presented in figures 5.2 and 5.5 is less prone to misinterpretation compared to data generated by GC-less benchmarks.

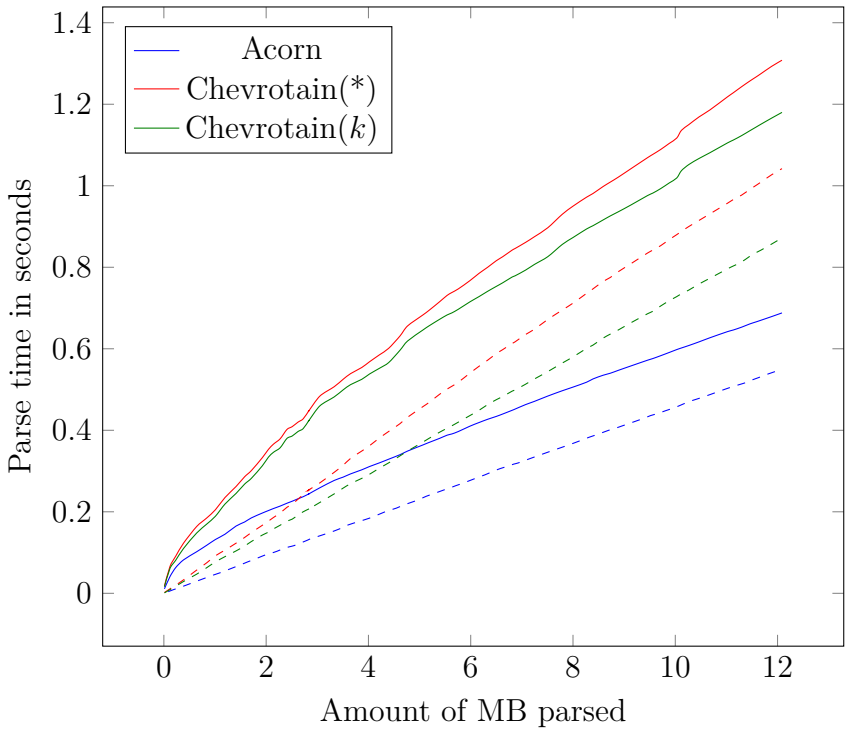

Figure 6.2: Parse time of the JS test corpus when performing the benchmark without an initial warm-up phase. The dashed plots represent the data from figure 5.1.

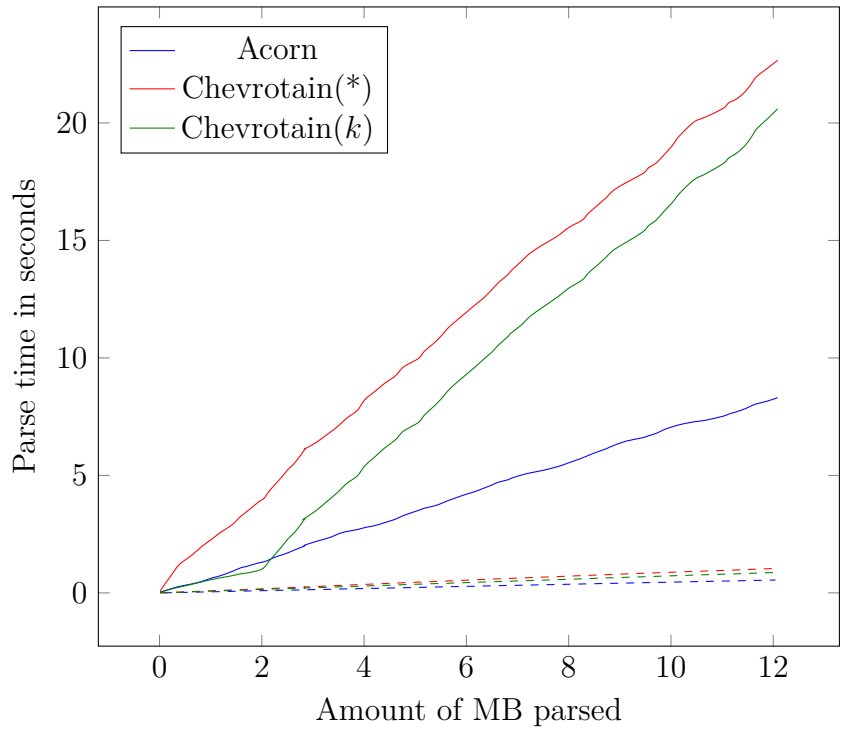

Figure 6.3: Parse time of the JS test corpus while also measuring memory consumption and regularly invoking GC. The dashed plots represent the data from figure 5.1.

