Consider it Parsed!

Max Hendriks **Supervisor: Vadim Zaytsev**

Formal Methods and Tools, University of Twente, Enschede, Netherlands

ABSTRACT

Relational parsing with context-free memoization, first presented in 2020, promises generalised context-free parsing at speeds exceeding that of the current state-of-the-art parser generator: ANTLR4. Here, we present an independent attempt at bringing relational parsing to life based only on its original documentation, identify challenges when implementing it, and examine the capabilities of relational parsing. Our implementation of relational parsing can parse some challenging types of grammars, half of which ANTLR4 could not. However, there is a type of grammar that we cannot parse and for most tested grammars we did not manage to improve parsing speed by using context-free memoization. We expect these two caveats result from the way we implemented relational parsing rather than the method itself. We discuss some open questions that need answering before relational parsing can take its place as a practical and usable context-free parser.

Keywords: Parsing, Relational Parsing, Parser generators, ANTLR4, ALL(*)

INTRODUCTION

In this thesis we will discuss a recently developed method for parsing context-free languages called relational parsing, developed and introduced in a paper by Herman (2020). In his experiments he found that while relational parsers are generalised parsers, it was also able to outperform ANTLR4 and several other parser generators. We explore this method of parsing, identify some of the main challenges in implementing a relational parser and attempt to increase the amount of knowledge available on relational parsing.

First we introduce the concept of parsing in general, we explain what grammars are and what types exist. Then we continue on with an argument for why parser generators are useful and why we focus on parsing context-free languages. This is followed by a brief overview of the types of algorithms that exist for parsing context-free languages and an important technique that helps speed up these common parsing algorithms. Then we follow with an overview of the current industry standard, ANTLR4, and how its algorithm ”ALL(*)” functions. Further, we describe relational parsing, the challenges to overcome when implementing a relational parser and evaluate our attempt at bringing it to life.

GENERAL REVIEW OF LITERATURE

Parsing is the process of analysing a string of symbols conforming to the rules of a formal grammar. It can mean different things in different contexts. In linguistics, for example, it is commonly used to assign meaning to a sentence. For this purpose, sentences are parsed using a ”tree-diagrammatic representation of phrase structure rules called a PHRASE MARKER”. (Mitchell, 1994) Within computational linguistics, this same process is performed automatically by a computer, producing a parse tree. Some parsing algorithms may produce a parse forest for a syntactically ambiguous sentence input. Within computer science, the technique of parsing is commonly used to syntactically analyse input code into its component parts to then be compiled to machine code by a compiler or interpreted by an interpreter to be run as-is.

Grammars

In formal language theory, a grammar contains a list of production rules that describe how to form strings from an alphabet of symbols that are valid according to a language’s syntax. These grammars include a start symbol from which the production rules are applied sequentially to generate any string in the grammar’s language. Because of this, grammars are typically thought of as language generators, however, in computer science they are often used as language recognisers or parsers.

There are multiple types of grammars, that are commonly organised in a hierarchy called the Chomsky hierarchy (Chomsky, 1956). The Chomsky hierarchy organises grammars into several levels according to the types of production rules that are allowed in a grammar. The levels range from type-0 to type-3, where type-3 is the most restrictive type of grammar and type-0 is the least restrictive. The more restrictive the grammar, the easier it is to recognise and parse.

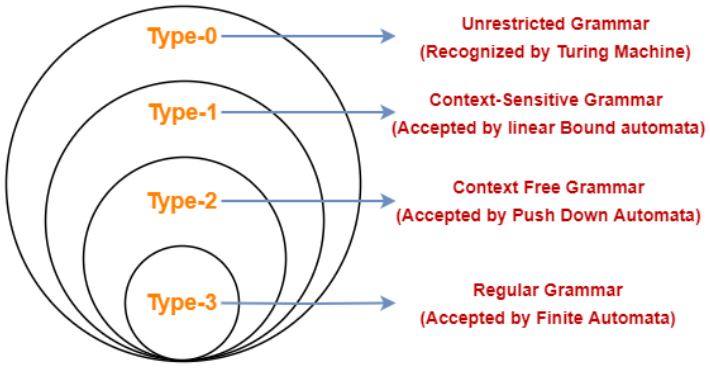


Figure 1. The Chomsky Hierarchy.

To describe these types of grammars, we represent any grammar G by a four-tuple (T, N, P, s_{start}) where T is the finite terminal alphabet and N is the finite nonterminal alphabet. P is a finite set of production rules written in the form of $A \rightarrow \omega$ where A is in N and ω in $(N \cup T)^*$. Finally, s_{start} in N is the starting symbol. Whenever we write a production rule, any capital letter represents a nonterminal in N, any lowercase letter represents a terminal in T and ε represents the empty string.

The language generated by a grammar is the set of strings without nonterminal symbols generated from the starting symbol s_{start}. If there are multiple ways of generating the same string, the grammar is said to be ambiguous.

The language generated by a grammar can be defined using the following notation $\{xyz\}$. In this example, the language consists only of the single string of terminals "xyz". If in a language definition we write x^j, we mean x concatenated to itself j times. If j=2, then this results in the string "xx".

In the following sections we describe the various types of grammars and provide some example grammars and productions. These example productions help give a feel for how grammars generate their strings, and are relevant to the discussion of parsers later on.

Type-3

Type-3 grammars, commonly called the regular grammars, come in two types: right-linear and left-linear. (Shi, 2021) In a right-linear language, all rules are one of the following forms:

$$A \rightarrow a$$
$$A \rightarrow aB$$
$$A \rightarrow \varepsilon$$

In a left-linear language, all rules follow the forms:

$$A \rightarrow a$$
$$A \rightarrow Ba$$
$$A \rightarrow \varepsilon$$

All regular languages generate strings of the form $\{a^i b^j ... | i, j, ... \in \mathbb{N}\}$, where there is no relation between i and j (or any other counters).

Right-linear rules and left-linear rules cannot be mixed in a single regular grammar. For example, the grammar with the rule set $\{S \rightarrow aB, B \rightarrow Sb, S \rightarrow \varepsilon\}$ is type-2, and describes the language $\{a^i b^i | i \in \mathbb{N}\}$.

All regular languages can be recognised by a nondeterministic finite automaton.

Type-2

Type-2 grammars are commonly referred to as context-free grammars. In this type of grammar, production rules are of the form $A \to \alpha$ where $A \in N$ is any nonterminal symbol and $\alpha \in (T \cup N)^*$ any string of terminal and nonterminal symbols. (Shi, 2021)

A simple context-free grammar is the following:

$$S \to aSb$$
$$S \to \varepsilon$$

An example production of this grammar is the following sequence $S \xrightarrow{1} aSb \xrightarrow{1} aaSbb \xrightarrow{1} aaaSbbb \xrightarrow{2} aaabbb$. All context-free grammars can be recognised by nondeterministic push down automata.

This class of grammars is especially interesting for designing programming languages, as this is the most restricted type where it is possible to enforce well-formed closing parentheses using only the grammar definition. Consider the following rule: $S \to (A)$. Clearly, applying this rule will ensure that all new tokens produced by A will be enclosed in two matching brackets. (Shi, 2021) This type of notation is very commonly used in programming languages to mark and define scopes, function definitions, calls and more.

Type-1

Type-1 grammars are also known as context-sensitive grammars. Production rules of context-sensitive grammars are of the form $\alpha A \beta \to \alpha \gamma \beta$ with $A \in N$, $\alpha, \beta \in (N \cup T)^*$ and $\gamma \in (N \cup T)^+$. This means that to apply the production rule A, it must exist in the context where α is on its left and β on its right. (Shi, 2021) Also, any nonterminal on the right of a context-sensitive rule is not allowed to be nullable with ε.

One canonical example of a context-sensitive language is the $\{a^n b^n c^n | a,b,c \in \mathbb{N}\}$, generated by the following grammar:

$$S \to abc$$
$$S \to A$$
$$A \to aABc$$
$$A \to abc$$
$$cB \to Bc$$
$$bB \to bb$$

An example production of this grammar with $n = 3$ is the sequence $S \xrightarrow{2} A \xrightarrow{3} aABc \xrightarrow{3} aaABcBc \xrightarrow{4} aaabcBcBc \xrightarrow{5} aaabBccBc \xrightarrow{6} aaabbccBc \xrightarrow{5} aaabbcBcc \xrightarrow{5} aaabbBccc \xrightarrow{6} aaabbbccc$.

All context-sensitive grammars can be recognised by linear bounded automata.

Type-0

The final class, type-0, is known as the class of unrestricted grammars. This class allows any type of production rule. Unrestricted grammars allow production rules of the form $\alpha \to \beta$ with $\alpha \in (T \cup N)^* N (T \cup N)^*$ and $\beta \in (T \cup N)^*$. (Shi, 2021)

An example of an unrestricted grammar generating the language $\{a^n b^n c^n | a,b,c \in \mathbb{N}\}$ is the following:

$$S \to aBSc$$
$$S \to \varepsilon$$
$$Ba \to aB$$
$$Bc \to bc$$
$$Bb \to bb$$

An example production of this grammar with $n = 3$ is the sequence $S \xrightarrow{1} aBSc \xrightarrow{1} aBaBScc \xrightarrow{1} aBaBaBSccc \xrightarrow{2} aBaBaBccc \xrightarrow{3} aBaaBBccc \xrightarrow{3} aaBaBBccc \xrightarrow{3} aaaBBBccc \xrightarrow{4} aaaBBbccc \xrightarrow{5} aaaBbbccc \xrightarrow{5} aaabbbccc$

While this grammar generates the same language as the example in the context-sensitive grammar section, it does so by applying fewer production rules. As this grammar class is unrestricted, it can be

used to generate many more languages than possible with the previous types of grammars. The languages generated by unrestricted grammars are also called recursively enumerable languages.

A recursively enumerable language is a language for which there exists a Turing machine, that will halt and accept when given a string in the language as input, but will either halt and reject or loop forever when given a string not in the language.

Why context-free for programming languages?

Context-free languages have quite a few limitations. Most programming languages contain context-sensitive features which cannot be expressed using only context-free grammars. A few examples of context-sensitive features identified by Laurent and Mens (2016) follow:

- In C, if the parser encounters x*y, it must decide whether the statement expresses the product of x by y or the declaration of a variable y that is a pointer to type x. This is done by analysing the definitions preceding this statement.
- In Haskell and standard ML, operators with custom precedence and associativity can be introduced by the programmer. These definitions have to be interpreted by the parser to parse the remainder of the input.
- In Python, code blocks are defined by indentation. Therefore, a Python parser needs to detect when the indentation level increases or decreases.
- In XML, opening tags and closing tags must be matched at the same level of nesting. <foo>< /foo> is valid while <foo>< /bar> is not. An XML parser must memorise the names of open tags at arbitrary levels of nesting.
- In TCP and other network protocols, length-delimited fields are used which do not have a fixed length. Instead, the length is indicated by a field that precedes them.

However, even if we try to express our programming languages using context-sensitive grammars, we will discover that there are language features that also cannot be fully expressed with a context-sensitive grammar. An example of this is variable declaration before use. Writing this rule into a context-sensitive grammar requires us to write derivation rules for every possible variable declaration. In languages with a small name space, this might be possible, but in modern languages with a large name space the set of names is too large to encode into context-sensitive grammars. (Cooper and Torczon, 2012)

However, as we travel up the Chomsky hierarchy, parsing algorithms become slower and slower. Context-free languages can be parsed in a reasonable time. The Earley parsing algorithm, for example, allows us to parse every context-free language in cubic time, which is much faster than any context-sensitive parsing algorithm. According to Cooper and Torczon (2012), parsing context-sensitive languages is P-Space complete, and going even higher up in the Chomsky hierarchy exacerbates this problem. Therefore, finding the correct level for our programming language grammars is a balancing act of parsing performance versus grammar expressivity.

To allow context-free grammars to parse programming languages with context-sensitive features, we write context-free grammars that parse at least all valid inputs for the programming language as well as some invalid ones. We then use either context-sensitive preprocessing to produce a context-free token stream and/or use post-processing techniques on the parse tree to filter out any invalid inputs, leaving only the valid options. As an example, the python preprocessor inserts ”INDENT” and ”DEDENT” tokens whenever an indentation change is detected.

Because context-free parsers are relatively quick, and because pre- and post-processing techniques allow us to still use context-sensitive language features with context-free grammars, context-free parsers are the parsers we should be using to parse our programming languages.

Recognisers and parsers

Given a string of symbols, we can ask whether this string is syntactically valid according to a particular grammar. An algorithm that decides this is called a recogniser. Language parsers are also language recognisers and attempt to discover which production rules can be applied to generate the input string. By doing this they produce a parse tree (or forest) when the input is valid and usually return an error if the input is invalid. Some parsers may also include functionality to handle encountered errors, producing

a best-effort parse tree that, while incomplete, may expose multiple errors rather than only the first one encountered, helping end-users write syntactically correct inputs.

Hand-written parsers and generated parsers

The parsers employed by computer scientists to build compilers for programming languages can commonly be divided into two categories: hand-written parsers and generated parsers. Most major programming languages employ hand-written parsers for their compilers. There are multiple reasons for this: Handwritten parsers tend to be more user-friendly as they can output more specific error messages, provide error recovery and well-written parsers are generally faster than generated parsers.

However, there are also benefits to using generated parsers: It is usually much easier to write a specification for the language than it is to manually code the language structure into a parser. Grammar definitions may also be much easier to maintain as grammar definitions usually flow more naturally than code. Finally, generated parsers may still be faster than hand-written parsers as writing a good parser is very difficult.