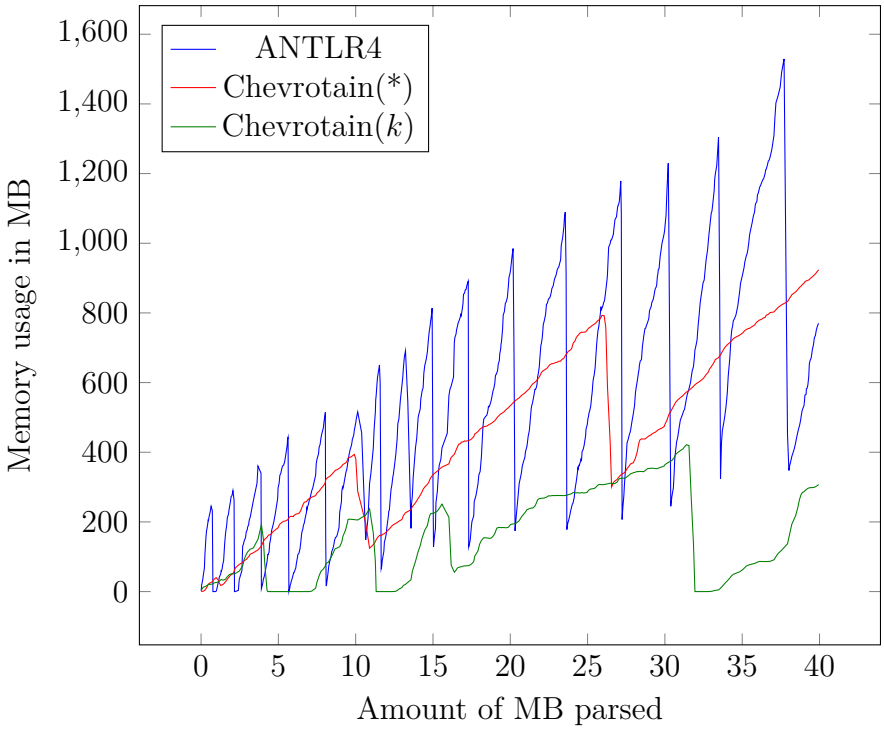

Figure 6.4: Memory consumption for Java parsers without manually invoking the GC.

7 Conclusion

The area of unbounded lookahead for LL parsers is mainly unexplored and complex. Only little existing research covers this topic, with the most promising being the *ALL(*)* method presented by Parr, Fisher and Harwell [4, 5] for the parser generator ANTLR4. However, informal benchmarks using the V8 JS engine have shown that optimized LL(k) parsers outperform ANTLR4 based parsers by multiple factors. This thesis aims to provide an unbiased comparison of LL(k) and LL(*) lookahead methods by implementing the *ALL(*)* method in the LL(k) parser *Chevrotain*.

We presented an implementation of the *ALL(*)* method for Chevrotain and were able to show that employing this lookahead strategy slightly decreases parsing performance for common use cases compared to an optimized LL(k) lookahead algorithm. However, it vastly expands the range of parseable grammars. Additionally, we were able to improve significantly on the implementation of the *ALL(*)* method compared to ANTLR4 for predicated ambiguity detection. Furthermore, our experiments have confirmed that Chevrotain parsers are faster and more memory efficient than ANTLR4 parsers for equivalent grammars.