Because writing grammars for a parser generator is relatively easy, a language designer can design, test, and iteratively improve upon existing grammars much more quickly than if they work with a hand-written parser. This allows for more and quicker innovation in the design of new types of programming languages.

Caching and memoization

As Norvig (1991) explains, memoization refers to the process whereby a function remembers the results of previous computations and is a specific form of the more general method of caching. This technique can bring extreme efficiency gains when applied to, for example, recursive functions like those that calculate Fibonacci numbers:

```
fn fib(n) {
    if n <= 1 {
        return n;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

If the function is run as-is, many Fibonacci numbers will be calculated over and over: when calculating fib(5), fib(2) is calculated three times. If instead, we remember all previous computations in a table, we can calculate every value once, enter it into the table and retrieve this computed value whenever we need it in the future. Applying this optimisation to the above fib(n) algorithm transforms it from a $O(c^n)$ algorithm to a $O(n)$ one.

The technique of memoization can be used in parsing applications to bring great performance boosts. Take for example the LL(1) algorithm (further described later); it performs memoization by precomputing a parse table from the target grammar. This simplifies parsing as the algorithm does not have to search the grammar's list of rules when trying to predict a rule when matching a terminal symbol to a nonterminal one. Instead, the precomputed table will tell the algorithm that it should apply a certain rule or spit out an error because the input string is invalid.

Most parsing algorithms include at least one method of memoization as not including it means the algorithm would be unable to parse anything efficiently. However, one also cannot simply memoize everything as this would lead to an immense memory requirement and lookup overhead. Therefore, different algorithms memoize different things in different ways to find the biggest performance gain for that particular algorithm.

Algorithms

We will look at several parsing algorithms to give a snapshot of the types of algorithms that already exist, what their properties are and why there is a need for developing other algorithms.

LL(k) algorithms

LL(k) parsers are top-down parsers which parse grammars using a fixed amount of look-ahead where k determines the amount of look-ahead a parser uses. The languages parsed by LL(k) parsers are called

LL(k) languages and the grammars generating these are called LL(k) grammars. Ambiguous grammars are not LL(k) and unambiguous grammars are not necessarily LL(k). One important class of LL(k) parsers are the LL(1) parsers which parse their inputs with a look-ahead of 1 token. Lewis and Stearns (1968) were one of the first to define the LL(k) property and explain how to construct an LL(k) recogniser.

LL(k) parsers are usually extremely fast, often parsing their inputs in linear time. If a non-LL(k) grammar generates an LL(k) language, it can be rewritten to become an LL(k) grammar. However, if we rewrite a grammar into an LL(k) form, its structure may change drastically, often making its structure more complicated. This may also become more difficult to find errors in the grammar and generated parse trees may no longer reflect the structure originally desired. This then requires generated parse trees to be modified at a later step to conform to the original structure. If rewriting a non-LL(k) grammar is undesired or impossible, a general parsing algorithm is needed. LL(k) parsing is described in-depth in Parsing Techniques by Grune and Jacobs (2008).

Earley

As described by Aycock and Horspool (2001), Earley's algorithm is a general parsing algorithm, meaning it can recognise and parse inputs described by any context-free language. Earley parsers are often described as top-down parsers with bottom-up recognition. In essence, it functions by producing predictions in a top-down manner from the grammar and then performing bottom-up shifts and reduces to recognise the input.

As Earley (1970) wrote when he introduced his parsing algorithm, the general case complexity of Earley's algorithm is $O(n^3)$ and $O(n^2)$ for unambiguous grammars, where n represents the length of the input string.

LR(k) algorithms

LR(k) algorithms are bottom-up parsers with a top-down part that function very similar to Earley's algorithm. LR(k) parsers execute a top-down exploration of a given grammar to see what states are reachable and what types of reductions can be performed by a parser in practice for that grammar.

However, compared to Earley, LR(k) grammars put restrictions on the types of grammars accepted so that the algorithms can be run deterministically and in $O(n)$ time. LR(k) parsers parse by building a deterministic automaton that recognises parts of the input which can be reduced into a grammar nonterminal.

Just like with LL(k), the k in LR(k) refers to the amount of look-ahead the parsers can use to parse input strings. The first class LR(0) parses without any look-ahead but is extremely limited in its parsing capabilities. LR(1) expands this a bit and higher numbers will allow for more grammars. However, some grammars cannot be parsed by any LR(k) parser. For example, the grammar $S \rightarrow aSa|a$ producing strings with an odd number of as is impossible to parse using LR(k). This is because LR(k) cannot recognise reductions in the middle of a string. A detailed description of LR(k) can be found in Parsing Techniques (Grune and Jacobs, 2008).

Variants of LL and LR

Many other parsing algorithms build upon LL(k) or LR(k) to expand their efficiency, reduce memory requirements or increase the number of compatible grammars.

Take for example LALR(1), which takes LR(0) and expands it with a look-ahead capability of 1. This is different from LR(1) in the fact that the deterministic automaton of LALR(1) is the same size as that of LR(0), and is exponentially smaller than that of LR(1) and uses only a small amount of extra memory compared to LR(0). Another variant of LR(k) is SLR(k). An SLR(1) parser is the same size as an LALR(1) parser, but is less powerful. Therefore, LALR(k) parsers are generally preferred. Both LALR(k) and SLR(k) parsers were first introduced by DeRemer (1969).

An algorithm for unbounded LL parsing also exists, called LL(*) (Parr and Fisher, 2011). LL(*) parsers use an arbitrary look-ahead that changes to be the minimum look-ahead required to recognise rules.

Generalised versions of LL and LR (called GLL (Scott and Johnstone, 2010) and GLR (Tomita, 1985a)), can parse any context-free grammar are generally obtained by relaxing the restrictions of deterministic LL and LR parsing and using either depth-first or breadth-first search methods whenever non-deterministic sections in the grammar are encountered.

Detailed descriptions of LALR(1), GLL and GLR and other algorithms can also be found in Parsing Techniques (Grune and Jacobs, 2008).

Data-dependent parsing with Iguana

As introduced in a paper by Afroozeh and Izmaylova (2016), Iguana is a parser generator built upon GLL and extends the class of parseable grammars from the purely context-free to a class called data-dependent. Data-dependent grammars extend context-free grammars with arbitrary computation, variable binding, and constraints. This allows users to add syntactic constructs such as operator precedence and indentation-sensitive constructs to a language without needing to modify the internal machinery of a parsing algorithm. Such constructs can be specified at a high level - above the data-dependent level - and be automatically desugared by Iguana to a data-dependent grammar.

ANTLR4 and ALL(*)

ANTLR4 (hereafter called ANTLR) is a modern parser generator. According to Parr (2022), Twitter uses ANTLR to parse search queries, Lex Machina to extract information from legal texts, Oracle for their SQL Developer IDE and more. In essence, it is the current-day standard in both industry and academia for the development of languages, tools and frameworks.

When given a suitable grammar, ANTLR can generate a parser for various host programming languages like Java, C#, Python and more. The parsing algorithm it uses is a variation of LL(*) called Adaptive LL(*) (henceforth called ALL(*)). Even though this is a top-down algorithm, ANTLR can parse most left-recursive grammars. It does this by translating the grammar into a non-left-recursive grammar before passing it on to ALL(*). ANTLR rejects grammars with indirect left-recursive rules.

As Parr et al. (2014) explains, ANTLR grammars use yacc-like syntax with EBNF operators like the Kleene star (*) and token literals in single quotes. In this way, the grammars contain both lexical and syntactic rules in a combined specification. In addition to standard grammar rules, programmers are also able to include side-effecting actions (semantic actions). These semantic actions are written in the host language to which the parser will compile and can access and modify the internal state of the parser. Side-effect-free actions (semantic predicates) are also supported. These semantic predicates are boolean expressions written in the host language determining the validity of a particular production rule and are also one of the features supported by data-dependent grammars. Semantic actions and predicates dramatically increase the power of ALL(*) as they provide some context-sensitive decision-making while parsing an otherwise context-free grammar. Figure 2 is an example of a left-recursive grammar accepted by ANTLR.

```
grammar Ex; // generates class ExParser
// action defines ExParser member: enum_is_keyword
@members {boolean enum_is_keyword = true;}
stat: expr '=' expr ';' // production 1
    | expr ';'          // production 2
    ;
expr: expr '*' expr
    | expr '+' expr
    | expr '(' expr ')' // f(x)
    | id
    ;
id  : ID | {!enum_is_keyword}? 'enum' ;
ID  : [A-Za-z]+ ; // match id with upper, lowercase
WS  : [ \t\r\n]+ -> skip ; // ignore whitespace
```

Figure 2. Sample left-recursive ANTLR 4 predicated grammar Parr et al. (2014)

As explained by Parr et al. (2014), ALL(*) is a top-down parsing strategy and supports all non-left-recursive grammars.

Parsers like LL(k) rely on static analysis of the input grammar. ALL(*) instead moves this analysis to run-time and incrementally builds a library of look-ahead deterministic finite automata (DFAs), one for every nonterminal in the grammar. A DFA consists of a set of states S, a finite set of input symbols T, a transition function $\delta : S \times T \rightarrow S$, an initial start state S_{start}, and a set of accept states S_{accept}. Because DFAs are deterministic by definition, every state may only have one outgoing transition for each input

symbol. Given a state $s \in S$ and an input symbol $t \in T$, the transition function outputs the next state s', if any exists. If, by following a string of input symbols, a state is returned that is in the set of accept states, the DFA is said to accept that string.

Leveraging the above process, Parr matches look-ahead phrases to predicted production rules with prediction function *adaptivePredict*. Familiar phrases can be quickly looked up in the look-ahead DFA, allowing ALL(*) to skip grammar analysis for these phrases. Unfamiliar phrases trigger the grammar analysis mechanism predicting a production rule to apply and updates the DFA with states and transitions to include this new phrase. The analysis is performed by a "GLR-like mechanism" that explores all possible decision paths for the current stack of in-process nonterminals and remaining input tokens. This mechanism of prediction and subsequent DFA construction is how ALL(*) performs memoization to speed up its parsing efforts.

The DFAs are constructed similarly to how the well-known NFA-to-DFA subset construction algorithm functions. However, instead of mapping a nondeterministic finite automaton (NFA), ALL(*) internally simulates the actions of an augmented transition network (ATN) representation of a grammar. Therefore ALL(*) constructs its DFAs from the simulated ATN instead. As DFA construction is a major part of what makes ALL(*) fast, we will discuss how it works.

DFA construction

As Parr explains Parr et al. (2014), when *adaptivePredict* reaches a decision point for the first time, it initialises a look-ahead DFA for that decision. The DFA start state D_0 represents the set of ATN subparser configurations reachable without consuming any input symbols. Using the *stat* rule from the grammar in Figure 2, we have the ATN subgraph in Figure 3. The DFA construction of D_0 would first add ATN configurations $(p, 1, [])$ and $(q, 2, [])$ where p and q are ATN states corresponding to production rules 1 and 2 and $[]$ is the empty subparser call stack.

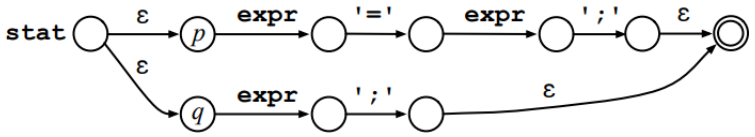
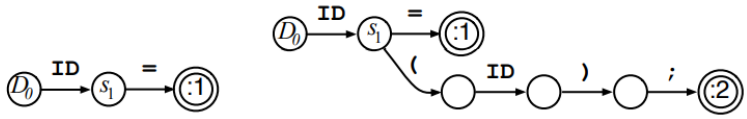


Figure 3. ATN for ANTLR rule stat in Figure 2 Parr et al. (2014)

Following the construction of D_0, a new DFA state is created indicating where ATN simulation could reach after consuming the first look-ahead symbol and connecting the previous and new states with an edge labelled with that symbol. This process is repeated until all ATN configurations in a newly-created DFA state predict the same production. The last state is marked as an accepting state and returns to the parser with that production number.

Figure 4a depicts the look-ahead DFA for decision *stat* after analysing the input $x = y;$. If a similar phrase is encountered again, the DFA does not look beyond $=$ as that symbol is sufficient to distinguish expr's production rules. :1 in the accept state means "predict production 1".



(a) After `x=y;` (b) After `x=y;` and `f(x);`

Figure 4. look-ahead DFA for decision stat in Figure 3 Parr et al. (2014)

Oftentimes, *adaptivePredict* finds an existing DFA for a particular decision. If a DFA exists, *adaptivePredict* looks for a path from the start state to an accepting state. If a non-accepting DFA state is reached without an edge for the current look-ahead symbol, the DFA is extended by simulating the ATN. For example, when analysing another input phrase for *stat* like $f(x);$, *adaptivePredict* finds an existing *ID* edge from D_0 and jumps to s_1 in the DFA. However, there is no existing edge from s_1 for (so

adaptivePredict simulates the ATN to build a path to an accepting state. In this case, a second path is built predicting the production rule 2 (Figure 4b).

Because the sequence $ID(ID)$ predicts both production rules 1 and 2, the simulation continues until the DFA has edges for both the = and ; symbols. If ATN simulation computes a target state that is already included in the DFA, a new edge is added targeting that state and then continues with DFA simulation. Adding edges like this introduces cycles in the DFA.

As *adaptivePredict* encounters more and more unfamiliar phrases, the DFA is extended more and more, thereby decreasing the likelihood that unfamiliar phrases are encountered in the future. And because DFA simulation is quicker than ATN simulation, parsing will speed up as it goes on.

Two-stage parsing

Two-stage parsing is another optimisation that makes parsing with ALL(*) quick. To explain the need for two-stage parsing, we first need to understand that ALL(*) cannot always rely on the look-ahead DFA to make a correct parsing decision. Occasionally, ALL(*) prediction must consider the parser call stack at the start of prediction. The following is a simple grammar that exhibits a stack-sensitive decision in nonterminal A: $S \rightarrow xB|yC \quad B \rightarrow Aa \quad C \rightarrow Aba \quad A \rightarrow b|\varepsilon$.

Without the parser stack, A's productions cannot be distinguished by any look-ahead. Taking look-ahead ba, $A \rightarrow b$ is predicted when B reaches A, but $A \rightarrow \varepsilon$ is predicted when C reaches A. If the call stack is ignored in this situation, a prediction conflict is reached upon look-ahead ba. Parsers that ignore the call stack for prediction are called Strong LL (SLL) parsers.

Creating a different look-ahead DFA for every possible parser call stack is not feasible as the number of stack permutations is exponential to the stack depth. Instead, ALL(*) takes advantage of the fact that most decisions are not stack-sensitive and builds DFA ignoring the stack. If SLL parsing finds a prediction conflict, it is unknown whether the phrase is ambiguous or stack-sensitive. In that case, the look-ahead must be re-examined using the parser stack. This hybrid stack-sensitive/stack-insensitive parsing is called optimised LL mode. It improves performance by caching stack-insensitive prediction results in look-ahead DFA wherever possible and retains full stack-sensitive capabilities when the DFA is inconclusive.

As SLL is weaker but faster than LL it makes sense to parse as much of the input as SLL and because Parr Parr et al. (2014) has found that most decisions are SLL in practice, ALL(*) attempts to parse entire inputs in SLL mode only. If SLL finds a syntax error, this may be either an SLL weakness or a real syntax error, so the entire input is then parsed again in optimised LL mode. These are the two stages in ALL(*)s two-stage parsing strategy. While two-stage parsing may parse the entire input twice, Parr has found that in practice it can dramatically increase the speed over using only optimised LL mode.

RELATIONAL PARSING

In 2020, Herman introduced his new parsing technique called relational parsing Herman (2020). Similar ANTLR and other parsing techniques, relational parsing simulates possible runs of pushdown automata (PDAs). It inductively computes the relations between languages of PDA configurations and the ways of reaching them when reading consecutive input symbols. These languages can be computed from the previous one by a constant number of simple, well-known language operations: union, concatenation and Brzozowski derivatives. These languages, called atomic languages, are precomputed from the grammar. They are regular and are represented as NFAs. For the languages computed during parsing, a directed acyclic graph (DAG) structure is used. As all cyclicity is embedded within the NFAs of atomic languages themselves, we never need to add edges to existing DAG nodes.

In addition to introducing the relational parsing algorithm, Herman also introduces a technique for context-free memoization. It exploits the typical structure of the DAG representation, splitting it into a stack of smaller components. Each input symbol can be handled by only examining the top element of this stack and can therefore be memoized. This simplifies the parsing of many input symbols to a dictionary lookup and a few operations on a stack.

Important operations

Several language operations are important for relational parsing; these operations are described below. In these following examples, we denote by S the combined set of symbols of G: $S = T \cup N$.

Left-linear closure

The left-linear closure of a language $\Sigma \subseteq S^*$ is the language $[\Sigma]$ obtained by applying any number of production rules to the leftmost nonterminal in any string $\sigma \in \Sigma$: $[\Sigma] = \{\sigma' \in S^* | \sigma \overset{*}{\mapsto} \sigma' \text{ for some } \sigma \in \Sigma\}$.

Brzozowski derivatives

The Brzozowski derivative of a language Σ by a string s is the language $\Sigma^{(s)}$ containing all strings obtainable by cutting off the prefix s: $\Sigma^{(s)} = \{\sigma' | s \cdot \sigma' \in \Sigma\}$. Note that the Brzozowski derivative acts as both a mutator and a filter, as only the strings containing the prefix s are included in the resulting set. Any strings without the prefix are excluded altogether.

Atomic languages

To efficiently parse symbols, we precompute the set of atomic languages $\mathbb{\Gamma}_G$ of G. This set consists of Σ_ε, which is the language before parsing any symbol, and $[s]^{(t)}$ for each $s \in S$ and $t \in T$, which is the left-linear closure followed by the Brzozowski derivative of every symbol s and terminal t in the alphabet of G.

A nonterminal in a grammar is said to be nullable if there exist rules which, if applied, result in no terminal symbols being generated from that nonterminal. The simplest way to make a nonterminal nullable is by adding an epsilon rule for that nonterminal. For example, to make nonterminal S nullable, we add $S \to \varepsilon$ to the grammar.

These nullable symbols may add extra complexity during parsing, so most parsing algorithms find some way to eliminate these nullable symbols by, for example, rewriting the grammar. This is often done by removing epsilon rules from nullable nonterminals and adding rules where the nullable nonterminal appears in the right hand side with that nonterminal removed. We then have the rule in both its normal form and in its "nulled" form.

Relational parsing also removes nullable nonterminals, but instead of doing this by rewriting the grammar, we integrate the nulling of nonterminals into the computation of atomic languages. This means that the atomic languages in practice are not exactly the set $[s]^{(t)}$, but $\underline{[S]^{(t)}}$, meaning we add words with all possible combinations of nullable symbols removed.

To illustrate what these atomic languages look like with and without nulling, consider the following grammar:

$$S \to a$$
$$S \to Sa$$
$$S \to SbSc$$

The atomic languages for this grammar without nullable rules are the following:

$$\begin{aligned} \underline{\Sigma_\varepsilon} &= S \\ \underline{[S]^{(a)}} &= (a + bSc)^* \\ \underline{[a]^{(a)}} = \underline{[b]^{(b)}} &= \underline{[c]^{(b)}} = \varepsilon \end{aligned}$$

Note that atomic languages are always regular, so in our notation for $[S]^{(a)}$ we used the Kleene star operator X^* to denote "zero or more instances" and $+$ for choice.

Now, when we add a rule nulling S to the grammar: $S \to \varepsilon$, the set of atomic languages becomes:

$$\begin{aligned} \underline{\Sigma_\varepsilon} &= S + \varepsilon \\ \underline{[S]^{(a)}} &= (a + bSc + bc)^* \\ \underline{[S]^{(b)}} &= (Sc + c)(a + bSc + bc)^* \\ \underline{[a]^{(a)}} = \underline{[b]^{(b)}} &= \underline{[c]^{(c)}} = \varepsilon \end{aligned}$$

Recognising a string

Now, to recognise a string $\tau = t_1 \cdots t_n$, we start with the language Σ_ε and inductively compute $\Sigma_{t_1 \cdots t_k} = [\Sigma_{t_1 \cdots t_{k-1}}]^{t_k}$. Each language obtained is called a phase of Σ and the computation itself is called a phase shift. Finally, to check whether the string τ is recognised, we check whether $\varepsilon \in [\Sigma_\tau]$

Precomputing the atomic languages of G allows us to forego computing left-linear closures and most derivatives while parsing and instead reduces most of the recognition effort to prepending atomic languages to the current language and performing some derivatives on the result, which is much quicker and simpler.

Memoization of recognition

While the set of precomputed atomic languages already serves as a simple form of memoization, with relational parsing we can do even better; Consider the language $\underline{\Sigma_\tau}$ computed after processing the input prefix τ and imagine we can factorise it as $\underline{\Sigma_\tau} = \Sigma \cdot \Delta$, with $\varepsilon \notin \Sigma$. If we process the next input symbol t, the language we compute is:

$$\begin{aligned}
\underline{\Sigma_{\tau \cdot t}} &= \bigcup_{s \in S} \underline{[s]^{(t)}} \cdot \underline{\Sigma_\tau^{(s)}} \\
&= \bigcup_{s \in S} \underline{[s]^{(t)}} \cdot (\Sigma \cdot \Delta)^{(s)} \\