Consequently, we still see room for improvement for *ALL(*)*. Since our employed predicate evaluation strategy has its drawbacks, consolidating it with the advantages of ANTLR4's approach could be one potential field of future work. In addition to this, other macro and micro-optimization tactics can be explored to increase parsing performance even further. As presented in section 4.2.2, static ambiguity detection had to be dropped from Chevrotain due to its complexity when operating in an unbounded lookahead scenario. Further research in this area is needed to provide accurate validations.

As for the practical application of this work, both the authors of this thesis and the maintaining contributors of Chevrotain agree to upstream the change. Chevrotain will employ the *ALL(*)* method as its definitive lookahead strategy.

Bibliography

[1] A. V. Aho; J. D. Ullman: *The theory of parsing, translation, and compiling*. USA: Prentice-Hall, Inc., 1972, 2051 pp.

[2] D. Grune; C. J. H. Jacobs: *Parsing techniques: a practical guide* (Monographs in computer science), 2nd ed. New York: Springer, 2008, 662 pp.

[3] S. C. Johnson: *Yacc: Yet another compiler-compiler*. Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[4] T. Parr; K. Fisher, 'LL(*): The foundation of the ANTLR parser generator,' *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 425–436, Jun. 4, 2011.

[5] T. Parr; S. Harwell; K. Fisher, 'Adaptive LL(*) parsing: The power of dynamic analysis,' *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 579–598, Oct. 15, 2014.

[6] B. Ford, 'Packrat parsing: Simple, powerful, lazy, linear time, functional pearl,' *ACM SIGPLAN Notices*, vol. 37, no. 9, pp. 36–47, Sep. 17, 2002.

[7] B. Ford: 'Parsing expression grammars: A recognition-based syntactic foundation,' in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '04, New York, NY, USA: Association for Computing Machinery, Jan. 1, 2004, pp. 111–122.

[8] T. J. Parr; R. W. Quong, 'ANTLR: A predicated-LL(k) parser generator,' *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[9] T. J. Parr; R. W. Quong: 'Adding semantic and syntactic predicates to LL(k): Pred-LL(k),' in *Compiler Construction*, P. A. Fritzson, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1994, pp. 263–277.

[10] T. Parr: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Jan. 15, 2013, 432 pp.

[11] S. McPeak; G. C. Necula: 'Elkhound: A fast, practical GLR parser generator,' in *Compiler Construction*, E. Duesterwald, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2004, pp. 73–88.

[12] *Comparison of parser generators*, in *Wikipedia*, Page Version: 1058539493, Dec. 4, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Comparison_of_parser_generators&oldid=1058539493 (visited on 12/11/2021).

[13] K. Kumar; S. Dahiya, 'Programming languages: A survey,' *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 5, no. 5, p. 8, 2019.

[14] SAP SE. 'Chevrotain.' (Dec. 11, 2021), [Online]. Available: https://chevrotain.io (visited on 12/11/2021).

[15] G. Southern; J. Renau: 'Overhead of deoptimization checks in the v8 javascript engine,' in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10.

[16] G. Krylov; M. Patrou; G. W. Dueck; J. Siu: 'The evolution of garbage collection in v8: Google's JavaScript engine,' in *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, Jun. 2020, pp. 1–6.

[17] R. Zhuykov; E. Sharygin, 'Ahead of time optimization for JavaScript programs,' *Proceedings of the Institute for System Programming of the RAS*, vol. 27, no. 6, pp. 67–86, 2015.

[18] S. C. Reghizzi; L. Breveglieri; A. Morzenti: *Formal Languages and Compilation*, 3rd ed. 2019 Edition. New York, NY: Springer, 2019, 512 pp.

[19] A. Afroozeh; A. Izmaylova: 'Faster, practical GLL parsing,' in *Compiler Construction*, B. Franke, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2015, pp. 89–108.

[20] S. Sippu; E. Soisalon-Soininen: *Parsing Theory: Volume I Languages and Parsing*, Softcover reprint of the original 1st ed. 1988 edition. Berlin: Springer, Aug. 23, 2014, 236 pp.

[21] A. V. Aho; R. Sethi; J. D. Ullman, 'Compilers, principles, techniques,' *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[22] J. Earley, 'An efficient context-free parsing algorithm,' *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, Feb. 1, 1970.