&= \bigcup_{s \in S} \underline{[s]^{(t)}} \cdot \Sigma^{(s)} \cdot \Delta \\
&= (\bigcup_{s \in S} \underline{[s]^{(t)}} \cdot \Sigma^{(s)}) \cdot \Delta
\end{aligned}$$

This means that the computation of the next phase depends only on Σ and Δ is a context that does not have to be considered. If we need to compute a phase from a language $\Sigma \cdot \Delta'$ we can reuse the prefix $\bigcup_{s \in S} \underline{[s]^{(t)}} \cdot \Sigma^{(s)}$ and concatenate it with Δ'.

Parsing a string

Up to now, we have only concerned ourselves with recognising strings, to move on to full parsing, we need to not only consider parsed languages but the relations between these languages and the derivations by which they can be reached.

For a sting $\tau \in T^*$, a relation $\emptyset \subseteq S^* \times D^*$ is τ-parsed if for every $\gamma \in S$ we have
$s_{start} \overset{\delta}{\Rightarrow} \tau \cdot \gamma \iff \exists (\sigma, \overleftarrow{\delta_1}) \in \emptyset : \delta = \delta_1 \cdot \delta_2$ and $\sigma \overset{\delta_2}{\Rightarrow} \gamma$.

Achieving this is simple in theory, one simply needs to substitute atomic languages for the corresponding atomic relations. However, one complication in practice is the presence of nullable symbols, as we cannot simply skip arbitrary instances of nullable symbols like we can in atomic languages. We need to keep track of the positions of nullable symbols and the derivation steps that lead to these nullings.

To solve this problem, we define positively parsed relations over $(S \cup D)^* \times D^*$ where earlier derivation steps capture the way of reaching a particular configuration and later steps the pending erasure of nullable symbols. These positively parsed relations allow us to specify the location and erasure of nullable symbols and are sufficient to express any atomic relation.

Memoization of parsing

Context-free memoization also poses a challenge when computing derivations; While factors of parsed languages repeat very often in practice as Herman demonstrated Herman (2020), leftmost derivations never repeat as each prefix of such an input has a different number of parsed terminals. This means that direct memoization of the parsing information, or labels, will not work; It is necessarily different at every computed phase.

However, these labels are a function of those beforehand, and this function depends on only the factors of parsed languages that do repeat. Therefore, if we memoize not the labels themselves, but the expressions by which we can compute new labels from old ones, we can leverage the repetition uncovered by relational parsing.

Existing experimental results of relational parsing, ALL(*) and others

ALL(*) performance relative to other methods

In his paper on ALL(*) Parr et al. (2014), Parr includes a report on the performance of ALL(*). The theoretical complexity of ALL(*) parsing is $O(n^4)$. In addition to these theoretical results, Parr empirically compared ALL(*) to other popular parsing algorithms when parsing the 12,920 source files of the Java 6 library and compiler. The graphs resulting from these tests can be seen in Figure 5.

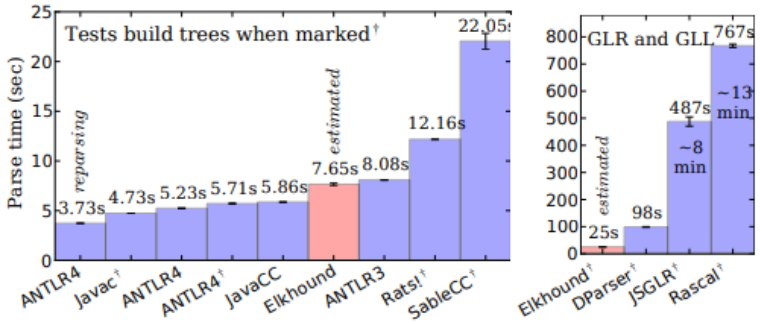


Figure 5. "Comparing Java parse times for 10 tools and 8 strategies on Java 6 library and compiler source code (smaller is faster)" Parr et al. (2014)

In their experiment, ALL(*) outperforms all other parser generators and only performs about 20% slower than the hand-built parser of the Java compiler itself. More detailed results and possible explanations can be found in the original paper by Parr Parr et al. (2014).

Relational parsing performance relative to ALL(*) and other methods

As Herman (2020) calculates, the theoretical complexity of relational parsing matches that of the state-of-the-art parsing algorithms; it is $O(n^3)$, $O(n^2)$ on unambiguous grammars and $O(n)$ on LR-regular grammars. To empirically compare relational parsing to available tools, Herman tested relational parsing with context-free memoization against ANTLR4, YAEP, and Iguana using ANTLRs "Java8" grammar and the elasticsearch java source files as well as with a "simple, unambiguous grammar for XML" with a single file containing "about 4.5M terminals and rich internal structure".

Four tests were performed; recognition (check whether the input matches the grammar), counting (calculating the number of parse trees without producing them), tree (producing an arbitrary parse tree) and forest (representing all possible parse trees), though not all tools supported all types of tests. The results of these experiments can be seen in Figures 6 and 7.

In general, relational parsing with context-free memoization was reported to be faster than all other algorithms in all modes of testing.

METHODOLOGY

Replications in Computer Science

As discussed by Shepperd (2018), replication studies in computer science are not automatically useful. This is especially the case when studies performing small experiments with a small effect are exactly replicated, i.e. reproducing the same small experiments. In these cases, it is trivial to replicate the results of, and therefore confirm, another study, resulting in almost no gain of knowledge. However, performing replication studies to enhance the trustworthiness of an empirical result is considered to be a central tenet of the scientific method and it, therefore, stands to reason, that properly performing a replication study is also useful in the field of computer science.

In particular, Shepperd (2018) makes four recommendations to make proper findings within the field of computer science:

1. Provide information on the dispersion (variance) of the variables involved in experiments.

2. Construct prediction intervals before conducting a replication.

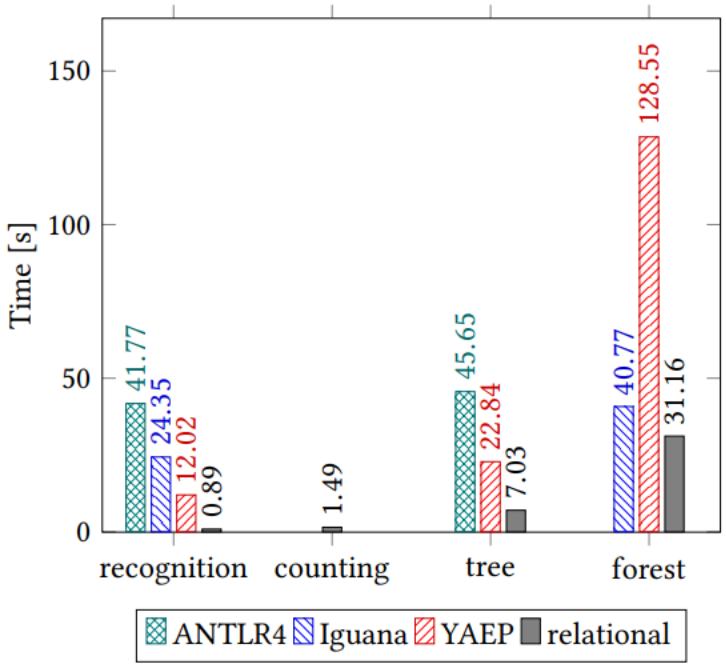


Figure 6. ”Parsing time for highly ambiguous Java grammar” Herman (2020)

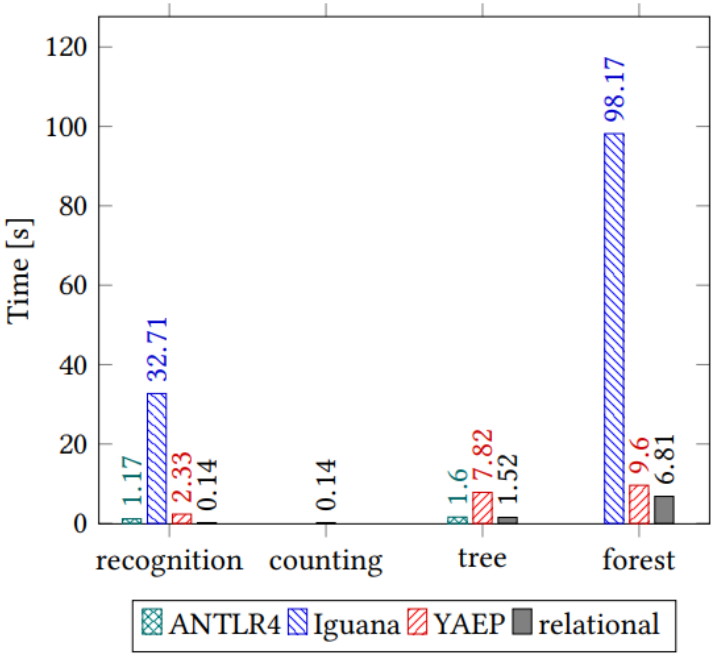


Figure 7. ”Parsing time for the unambiguous XML grammar” Herman (2020)

3. Limit replications to matters of reproducibility.

4. Conduct independent studies of important research questions where the effects matter to practising software engineers and combine results using meta-analytic techniques. Avoid close replications as these may violate the underlying assumption of independence and consider corrections needed due to potential publication bias of the first study.

Why replicate relational parsing?

In the original paper on relational parsing (Herman, 2020), Herman gave a very high-level mathematical explanation of relational parsing and provided evidence that relational parsing may have very good performance when compared to state-of-the-art parser generators. To confirm that this novel technique is indeed able to perform on par or better than, for example, ANTLR, more research is needed on relational parsing.

To aid this investigation we attempt to re-implement relational parsing according to the explanation by Herman (2020) and then, in more detail, describe the various steps necessary to implement a relational parser generator and evaluate its speed on several types of grammars. The source code of the original work is publicly available (Grzegorz, 2023), however, to examine how feasible it is to implement the algorithms using only its published documentation as well as any potential shortcomings of this documentation, we decided to implement relational parsing based only on Herman's paper, ignoring the previous implementation.

If this allows us to describe (some of) the strengths and weaknesses of relational parsing and make it more accessible to implement, this may make relational parsing easier or more interesting to investigate in the future.

DESIGN OF RELATIONAL PARSING RE-IMPLEMENTATION

We will now discuss the re-implementation of relational parsing produced for this study. We start off explaining our choice of programming language and then move on to explaining the various algorithms implemented. The source code can be viewed at `https://github.com/BurritoZz/Relational-Parsing-Rust`.

Choice of programming language

Rust is our choice of language for the re-implementation as it is one of the faster languages while also guaranteeing memory safety. When changes are made to underlying data structures, its strong typing system helps keep methods in sync with the data structures they operate on, allowing for easier incremental changes to the software.

Base data structures

We start the discussion of our implementation by listing some important data structures underlying the rest of the implementation.

Starting with the types necessary to define grammar; Terminal symbols are defined to be lowercase characters. Similarly, nonterminal symbols are defined as uppercase characters. These symbols can be composed into words via the Symbol enum, unifying the Terminal and Nonterminal types and adds an Epsilon variant to be used for representing empty words. The Word type represents strings of Symbol elements.

Using the above types, we define a grammar as follows; The Terminals and Nonterminals are collected in HashSets of the corresponding types. The start symbol must be one of the Nonterminals in the Nonterminals set. Derivation rules are stored as HashMaps, mapping a Nonterminal to its corresponding HashSet of Words. Every Word in such a set represents one derivation rule.

Finally, we have a field for storing the grammar's finite state automaton. The finite state automaton is calculated according to the other fields in the grammar and is stored here to aid in our parsing efforts later on. The definition and construction algorithm of the finite state automaton will follow later.

Atomic language calculation

The calculation of atomic languages from the starting grammar follows two phases; First, we calculate the regular expressions by which we can represent these atomic languages, and then convert these regular expressions into a finite state automaton for use in the parsing phase.

Regular expression calculation

The calculation of regular expressions is best explained via the use of an example. We will use the following grammar with an epsilon rule (hereafter abbreviated to e-rule) and refer to it as the "e-rule grammar".

$$S \to \varepsilon$$
$$S \to a$$
$$S \to Sa$$
$$S \to SbSc$$

We start by making a queue for all nonterminal and terminal combinations: $[S]^{(a)}$, $[S]^{(b)}$, and $[S]^{(c)}$. For this grammar, these are the three possible atomic languages that may be generated by the grammar.

For every potential atomic language, we follow the same process: First, we sort all rules belonging to the nonterminal into three sets: direct, recursive and different_atomic. The rules that start with the terminal symbol on their right-hand side go into the direct set. Those rules starting with the same nonterminal as the atomic language itself go into the recursive set. Rules starting with a nonterminal different from the current nonterminal are added to the different_atomic set.

Any rules starting with a terminal that is different from the terminal of the atomic language we are calculating are discarded. Epsilon rules tell us that nonterminal can be nulled but are otherwise not used.

The rules in the recursive and different_atomic sets do not start with the necessary terminal symbol to directly produce a Brzozowski derivative by that terminal, we, therefore, do not know yet whether we can perform the derivation necessary. To represent this uncertainty, we replace the starting nonterminal of these rules with a new placeholder symbol representing an atomic language. Later we may be able to substitute finished atomic languages into these placeholders and add the rule to our atomic language.

Next, we look at whether any of the rules have nullable nonterminals. We go through every set and find out whether any nonterminals in these rules can be nulled by applying one or more rules. If so, we make new rules for all possible combinations of nulled and non-nulled symbols and sort these new rules among the three sets as before.

Of importance in this process is that we keep track of the position of nulled symbols and the rules used to null them. To do this, we add placeholder symbols containing the applied derivation rules. Note that to null a nonterminal we may apply more than one derivation rule, if, for example, we have a grammar with rules like $S \to A, A \to \varepsilon$, nonterminal S could be nulled by applying those two rules.

The new rules generated for the above grammar are the following: for $S \to Sa$ we get the new rule $S \to \varepsilon a$. For $S \to SbSc$ we get $S \to \varepsilon bSc$, $S \to Sb\varepsilon c$, and $S \to \varepsilon b\varepsilon c$. Here we use ε as a placeholder symbol representing a nulled nonterminal. In our implementation, we use the enum variant representing applied nulling rules.

For the above grammar and atomic language $[S]^{(b)}$, the three sets would look as follows:

$$direct : \{S \to \varepsilon bSc, S \to \varepsilon b\varepsilon c\}$$
$$recursive : \{S \to [S]^{(b)}a, S \to [S]^{(b)}bSc, S \to [S]^{(b)}b\varepsilon c\}$$
$$different\_atomic : \emptyset$$

From these sets we now construct the regular expression by the following procedure: Rules in the direct set are collected and form the basis of our regular expression: $(\varepsilon bSc + \varepsilon b\varepsilon c)$.

The rules in the recursive set are similarly collected, but we factor out the placeholder symbol at the start and because the rules are recursive and can therefore be applied as many times as we want we add the kleene-star: $(a + bSc + b\varepsilon c)^*$.

If the different_atomic set is empty as in our example, we are finished and the final regular expression is formed by concatenating the direct and recursive portions: $[S]^{(b)} = (\varepsilon bSc + \varepsilon b\varepsilon c)(a + bSc + b\varepsilon c)^*$. Note that the only difference between this expression and the example given in is that here we keep track of nulled symbols. $[S]^{(a)}$ is calculated in a similar manner. When attempting to calculate $[S]^{(c)}$ however, we find that both the direct and different_recursive sets are empty, with only the recursive set containing anything. In such cases, the atomic language cannot be calculated and that terminal-nonterminal pair has no atomic language.

If we have a grammar for which the different_atomic set is not empty, things become more complicated. Take, for example, the following indirectly left-recursive grammar:

$$A \to Ba$$
$$A \to a$$
$$B \to Ca$$
$$B \to b$$
$$C \to Aa$$
$$C \to c$$

The sets for $[A]^{(c)}$ would look as follows:

$$\begin{aligned} direct &: \emptyset \\ recursive &: \emptyset \\ different\_atomic &: \{A \to [B]^{(c)}a\} \end{aligned}$$

Now, to calculate $[A]^{(c)}$, we need to first calculate $[B]^{(c)}$. Its sets are similar:

$$\begin{aligned} direct &: \emptyset \\ recursive &: \emptyset \\ different\_atomic &: \{B \to [C]^{(c)}a\} \end{aligned}$$

Finally, for $[C]^{(c)}$, we have:

$$\begin{aligned} direct &: \{C \to c\} \\ recursive &: \emptyset \\ different\_atomic &: \{C \to [A]^{(c)}a\} \end{aligned}$$

Even though we now have something in the direct set, the different_atomic set for $[C]^{(c)}$ is not empty and $[C]^{(c)}$ is therefore still dependent on $[A]^{(c)}$. To solve this circular dependency we take rules out of the different_atomic sets, perform substitutions on them and sort the new rules into one of the three sets again until we have no more rules in the different_atomic sets.

After this substitution process is applied a few times, the sets for $[A]^{(c)}$ should look as follows:

$$\begin{aligned} direct &: \{A \to caa\} \\ recursive &: \{A \to [A]^{(c)}aaa\} \\ different\_atomic &: \emptyset \end{aligned}$$

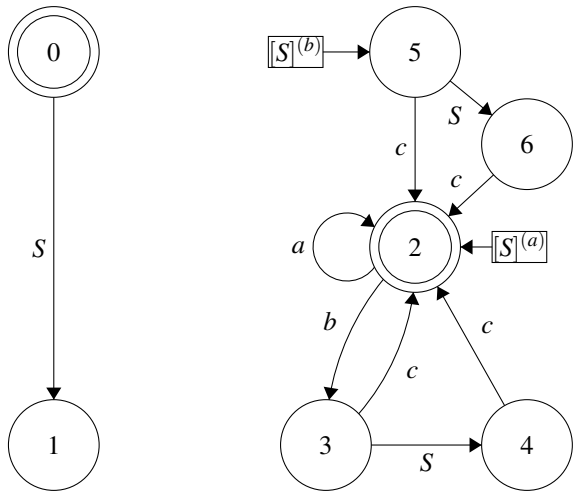
Now we can construct the regular expression as before and end up with $[A]^{(c)} = (aa)(aaa)^*$

If at any point we have an atomic language referencing another that is already fully calculated (i.e. no circular dependency exists), we can simply substitute the entire atomic language and skip the step-by-step substitution performed above.

For the process of calculating atomic languages, we must carefully keep a queue of languages to be computed and perform substitution steps in a round-robin fashion to give all atomic languages equal opportunity to finish calculating. If we don't use round-robin scheduling, the atomic language calculation may run in an infinite loop.

Finite state automaton construction

Now that we have calculated the regular expression (regex) representation of our atomic languages, converting them to a finite state automaton (FSA) to be used in parsing is relatively simple. The atomic languages for the e-rule grammar are: $[S]^{(a)} : (a + bc + bSc)^*$ and $[S]^{(b)} : (c + Sc)(a + bc + bSc)^*$. Its corresponding FSA is:



In our implementation, these finite state automata are represented as a set of states, a set of accepting states, a starting state, a map of edges and a map connecting atomic languages represented as a symbol-terminal pair to a state in the automaton. A state is simply a non-negative integer. The map of edges first maps a state to a second map, this second map maps a symbol to a set of states and rules. The finite state automaton may be nondeterministic, mapping from one state to multiple other states via a single symbol. This implementation can be viewed in *finite_state_automaton.rs*

The regex representation built in the previous stage is separated into ”nodes”; Each section within brackets is such a node. Taking $[S]^{(b)}$ of the e-rule grammar as an example, ($\varepsilon bSc + \varepsilon b\varepsilon c$) is one node and $(a + bSc + b\varepsilon c)^*$ is another. To build the FSA for a grammar, we first create the starting state 0 and connect it to the ending state 1 via an edge labelled with the grammar's start symbol.

We then continue by randomly choosing an atomic language to process. We do this by going through it node-by-node in reverse. For any regex node, we generate an ending state if there is not already one, and we generate a starting state. This pair of starting and ending states is registered using the current regex node as a key, so that we may re-use the starting and ending state when building an FSA for later regex nodes.

Once we have our starting and ending state, we build every alternative word by starting at the starting state, creating a new state and edge for every symbol in the word, until we reach the last symbol of the word. When we are at the last symbol, we create a final edge, connecting the penultimate state to the final state that we generated previously. On the last edge of a word, we attach the derivation rules used to generate that word, these rules are later used to build parse traces while parsing. If during this process, we find that a new word starts with one or more symbols that are identical to a previously processed word, we can simply follow the edges of that previous word until we find a symbol that is different from the previous word. We then continue generating from that state and create states and edges for the rest of the symbols in the current word.

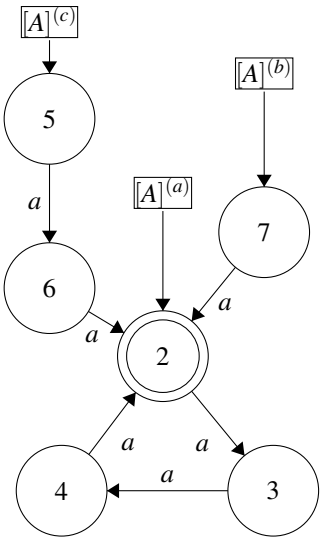
Two factors are complicating this process; Firstly, due to the nature of our parsing algorithm, when a word ends with a nonterminal symbol, this nonterminal symbol cannot be the final edge that connects to the ending state. In that case, we need to pad the word with an epsilon edge, connected to the final state with the derivation rules of that word attached to it. If we do not do this and instead attach the derivation rules to the edge with the nonterminal symbol, the parse rules of that nonterminal will be in the wrong position.

The other factor is that the regular expression we created earlier on may have contained some nulled nonterminals. If it does, the rules applied to null those symbols are included in some position in the regular expression. If this nulling comes before another symbol in the word, we can simply attach this nulling rule to the edge of that following symbol. If the nulled nonterminal is in the last position of the word, we also need to create an epsilon edge and attach the nulling rule to that edge. If multiple nulling rules follow one another, we concatenate the rules and attach the concatenation to the following non-nulling edge.

We go through regex nodes in reverse order, because it allows us to merge parts of the automaton for multiple separate regexes whenever possible. This merging can be observed in the FSA for the e-rule grammar shown before: the ending of $[S]^{(b)}$ is the same as the entirety of $[S]^{(a)}$, meaning that if we connect

the end of the first node of $[S]^{(b)}$ to the start of $[S]^{(a)}$ we do not need to create extra states and edges in the FSA for the rest of $[S]^{(b)}$. This process also works if we build the FSA for $[S]^{(b)}$ before $[S]^{(a)}$.

The concatenation of FSA sections is more pronounced for the indirect left recursive grammar. The atomic languages of nonterminal A are as follows: $[A]^{(a)} : (aaa)*$, $[A]^{(b)} : (a)(aaa)*$, and $[A]^{(c)} : (aa)(aaa)*$. If we were to create separate automata for each atomic language, we would already end up with three separate automata for only nonterminal A. However, because we merge them at appropriate locations, we can represent all three atomic languages with a single connected automaton (Only the atomic languages for nonterminal A shown):



In the above example, merging the automata saves us from having to keep track of six states and six edges. While the size reduction is still quite small for this grammar, this may make a significant difference for larger grammars, especially those with more terminal symbols.

Languages and LanguageList

Relational parsing relies on the fact that when using atomic languages, parsing new symbols devolves into applying a primitive language operation to a previous language and sometimes prepending an atomic language to it. To properly take advantage of this, we build our parsing data structure to reflect this observation; we define every language to be a combination of prefixes to previous languages.

When we do this, the entire data structure may be viewed as a Directed Acyclic Graph (DAG) where every node represents a language. These edges connecting nodes are labelled with states from the FSA. When a symbol is parsed, these labels are used to calculate new edges and form new languages. Edges in this DAG may additionally carry (partial) parse traces and nodes may carry completed parse traces if the language contains the empty string ε.

In our implementation, the entire DAG structure is contained in a data structure called *LanguageList* and languages themselves are represented as a *Language*. Both of these implementations can be found in *language_list.rs*.

The LanguageList is simply a list of languages. Languages in this list refer to other languages not by their position in the list, but by their relative depths in the list. Using a dynamic depth value instead of a fixed position makes memoization easier later on.

If a language contains ε, the fin flag is set to true. If this flag is set, the field completed_parses may contain a set of completed parse traces. The edges field connects a language to one or more previous languages. It is through these edges that languages express anything other than ε.

Representing languages like this has several advantages. If we instead represent these languages as a much simpler list of FSA states to be expanded and shrunk during parsing, and the grammar is ambiguous, we need to duplicate the list of states for every ambiguity encountered during parsing. With the wrong grammar, this list of lists would grow and become unwieldy very quickly. In addition, many lists would be partially identical to one another. This is because when a list of FSA states is populated with some states and then an ambiguity is introduced, all the states that were already in the list are duplicated. In

the LanguageList structure, we avoid this issue as any encountered ambiguity only adds a single edge or a node with a few edges to the list instead of requiring us to duplicate (part of) the list. In this way, the operations necessary to perform on this list become cheaper at the expense of having a more difficult data structure to reason about.

The edges in languages refer to other languages via a depth parameter. This depth parameter points to another language in the LanguageList relative to the current position in the list. So, as an example, if we have a language at position 3 in the list, with an edge of depth 2, then it points to a language at position 1 in the list. Using an absolute pointer also works for parsing, but makes memoizing the parsing process much more difficult later on, this is explained in the later section on memoization.

Parsing

To use the previously outlined data structure for parsing, we implemented one final data structure called *ParseRound* that keeps track of partial results while parsing a single symbol. Every time we parse a new symbol, we create a new ParseRound.

Parsing a string using Relational Parsing relies on two basic operations; *derivative* and *prepend*. Derivative takes a language and a symbol and calculates the Brzozowski derivative of that language by symbols. Prepend prepends an atomic language to a language. In our case, these operations are implemented onto the ParseRound data structure in the form of derive, prep_derive and prepend.

Our parsing algorithm is based on the recognition algorithm outlined by Herman (2020). We start by initialising our LanguageList, retrieving the start state from the FSA and inserting a starting language connected to $\{\varepsilon\}$ via the FSA start state. Then, for every token in the input string, we perform the following steps:

We pop the top-level language (calling it *curr_lang*) off the language list and initialise a new ParseRound. Then we try to calculate a new derivative from *curr_lang* and the current token and save the result in the deriv field of ParseRound. Then, for every nonterminal in the grammar, we check if we have an atomic language for that nonterminal and the current token. If so, we perform a derivative on *curr_lang* by that nonterminal, save it in the *prep_deriv* field of ParseRound and follow it by a call to prepend with the atomic language we found. Finally, the result is saved to the prepend field of ParseRound. After this process, we write the contents of ParseRound to our LanguageList by calling the *register* method.

One noteworthy aspect of derivative and prepend is that if the FSA state attached to an edge is accepting, we need to copy edges from the target language to the current one. Because we use relative depths, we need to sum these depths so that our resulting language refers to the correct target. As an example, say we have a language in position 3 targeting a language in position 1 (with a depth of 2) and a language in position 6 referring to that language in position 3 (depth 3) with an edge that is accepting, then to add the edge referring to language 1, we sum the depths and end up with a target depth of 5.

One other location where we need to do extra depth calculations is in the register method of ParseRound. When we compute a new language via calling prepend and prep_derive, we create either one or two new languages. In the case that we create two new languages and also called derive, we need to add the edges of the derive call to the language computed from prepend. But because we are adding two languages instead of one and because the language added by prep_derive will be added underneath the one added prepend, the edges added by derive will be off by one. Therefore, the depths of all these edges need to be increased by one. If prep_derive did not add a new language, or if derive was the only method called, this does not need to happen and the edges can stay as they were. These depth calculations are necessary because of the choice to use relative pointers; If we used absolute pointers, they could be avoided.