[23] M. Bouckaert; A. Pirotte; M. Snelling, 'Efficient parsing algorithms for general context-free parsers,' *Information Sciences*, vol. 8, no. 1, pp. 1–26, 1975.

[24] P. McLean; R. N. Horspool: 'A faster earley parser,' in *Compiler Construction*, T. Gyimóthy, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1996, pp. 281–293.

[25] D. E. Knuth, 'On the translation of languages from left to right,' *Information and Control*, vol. 8, no. 6, pp. 607–639, Dec. 1, 1965.

[26] S. Sippu; E. Soisalon-Soininen: *Parsing Theory: Volume II LR(k) and LL(k) Parsing*. Springer Science & Business Media, Apr. 17, 2013, 432 pp.

[27] M. D. Mickunas; R. L. Lancaster; V. B. Schneider, 'Transforming LR(k) grammars to LR(1), SLR(1), and (1,1) bounded right-context grammars,' *Journal of the ACM*, vol. 23, no. 3, pp. 511–533, Jul. 1, 1976.

[28] E. Ukkonen, 'Upper bounds on the size of LR(k) parsers,' *Information Processing Letters*, vol. 20, no. 2, pp. 99–103, Feb. 15, 1985.

[29] M. E. Bermudez; K. M. Schimpf: 'A practical arbitrary look-ahead LR parsing technique,' in *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, ser. SIGPLAN '86, New York, NY, USA: Association for Computing Machinery, Jul. 1, 1986, pp. 136–144.

[30] E. Ukkonen: 'On size bounds for deterministic parsers,' in *Automata, Languages and Programming*, S. Even; O. Kariv, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1981, pp. 218–228.

[31] A. Okhotin; I. Olkhovsky: 'On the transformation of LL(k)-linear grammars to LL(1)-linear,' in *Computer Science – Theory and Applications*, H. Fernau, Ed., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 328–340.

[32] Z. Ahmed *et al.*: 'Bringing LTL model checking to biologists,' in *Verification, Model Checking, and Abstract Interpretation*, A. Bouajjani; D. Monniaux, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 1–13.

[33] N. Cardozo: 'A declarative language for context activation,' in *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-time Composition*, ser. COP '18, New York, NY, USA: Association for Computing Machinery, Jul. 16, 2018, pp. 1–7.

[34] F. Bernardo; C. Kiefer; T. Magnusson, 'A signal engine for a live coding language ecosystem,' *Journal of the Audio Engineering Society*, vol. 68, no. 10, pp. 756–766, Nov. 30, 2020.

[35] A. Izmaylova; A. Afroozeh; T. v. d. Storm: 'Practical, general parser combinators,' in *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '16, New York, NY, USA: Association for Computing Machinery, Jan. 11, 2016, pp. 1–12.

[36] S. Jarzabek; T. Krawczyk, 'LL-regular grammars,' *Inf. Process. Lett.*, 1975.

[37] A. Nijholt, 'LL-regular grammars,' *International Journal of Computer Mathematics*, vol. 8, pp. 303–318, Oct. 15, 1980.

[38] A. Nijholt: 'On the parsing of LL-regular grammars,' in *Mathematical Foundations of Computer Science 1976*, A. Mazurkiewicz, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1976, pp. 446–452.

[39] D. A. Poplawski, 'On LL-regular grammars,' *Journal of Computer and System Sciences*, vol. 18, no. 3, pp. 218–227, Jun. 1, 1979.

[40] P. Belcak, 'The LL(finite) strategy for optimal LL(k) parsing,' Jan. 20, 2021. arXiv: 2010.07874.

[41] R. Cohen; K. Culik: 'LR-regular grammars an extension of LR(k) grammars,' in *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, Oct. 1971, pp. 153–165.

[42] M. Tomita: *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. USA: Kluwer Academic Publishers, 1985, 201 pp.

[43] E. Scott; A. Johnstone, 'GLL parsing,' *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, pp. 177–189, Sep. 2010.

[44] T. Parr: *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Raleigh, N.C: Pragmatic Programmers, May 1, 2007, 361 pp.

[45] ECMA, 'ECMAScript language specification - ECMA-262 edition 5.1,' 2011. [Online]. Available: https://262.ecma-international.org/5.1/ (visited on 12/24/2021).

[46] H. M. Pandey, 'Advances in ambiguity detection methods for formal grammars,' *Procedia Engineering*, International Conference on Advances in Engineering 2011, vol. 24, pp. 700–707, Jan. 1, 2011.

[47] B. Basten: 'Ambiguity detection for programming language grammars,' Ph.D. dissertation, Universiteit van Amsterdam, Dec. 15, 2011.

[48] H. Hemmati: 'How effective are code coverage criteria?' In *2015 IEEE International Conference on Software Quality, Reliability and Security*, Aug. 2015, pp. 151–156.

[49] Antlr Project: *Grammars-v4*, Apr. 9, 2022. [Online]. Available: https://github.com/antlr/grammars-v4 (visited on 04/09/2022).

[50] M. Haverbeke: *Acorn*, GitHub. [Online]. Available: https://github.com/acornjs/acorn (visited on 12/29/2021).

[51] M. Bynens; J.-D. Dalton. 'Benchmark.js.' (2021), [Online]. Available: https://benchmarkjs.com/ (visited on 12/29/2021).

Appendix

Listings

```
class Parser extends CstParser {
    constructor() {
        super([a, b, c, d, e]) // An array of used tokens
        this.performSelfAnalysis()
    }

    A = this.RULE("A", () => {
        this.CONSUME(a)
        this.MANY(() => {
            this.CONSUME(b)
            this.CONSUME(c)
        })
        this.OPTION(() => {
            this.SUBRULE(B)
        })
    });

    B = this.RULE("B", () => {
        this.OR([
            {
                ALT: () => {
                    this.CONSUME(d)
                }
            },
            {
                ALT: () => {
                    this.CONSUME(e)
                }
            }
        ])
    })
}
```

Listing 1: A simple Chevrotain parser

```
function lookahead(alternatives) {
    const numOfAlts = alternatives.length
    for (let t = 0; t < numOfAlts; t++) {
        const currAlt = alternatives[t]
        const currNumOfPaths = currAlt.length
        nextPath: for (let j = 0; j < currNumOfPaths; j++) {
            const currPath = currAlt[j]
            const currPathLength = currPath.length
            for (let i = 0; i < currPathLength; i++) {
                const nextToken = this.LA(i + 1)
                if (!tokenMatcher(nextToken, currPath[i])) {
                    // mismatch in current path
                    // try the next path
                    continue nextPath
                }
            }
            // found a full path that matches.
            // this will also work for an empty alternative as
               the previous loop will be skipped
            return t
        }
        // none of the paths for the current alternative
           matched
        // try the next alternative
    }
    // none of the alternatives could be matched
    return undefined
}
```

Listing 2: Chevrotain Lookahead algorithm for $k > 1$

```
function singleTokenLookahead(alternatives) {
    const singleTokenAlts = map(alternatives, (currAlt) => {
        return flatten(currAlt)
    })

    const choiceToAlt = reduce(
        singleTokenAlts,
        (result, currAlt, idx) => {
            forEach(currAlt, (currTokType) => {
                if (!has(result, currTokType.tokenTypeIdx)) {
                    result[currTokType.tokenTypeIdx] = idx
                }
                forEach(currTokType.categoryMatches,
                   (currExtendingType) => {
                    if (!has(result, currExtendingType)) {
                        result[currExtendingType] = idx
                    }
                })
            })
            return result
        },
        {}
    )

    return function lookahead() {
        const nextToken = this.LA(1)
        return choiceToAlt[nextToken.tokenTypeIdx]
    }
}
```

Listing 3: Chevrotain Lookahead algorithm for $k = 1$

A Statutory Declaration

I herewith declare that I have composed the present thesis myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The thesis in the same or similar form has not been submitted to any examination body and has not been published. This thesis was not yet, even in part, used in another examination or as a course performance. Furthermore I declare that the submitted written (bound) copies of the present thesis and the version submitted on a data carrier are consistent with each other in contents.

Hamburg, 21.04.2022

Mark Sujew