The algorithm described above, called parse, can be found alongside ParseRound and implementations for derivative and prepend in *parse.rs*.

Memoization

To perform memoization on the parsing process, we built three data structures; *MemoBuilder*, which is initialised during parsing and later populates the other two structures, *Memo*, containing the memoization data for a particular language, and *Memoize* that contains all instances of Memo and the languages they belong to.

While parsing a new combination of language and input token, the edges that we create are memoized in MemoBuilder. We do this separately for edges created by calls to prepend, derive, and prep_derive.

During the call of ParseRound's register at the end of a parsing round, the MemoBuilder is converted into a Memo and registered into Memoize.

Now, when we encounter that language and input token combination again, we can perform a lookup in the Memoize structure and apply the modifications stored in the retrieved instance of Memo to our LanguageList instead of calculating new languages via the usual calls to prepend and derivative.

As we discussed before, the use of the depth parameter in edges adds some complexity during parsing when no memoization is available. However, these extra calculations necessary to adjust the depth are not wnecessary anymore when we use a memoized result as we save these edges with the necessary depth adjustments included. If we used absolute depth instead, we would have to perform this type of adjustment every time we refer to the memoization table. This is because the current position of a language is not relevant to the memoization table, if a language occurred in position 1 in the LanguageList and is later encountered again in position 5 with the same input token, we can still look up previous calculations in the memoization table. If we were using absolute depth, we would still need to memoize the relative depth by which these languages refer to each other and calculate a new absolute depth every time we perform a lookup. Herman (2020) asserted that parsing will devolve into mostly table look-ups (in his experiments for up to 99% of parsed tokens), it, therefore, makes sense to optimise for memoization look-ups at the expense of initial calculations by using relative depth pointers instead of absolute pointers.

EVALUATION

To evaluate our implementation of relational parsing, we test its performance both with and without memoization on input strings of increasing length with various grammars. This will tell us how the performance of our implementation changes with inputs of varying lengths and grammar archetypes.

We will now discuss the various grammars we used for this evaluation.

Grammar 1.

$$S \to a$$
$$S \to Sa$$
$$S \to SbSc$$

Grammar 1 is unambiguous, and is defined by Herman (2020). We will refer to it as the "basic grammar". It is left-recursive and contains a structure reminiscent of brackets in programming languages. We see that every time we apply the third derivation rule, adding a 'b' to the terminal string, we must also add a 'c' to the end of that string. This is similar to how brackets work in most programming languages where every opening bracket must be followed by a corresponding closing bracket.

Grammar 2.

$$S \to \varepsilon$$
$$S \to a$$
$$S \to Sa$$
$$S \to SbSc$$

Grammar 2, previously defined as the "e-rule grammar", adds an epsilon rule to the basic grammar, making it ambiguous. Because it accepts all input strings of the previous grammar, we can use it to observe how much the introduction of an ambiguity affects the performance of our relational parsing algorithm.

Grammar 3.

$$S \to aSa$$
$$S \to a$$

Grammar 3, which we refer to as the "odd number of a grammar" is one that is difficult to parse for most parsers even though it is not ambiguous. The reason for this difficulty is that all terminal symbols are identical. To start parsing, a parser has to find the middle 'a'. Parsers that rely on a fixed look-ahead

cannot do this for all inputs, because the parser cannot identify the middle a if the input is more than twice as long as the look-ahead. Backtracking parsers will have to try parse every token as a middle token, backtracking if one isn't in the middle and then trying the next. If the input consists of an even number of tokens, there is no valid parse tree, and a backtracking parser will have to start over at every token in the input before concluding that there is no valid parse tree.

Grammar 4.

$$S \to Sa$$
$$S \to a$$

Grammar 4 is a simple left-recursive grammar. As our "basic grammar" is already left-recursive, this grammar is simply used to compare performance between left and right recursion.

Grammar 5.

$$A \to Ba$$
$$A \to a$$
$$B \to Ab$$
$$B \to b$$

Grammar 5 is an indirectly left-recursive grammar. It is included because while top-down parsers typically already struggle with directly left-recursive grammars, indirectly left-recursive grammars pose even more of a problem. Some parsers, like ANTLR, rewrite grammars under the hood to eliminate left-recursion. While this is possible for both direct and indirect left-recursion, ANTLR only re-writes directly left-recursive grammars and does not re-write indirectly left-recursive grammars.

Grammar 6.

$$S \to aS$$
$$S \to a$$

Grammar 6 is directly right-recursive. While not typically a challenge for top-down or bottom-up parsers, it is still interesting to see how left and right recursion compare. For every rule ending with a nonterminal, we have to introduce an epsilon transition into our FSA.

Grammar 7.

$$A \to aB$$
$$A \to a$$
$$B \to bA$$
$$B \to b$$

Grammar 7 is indirectly right-recursive, added to test whether moving from direct to indirect right-recursion has any effect on performance.

Our relational parsing algorithm was tested on all the above grammars with the criterion benchmark library. For these benchmarks, we generated 10 input strings. The first has an approximate length of 10 characters, increasing by 10 for every subsequent string. The final strings have a length of about 100 characters.

Before every benchmark, we load the inputs into memory, tokenised these inputs and filled the memoization tables. We do this to compare the performance of memoized versus non-memoized parsing with as little overhead as possible and to keep the benchmarks as consistent as possible. We ran these benchmarks on a desktop computer with an Intel i5-10600K clocked at 4.10GHz, running Windows 10.

When performing a benchmark, every experiment is repeated 100 times to reduce the impact of random delays by, for example, OS scheduling.

To show how relational parsing compares to ANTLR, we also attach some minor experiments with ANTLR on some of the grammars.

RESULTS

The results of our experiments are included in the Appendix in Figures 8 until 14. The experiment for the e-rule grammar was cut short as the execution time exploded very quickly, as can be seen in Figure 9. The amount of tokens memoized as well as the percentage of tokens memoized can be seen in Figures 19 until 25.

DISCUSSION

As can be seen in the various figures, our non-memoized code almost always outperformed our memoized code. The only experiment where memoization had a positive effect was when parsing strings for the odd number of a grammar 10. When parsing for the right-recursive grammars, Figures 13 and 14, the performance was almost equal between the memoized and non-memoized code.

Also of note is that the left-recursive grammars perform about 50 times better than the right-recursive grammars. This has to do with two things; First, while right-recursive grammars require the insertion of epsilon transitions into the FSA, left-recursive grammars do not. Second, when parsing right-recursive grammars, the parser is required to call both the derive and prepend methods, whereas for left-recursive grammars the parser can do most parsing by only calling the derive method.

We also see that parsing for the basic grammar is even faster than parsing for the direct left grammar. This is probably a result of handling parsing information. When parsing the direct left grammar, the parser has to add a derivation rule to the parse trace for every symbol parsed, while this happens less often for the basic grammar.

One big stand-out in these experiments is the e-rule grammar in Figure 9. Whereas most experiments ended up taking computation time in the microsecond to millisecond range, even at input lengths of 100 characters, the computations for the e-rule grammar jump up to take several seconds even for inputs only 60 characters long. Because of the structure and ambiguity of this grammar, the amount of possible parse traces rise very quickly. Our implementation seems to handle a large amount of parse traces very poorly, causing a large jump in computation time from milliseconds to seconds between inputs of lengths 52 and 61.

Comparing this to ANTLR's performance on the 61-length input for the e-rule grammar, seen in Figure 15, we see that the input is parsed in 177 milliseconds. This measurement is extremely unfair to ANTLR, however, as this also includes the time for reading the input from the disk and lexing the input into a token stream, all of which was already taken care of in our relational parsing benchmarks.

So while this comparison is not fair, it still shows that for this grammar and input, ANTLR is able to parse it much faster than our implementation of relational parsing. However, ANTLR also only computes one of the many possible parse trees for this input, whereas we try to compute all possible parse trees.

Some of the other grammars that we used to test relational parsing are not usable with ANTLR. The odd number of a grammar is an interesting case; as seen in Figure 16, ANTLR can parse the strings "a" and "aaa". For the string "aa" ANTLR gives a parse tree, but this string has no correct parse tree for this grammar. "aaaa" fails as expected, however, the string "aaaaa" does have a parse tree but ANTLR is unable to find it. This pattern of not being able to find a parse tree continues for larger inputs, even though a parse tree does exist for every string with an odd number of 'a' characters.

We believe the explanation to be that even though the grammar itself is not ambiguous, the parser encounters an ambiguous state during parsing. When ANTLR encounters an ambiguity, it resolves the situation by choosing one possible parse tree and continue parsing only for that tree. In the case of the odd number of a grammar, when the input is longer than three characters, ANTLR encounters an ambiguous state and makes a decision that will always lead to an incomplete parse tree even if the input has a valid parse tree. It is this commitment to non-ambiguity that causes ANTLR to fail to parse this grammar.

As seen in Figure 17, ANTLR refuses to compile grammars containing indirect left recursion. Due to the nature of ALL(*), it is unable to parse strings for any grammar with left recursion, but ANTLR rewrites grammars containing direct left recursion to eliminate left recursion on the grammar level. For indirect left recursion, this is not done and as such, ANTLR spits out an error.

Interestingly, for the direct right-recursive and indirect right-recursive grammars, for which parse attempts can be seen in Figures 18 and **??**, ANTLR does compile right-recursive grammars but is unable to parse our example grammars, always seeming to expect another character to follow. While ANTLR can parse some right-recursive grammars, it fails to parse for these toy right-recursive grammars.

A previous version of our parser was able to parse the indirectly left-recursive example grammar, but failed for some other indirectly left-recursive grammars. The results for our example indirectly left-recursive grammar using that older version are still included. The current version of our parser rejects indirectly left-recursive grammars. There is space to implement support for such grammars using relational parsing, however our support for these grammars was buggy and has therefore been scrapped.

There is another class of grammars which our implementation fails to parse:

$$S \to A$$
$$S \to B$$
$$A \to ax$$
$$B \to ay$$

Any grammar of a form similar to the grammar above will not function as expected. Whenever our implementation encounters a situation where it can perform a derivation by at least two nonterminals and attempts to prepend two or more atomic languages in one parsing step, it produces incorrect results. As of now, our implementation is only able to call prep-derive and prepend once per parsing step. This is not a limitation of relational parsing, but instead a limitation of our implementation.

RELATED WORKS

Context-free parsing

Context-free parsing is the most commonly used technique for parsing programming languages using generated parsers. Combined with pre- and post-processing techniques, context-free grammars are sufficient to specify commonly used programming languages. Parsing at this level also carries good performance-guarantees: Binary Right Nulled GLR (BRNGLR) parsers have worst-case cubic run time (with relation to input length) for all grammars with linear performance for LR(1) grammars Scott and Economopoulus (2007). Since BRNGLR can parse any context-free grammar, this performance can be taken as the worst-case performance for any context-free grammar with other algorithms carrying potential performance improvements for particular context-free grammars or all context-free grammars.

Parsing algorithms can be divided into three main categories: top-down, bottom-up and others. In top-down parsing, we start parsing with the grammar's start symbol and try to produce the input by matching what we produce to the input string. Bottom-up parsers, in contrast, start with the input and try to reduce it to the start symbol by applying production rules in reverse. As mentioned before, there exist other context-free parsing algorithms which are neither top-down or bottom-up. One example of these is the so-called "intersection algorithm", which intersects a context-free grammar with the finite-state-automaton of a given input and produces a parse tree by cleaning up this intersection. Many of these various parsing techniques are described in detail in Parsing Techniques Grune and Jacobs (2008).

Parallel parsing

Various approaches exist to boost the performance of parsing algorithms. Some of these approaches result in the invention of entirely new algorithms, others improve existing algorithms in some way. One technique that may allow us to improve the performance of existing and future algorithms is that of parallelisation. Alblas et al. (1994), Barenghi et al. (2015), van Marcellus Paulus Lohuizen (2001).

State-of-the-art algorithms

GLL

Top down parsers are popular because they follow the structure of the grammar they parse, making them easy to write and debug. Recursive Descent (RD) is one such algorithm that was very popular, however the class of grammars that it parses is very limited. This limitation could be alleviated by allowing it to backtrack, but this could result in explosive run-times. The newer GLL algorithm also closely follows the structure of the grammar and uses RNGLR-like machinery to allow it to parse every context-free grammar. Unlike RD, GLL is worst-case cubic. Scott and Johnstone (2010), Scott and Johnstone (2013). The parser combinator FUN-GLL is based on GLL. Whereas conventional parser combinators inherit the

drawbacks of RD, FUN-GLL overcomes these and similarly delivers a wors-case cubic run-time. van Binsbergen et al. (2018), van Binsbergen et al. (2020).

SGLR

Most parsing algorithms expect a stream of tokens during parsing. The algorithms that produce these tokens from the source texts are called scanners. Scannerless parsing algorithms allow us to combine the lexical and context-free syntax into a single grammar and allow us to discard the lexical analysis phase that runs before parsing can start. However, combining lexical and context-free syntax can yield unwieldy grammars. (Visser, 1997) solves this problem by introducing grammar normalisation. He goes on by introducing scannerless-GLR (SGLR), building upon scannerless-LR(1) and GLR. SGLR was presented to a wider audience in 2002 van den Brand et al. (2002), included in integrated development environments to assist in language development Kats et al. (2009) and later improved with the introduction of incremental SGLR Sijm (2019).

Iguana

Iguana is another general parser generator introduced by (Afroozeh and Izmaylova, 2016) and runs nearly linearly on programming language grammars and is worst-case cubic. It is built upon GLL, however instead of producing all parse trees like GLL and relational parsing, it produces a single tree. When using Iguana, in addition to defining a context-free grammar, the user also defines disambiguation rules to specify (un)desired parse trees where necessary. These disambiguation rules are specified with the use of arbitrary computation, variable binding and constraints. These features extend the context-free grammar into what Afroozeh calls Data dependent grammars. Where relational parsing may attempt to produce an astronomical amount of parse trees for a highly ambiguous grammar, Iguana produces only the parse tree the user wants to be produced.

ALL(*)

Adaptive LL(*) is described at-length in subsection "ANTLR and ALL(*)". It is worst-case $O(n^4)$ but performs linearly on most grammars used in practice, thereby outperforming GLL and GLR by orders of magnitude Parr et al. (2014). ANTLR4 is the parser generator for ALL(*) and is currently the industry standard for parser generators. Parr (2022). Unlike the generalised parsing algorithms, ALL(*) is not able to parse every grammar. Most notably, left-recursion is dealt with by re-writing the grammar and indirect left recursion is not supported. It is deterministic, producing only one of the possible parse trees of an ambiguous grammar.

Packrat

Packrat parsers guarantee linear parse times, but cannot parse every grammar. Any LL(k) or LR(k) grammar can be parsed by a packrat parser, as well as "many languages that conventional linear-time algorithms do not support" Ford (2002). Left-recursion is dealt with by re-writing the grammar. The grammar $S \rightarrow aSa|a$ is an example Packrat will not parse. Furthermore, the algorithm is deterministic, only producing one result, even if there are other possible parse trees. Like SGLR, there is an incremental version of packrat Dubroy and Warth (2017) and there is a version for parsing *ParsingExpressionGrammars* (PEGs) Blaudeau and Shankar (2020).

Derivatives

The derivative of a language was first introduced by (Brzozowski, 1964) in the context of regular languages. It is an operator that produces the language that remains after a string of symbols s. If the language is called L, this remainder can be called the $derivative of L by s$. Derivatives were introduced as a tool for analysing the properties of regular expressions but proved to be useful for much more than that. (Might et al., 2011) introduced a rudimentary technique for parsing context-free grammars using derivatives. (Adams et al., 2016) later proved that Might's technique is worst-case cubic and that with a few modifications its performance could be sped up by a factor of 951, making it "on par with other parsing frameworks".

Relational parsing

This thesis is built upon the paper on relational parsing by Herman (2020). It introduces the concept of relational parsing which reduces parsing context free grammars to that of parsing regular languages by leveraging derivatives. It then improves performance by introducing a non-cyclic data structure which allows for memoizing much of the parsing effort.

Pushdown Automata

Herman cites several works as being relevant to relational parsing: (Lang, 1974) introduced the technique of effective simultaneous simulation of possible runs of a pushdown automaton (PDA). (Tomita, 1985b) (Tomita, 1988) introduced the graph-structured stack as a data structure for capturing its nondeterminism. This innovation led to the developments of GLR Kats et al. (2009), GLL Scott and Johnstone (2010) and GLC Nederhof (1993) Moore (2004).

Left Corner parsing

GLC is of particular interest: It combines top-down and bottom-up parsing with the left corner relation, containing a pair of nonterminals whenever one can appear leftmost in some derivation of the other. The atomic languages in relational parsing have Nondeterministic Finite Automata (NFA) states indexed in pairs of symbols related in exactly the same way. The authors of GLC make their algorithm more efficient by heuristically grouping some of the cases Nederhof (1993) Moore (2004). Relational parsing may be able to use these optimisations in the same way.

Stack activity

According to Herman, the primary source of inefficiency in generalised parsers is the handling of large graph-like data structures, offering poor cache locality. Scott and Johnstone (2005) introduced an algorithm which limits the size of these graphs by handling most of the grammar analysis with finite automata. Johnstone and Scott (2007) shows that appropriate PDAs can be constructed in multiple ways and that the best one for a given parser can be selected by profiling the parser on sample inputs. Relational parsing uses a different strategy to limit stack activity. The strategy by Johnstone limits the depth of multiple stacks, while relational parsing limits the width of stacks to a single stack.

Derivative parsers

Herman also mentions derivative parsers by (Might et al., 2011) and (Henriksen et al., 2019). He explains that the difference between these algorithms and relational parsing is in the way intermediate languages are represented. Parsing-With-Derivatives and Derivative Grammars both produce intermediate grammars that are context-free and therefore require the algorithms to handle cyclic data structures. These cyclic data structures prevent context-free memoization, which is why Herman argues that his graph structure is advantageous.

Memoization

When a program repeatedly performs calculations with the same input to produce the same output, this program may benefit from memoization. Memoization is the technique of storing the results of previous calculations so that the next time these results are requested, the program can look the results up in a cache, which may be much quicker than performing the original calculation. Memoization can, for example, be used to greatly speed up a recursive algorithm calculating fibonacci numbers Norvig (1991). As demonstrated by (Hall and McNamee, 1997), memoization can even be applied automatically to a variety of applications.

Memoization in Parsing

After demonstrating how to optimise recursive fibonacci using memoization, (Norvig, 1991) shows how to apply memoization to Earley parsing. While Norvig's technique of memoization broke support for left recursion, (Johnson, 1995) was able to develop a top-down memoizing parser that handles left-recursion.

(Becket and Somogyi, 2008) shows how to build packrat parsers from Definite Clause Grammars by memoizing. However, this nearly always results in a performance loss if everything is memoized. This loss stems from the overhead of accessing memoization tables which is larger than the performance hit of performing calculations from scratch. However, if memoization is carefully applied to a few parts of the parser, memoization may provide a gain in performance. (Kuramitsu, 2015) introduced Elastic Packrat, which adjusts the memoizing mechanism to limit the cost of memoization and improve its performance in relation to plain packrat parsing.

Replications

(Carver et al., 2014) state that performing replications is not standard in the field of computer science while generally being considered a necessary cornerstone to scientific study. However, there are many

open issues to address before the replication process can be fully formalised in software engineering research.

(Juristo and Gómez, 2012) discuss the concept of replication in the field of software engineering and identify the types of replications that are feasible to run in this discipline.

(Santos et al., 2021) show that the direct comparison of statistical data is not suitable for verifying the results of previous software experiments. Instead, the results of replication studies should be analysed using analytical methods such as meta-analysis. They argue that the results in baseline experiments should not need to be reproduced, but should be regarded as small pieces of evidence in a larger picture.

(Juristo and Vegas, 2009), (Juristo and Vegas, 2011) confirm that exact replications are infeasible in software engineering due to the complexity inherent in software development. They propose a process allowing researchers to generate new knowledge using non-exact replications.

(Kitchenham, 2008) argues that replications are best performed independently. Dependent replications conflict with the underlying assumptions of meta-analysis, which may be our best tool to compare experimental results in software engineering. Furthermore, proposes that the purpose of performing software engineering experiments is to help improve industry practice. When the results of experiments are consistent even when the subjects, materials and setting change, then these results become much more applicable to the industry setting.

Parser evaluation

There are multiple metrics used to test parsers: Parser speed, memory consumption and parser correctness.

Parseval is a popular evaluation technique in the field of natural language processing and was introduced in 1991. It works by measuring the degree to which parser outputs match analyses assigned to sentences in a manually annotated test corpus. These annotations are relatively simple, consisting of only brackets. However, this simplicity results in some limitations: The level of detail in analyses can be low for some corpuses, making it more difficult to distinguish between a good and bad parser. Furthermore, there can be incompatibilities between parsing systems and parseval.

In response, (Carroll and Briscoe, 2001) and (Carroll et al., 2003) introduce a different technique called grammatical relation annotation. In this style of annotation, sentences are annotated with grammatical relations. These relations give a more fine-grained evaluation compared to parseval, allowing researchers to pinpoint with greater accuracy where the problems with a given parser lie.

(Zaytsev, 2018) describes a testing methodology used to verify programming language parser correctness and performance. This methodology was used throughout the development of the compiler used in the described case study. They argue that implementing some sort of testing framework during the construction of a parser or compiler aids in the development process by protecting against regressions and indicating the progress of the implementation effort and list some types of tests that may be of particular use.

REFLECTION AND FUTURE WORK

Recreating relational parsing proved much more difficult than at first anticipated. The description provided by Herman (2020) is very abstract, and sparse with details and examples. It took a lot of trial and error to arrive at a result that resembles Herman's description. As our intention was to independently recreate relational parsing, we did not contact Herman, nor did we look at his source code. While this decision made the process more difficult than it otherwise could have been, it did help to identify what parts of relational parsing may be difficult to understand and implement. In the end, multiple unsolved challenges impacted the performance of our algorithm.

Firstly, the way we keep track of generated languages during parsing and parse traces probably slows down performance more than is necessary. Whenever the parser encounters an ambiguity, all active parse traces are duplicated so that it can create variants for every option in the ambiguity. Therefore, a highly ambiguous grammar will force the algorithm to do a lot of bookkeeping, severely slowing down the parser.

One possible solution is to alter the LanguageList data structure so that it no longer operates like a stack but more like a fixed list. When implementing this change, it is possible to keep partial parsing information in older languages, making it so that no one language has to keep track of a majority of the parsing information. The full parse traces could then be retrieved by visiting every language used during parsing and concatenating any partial parse traces. The drawback of this approach is that every language

generated during parsing has to be kept around, meaning that the list of languages grows linearly with the length of your parsing input and that there is an additional cost for finishing the parse traces.

The second challenge we encountered but did not fully overcome is that of memoization. In many of our test cases, recalling from memoization was slower than computing from scratch. This issue may have arisen because our test cases are not suitable for memoization, maybe because the grammars we tested are too small. Another option is that memoization is simply not worth it for relational parsing. However, as memoization yielded massive positive results in Herman (2020)'s experiments, we find it more likely that our implementation is limiting the usefulness of memoization.

When our algorithm is applying some memoization data, it accesses four or five different hashmaps. We believe this is the biggest source of our inefficiency. It should be possible to merge these hashmaps into one if we build a data structure that can unify the memoization data scattered across these several hashmaps. This would then greatly reduce the amount of hashing and lookup operations our algorithm performs during memoization and should therefore increase performance.

The final challenge where the need for improvement is obvious is the LanguageList data structure or the algorithms surrounding it. As discussed in the Results section, our algorithm is unable to correctly call prep-derive and prepend more than once during a parsing step. This is because of an assumption made early on in the project when attempting to reverse-engineer Herman's data structure for storing languages. The mismatch occurs because languages created by prep-derive need to be separate from each other but languages created while languages created get merged. Keeping multiple separate prep-derive languages is currently not possible.

To solve this, the ParseRound and LanguageList data structures would need to be rewritten. One possible solution for the LanguageList is that we completely do away with storing a single list of languages. Instead we could use some recursive data structure where every language exists on its own and new languages contain a pointer to an old one. We then keep track of a list of current languages and mutate these to produce new languages.

This would allow us to have multiple languages side-by-side with the full language of symbols accepted by the grammar at that point being the union of these current languages. The advantage here is that this solution is much more flexible, but may be more difficult to reason about.

With this change, it may also be possible to do away with the ParseRound data structure.

To add support for indirectly left-recursive grammars, we need an algorithm which can recursively traverse atomic languages and keep track of which ones it has already visited. This algorithm has not been implemented due to time constraints. When a user attempts to input an indirectly left-recursive grammar, they receive a message that these grammars are not supported at the moment.

CONCLUSION

We built a re-implementation of Relational parsing to investigate the intricacies of the parsing algorithm and to test its effectiveness at parsing. We did not manage to replicate the algorithm in its full scope, but did arrive at a result that resembles it in most respects. The limitations of our implementation seem to stem from our implementation rather than being a limitation inherent in Herman's algorithm.

In our experiments, memoization slows down parsing for most grammars, but does produce correct results. This shows that it could be useful if its performance were improved. Furthermore, we showed that relational parsing is able to parse grammars that most other parsers find tricky. However, our re-implementation is not yet a generalised parser.

There is still room for improvement in the world of context-free parser generators. ANTLR, being the current industry standard, is quick at what it does. However, it provides only one of the potentially many possible parse trees for ambiguous grammars. Additionally, it may silently fail to parse some grammars by producing incorrect parse trees or rejecting input that should produce correct parse trees.

A stable, user-friendly implementation of relational parsing may solve these problems for some use cases.

REFERENCES

Adams, M. D., Hollenbeck, C., and Might, M. (2016). On the complexity and performance of parsing with derivatives. *SIGPLAN Not.*, 51(6):224–236.

Afroozeh, A. and Izmaylova, A. (2016). Iguana: a practical data-dependent parsing framework.

Alblas, H., op den Akker, R., Luttighuis, P. O., and Sikkel, K. (1994). A bibliography on parallel parsing. *SIGPLAN Not.*, 29(1):54–65.

Aycock, J. and Horspool, N. (2001). Directly-executable earley parsing. In Wilhelm, R., editor, *Compiler Construction*, pages 229–243, Berlin, Heidelberg. Springer Berlin Heidelberg.

Barenghi, A., Crespi Reghizzi, S., Mandrioli, D., Panella, F., and Pradella, M. (2015). Parallel parsing made practical. *Science of Computer Programming*, 112:195–226.

Becket, R. and Somogyi, Z. (2008). Dcgs + memoing = packrat parsing but is it worth it? In Hudak, P. and Warren, D. S., editors, *Practical Aspects of Declarative Languages*, pages 182–196, Berlin, Heidelberg. Springer Berlin Heidelberg.

Blaudeau, C. and Shankar, N. (2020). A verified packrat parser interpreter for parsing expression grammars. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 3–17, New York, NY, USA. Association for Computing Machinery.

Brzozowski, J. A. (1964). Derivatives of regular expressions. *J. ACM*, 11(4):481–494.

Carroll, J. and Briscoe, T. (2001). Parser evaluation: a survey and a new proposal.

Carroll, J., Minnen, G., and Briscoe, T. (2003). *Parser Evaluation*, pages 299–316. Springer Netherlands, Dordrecht.

Carver, J., Juristo, N., Baldassarre, M., and Vegas, S. (2014). Replications of software engineering experiments. *Empirical Software Engineering*.

Chomsky, N. (1956). Three models for the description of language. *IRE Trans. Inf. Theory*, 2:113–124.

Cooper, K. D. and Torczon, L. (2012). Chapter 4 - context-sensitive analysis. In Cooper, K. D. and Torczon, L., editors, *Engineering a Compiler (Second Edition)*, pages 161–219. Morgan Kaufmann, Boston, second edition edition.

DeRemer, F. (1969). Practical translators for lr(k) languages.

Dubroy, P. and Warth, A. (2017). Incremental packrat parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 14–25, New York, NY, USA. Association for Computing Machinery.

Earley, J. (1970). An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

Ford, B. (2002). Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. *SIGPLAN Not.*, 37(9):36–47.

Grune, D. and Jacobs, C. (2008). *Parsing Techniques*. Springer New York.

Grzegorz, H. (2023). Relational Parsing.

Hall, M. R. and McNamee, J. P. (1997). Improving software performance with automatic memoization.

Henriksen, I., Bilardi, G., and Pingali, K. (2019). Derivative grammars: A symbolic approach to parsing with derivatives. *Proc. ACM Program. Lang.*, 3(OOPSLA).

Herman, G. (2020). Faster general parsing through context-free memoization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1022–1035, New York, NY, USA. Association for Computing Machinery.

Johnson, M. (1995). Memoization in top-down parsing. *Comput. Linguist.*, 21(3):405–417.

Johnstone, A. and Scott, E. (2007). Automatic recursion engineering of reduction incorporated parsers. *Science of Computer Programming*, 68(2):95–110. Special Issue on ETAPS 2005 Workshop on Language Descriptions, Tools, and Applications (LDTA '05).

Juristo, N. and Gómez, O. S. (2012). *Replication of Software Engineering Experiments*, pages 60–88. Springer Berlin Heidelberg, Berlin, Heidelberg.

Juristo, N. and Vegas, S. (2009). Using differences among replications of software engineering experiments to gain knowledge. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 356–366.

Juristo, N. and Vegas, S. (2011). The role of non-exact replications in software engineering experiments. *Empirical Software Engineering*.

Kats, L. C., de Jonge, M., Nilsson-Nyman, E., and Visser, E. (2009). Providing rapid feedback in generated modular language environments: Adding error recovery to scannerless generalized-lr parsing. *SIGPLAN Not.*, 44(10):445–464.

Kitchenham, B. (2008). The role of replications in empirical software engineering-a word of warning. *Empirical Software Engineering*.

Kuramitsu, K. (2015). Packrat parsing with elastic sliding window. *Journal of Information Processing*, 23(4):505–512.

Lang, B. (1974). Deterministic techniques for efficient non-deterministic parsers. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, page 255–269, Berlin, Heidelberg. Springer-Verlag.

Laurent, N. and Mens, K. (2016). Taming context-sensitive languages with principled stateful parsing. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, page 15–27, New York, NY, USA. Association for Computing Machinery.

Lewis, P. M. and Stearns, R. E. (1968). Syntax-directed transduction. *J. ACM*, 15(3):465–488.

Might, M., Darais, D., and Spiewak, D. (2011). Parsing with derivatives: A functional pearl. *SIGPLAN Not.*, 46(9):189–195.

Mitchell, D. C. (1994). Sentence parsing. In *Handbook of psycholinguistics*, chapter 11, pages 375 – 409. Academic Press.

Moore, R. C. (2004). *Improved Left-Corner Chart Parsing for Large Context-Free Grammars*, page 185–201. Kluwer Academic Publishers, USA.

Nederhof, M.-J. (1993). Generalized left-corner parsing. In *Proceedings of the Sixth Conference on European Chapter of the Association for Computational Linguistics*, EACL '93, page 305–314, USA. Association for Computational Linguistics.

Norvig, P. (1991). Techniques for automatic memoization with applications to context-free parsing. *Comput. Linguist.*, 17(1):91–98.

Parr, T. (2022). About the antlr parser generator. https://www.antlr.org/about.html. Accessed: 2022-06-23.

Parr, T. and Fisher, K. (2011). Ll(*): The foundation of the antlr parser generator. *SIGPLAN Not.*, 46(6):425–436.

Parr, T., Harwell, S., and Fisher, K. (2014). Adaptive ll(*) parsing: The power of dynamic analysis. *SIGPLAN Not.*, 49(10):579–598.

Santos, A., Vegas, S., Oivo, M., and Juristo, N. (2021). Comparing the results of replications in software engineering. *Empirical Software Engineering*.

Scott, E. and Johnstone, A. (2005). Generalized Bottom Up Parsers With Reduced Stack Activity. *The Computer Journal*, 48(5):565–587.

Scott, E. and Johnstone, A. (2010). Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

Scott, E. and Johnstone, A. (2013). Gll parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844. Special section on Language Descriptions Tools and Applications (LDTA'08 & '09) & Special section on Software Engineering Aspects of Ubiquitous Computing and Ambient Intelligence (UCAmI 2011).

Scott, J. and Economopoulus (2007). Brnglr: a cubic tomita-style glr parsing algorithm. *Acta Informatica*, 44:427–461.

Shepperd, M. (2018). Replication studies considered harmful. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, page 73–76, New York, NY, USA. Association for Computing Machinery.

Shi, Z. (2021). Chapter 6 - language cognition. In Shi, Z., editor, *Intelligence Science*, pages 215–266. Elsevier.

Sijm, M. P. (2019). Incremental scannerless generalized lr parsing. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2019, page 54–56, New York, NY, USA. Association for Computing Machinery.

Tomita, M. (1985a). *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, USA.

Tomita, M. (1985b). *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, USA.

Tomita, M. (1988). Graph-structured stack and natural language parsing. In *Proceedings of the 26th Annual Meeting on Association for Computational Linguistics*, ACL '88, page 249–257, USA. Association for Computational Linguistics.

van Binsbergen, L. T., Scott, E., and Johnstone, A. (2018). Gll parsing with flexible combinators. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2018, page 16–28, New York, NY, USA. Association for Computing Machinery.

van Binsbergen, L. T., Scott, E., and Johnstone, A. (2020). Purely functional gll parsing. *Journal of Computer Languages*, 58:100945.

van den Brand, M. G. J., Scheerder, J., Vinju, J. J., and Visser, E. (2002). Disambiguation filters for scannerless generalized lr parsers. In Horspool, R. N., editor, *Compiler Construction*, pages 143–158, Berlin, Heidelberg. Springer Berlin Heidelberg.

van Marcellus Paulus Lohuizen (2001). Parallel natural language parsing: From analysis to speedup.

Visser, E. (1997). Scannerless generalized-lr parsing.

Zaytsev, V. (2018). An industrial case study in compiler testing (tool demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2018, page 97–102, New York, NY, USA. Association for Computing Machinery.

APPENDIX

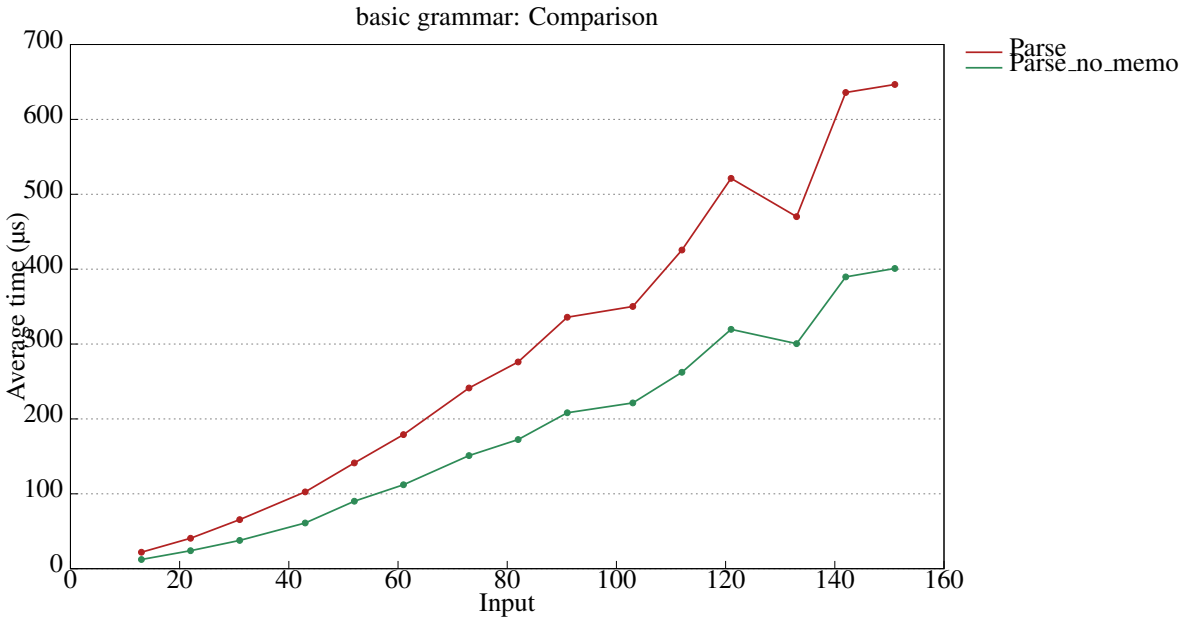


Figure 8. Comparison between memoized and non-memoized parsing for basic grammar

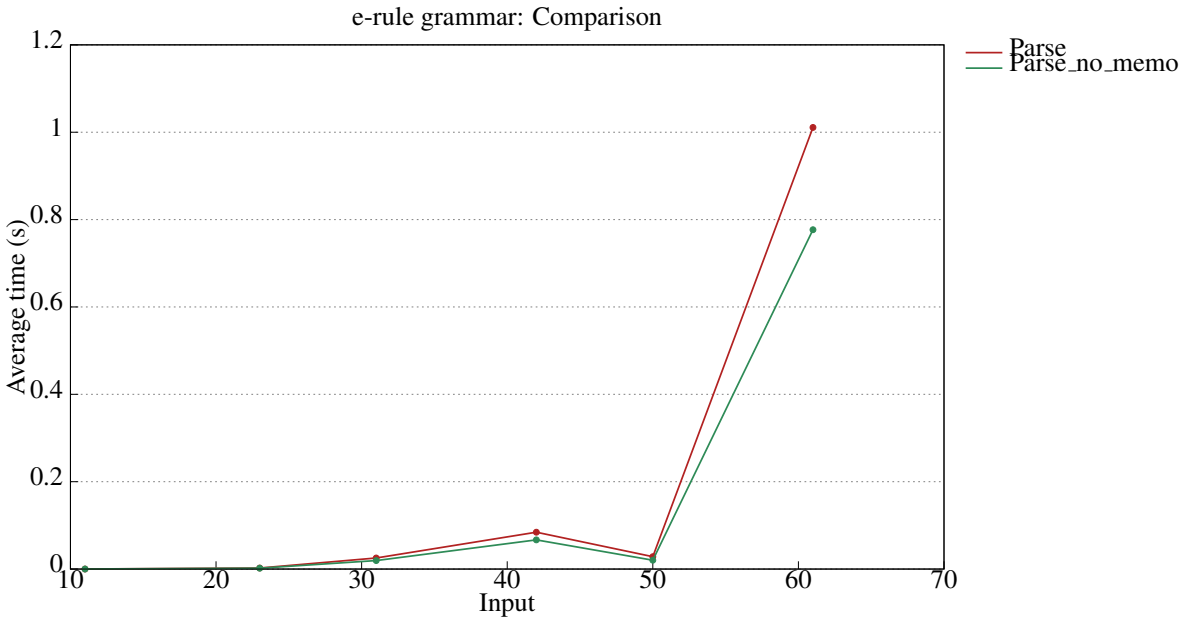


Figure 9. Comparison between memoized and non-memoized parsing for e-rule grammar

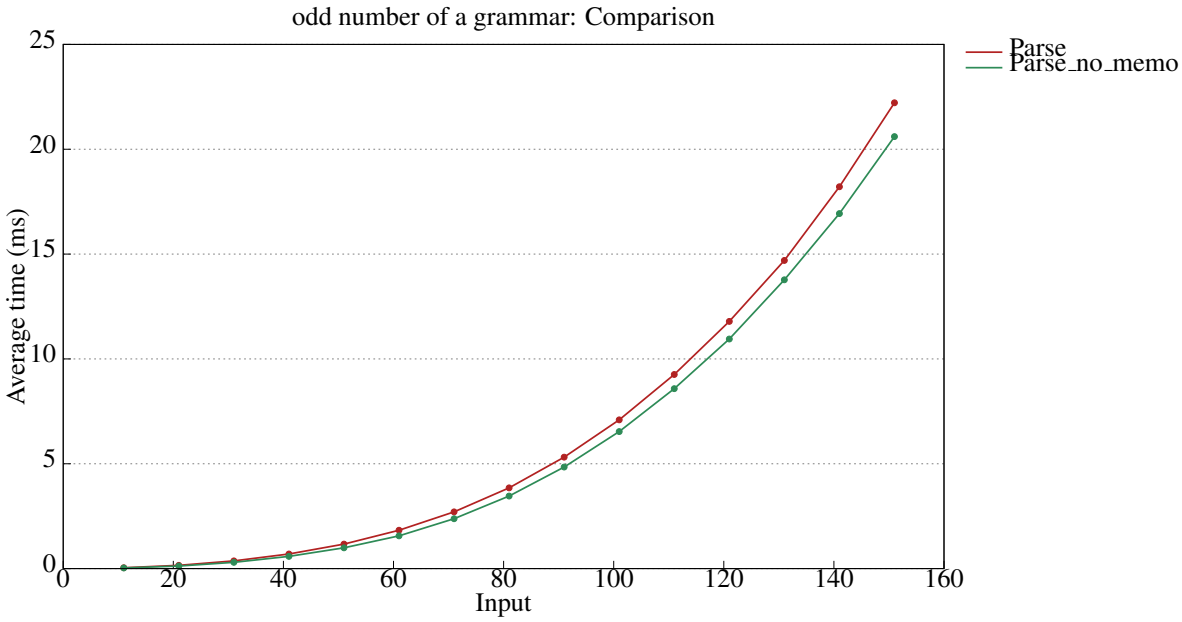


Figure 10. Comparison between memoized and non-memoized parsing for odd number of a grammar

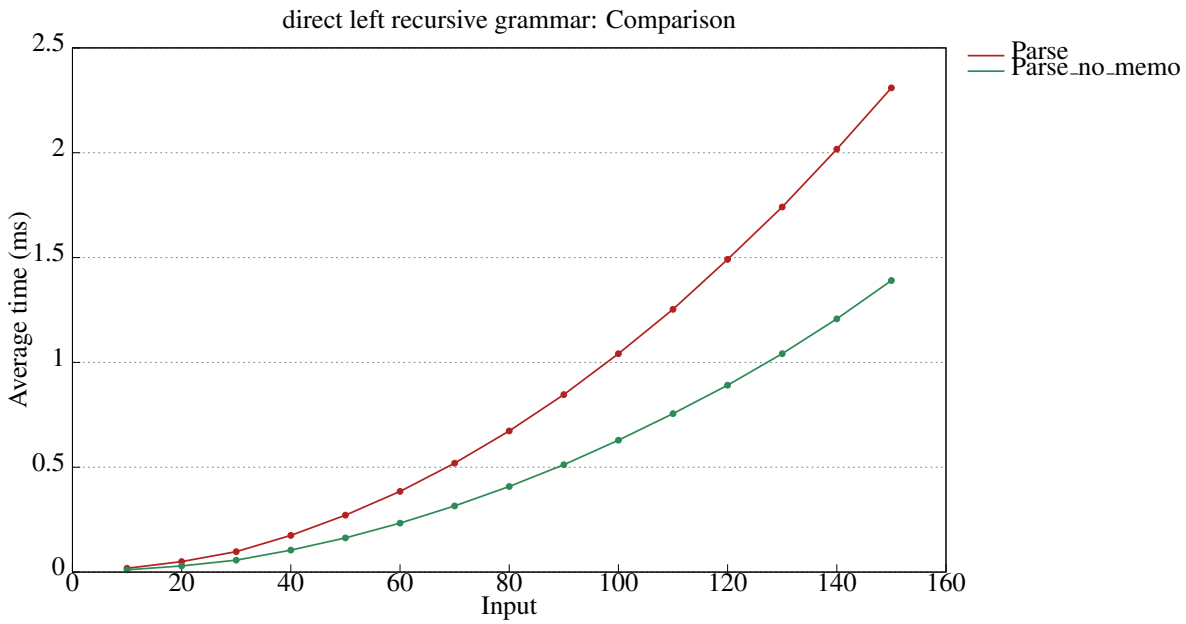


Figure 11. Comparison between memoized and non-memoized parsing for direct left-recursive grammar

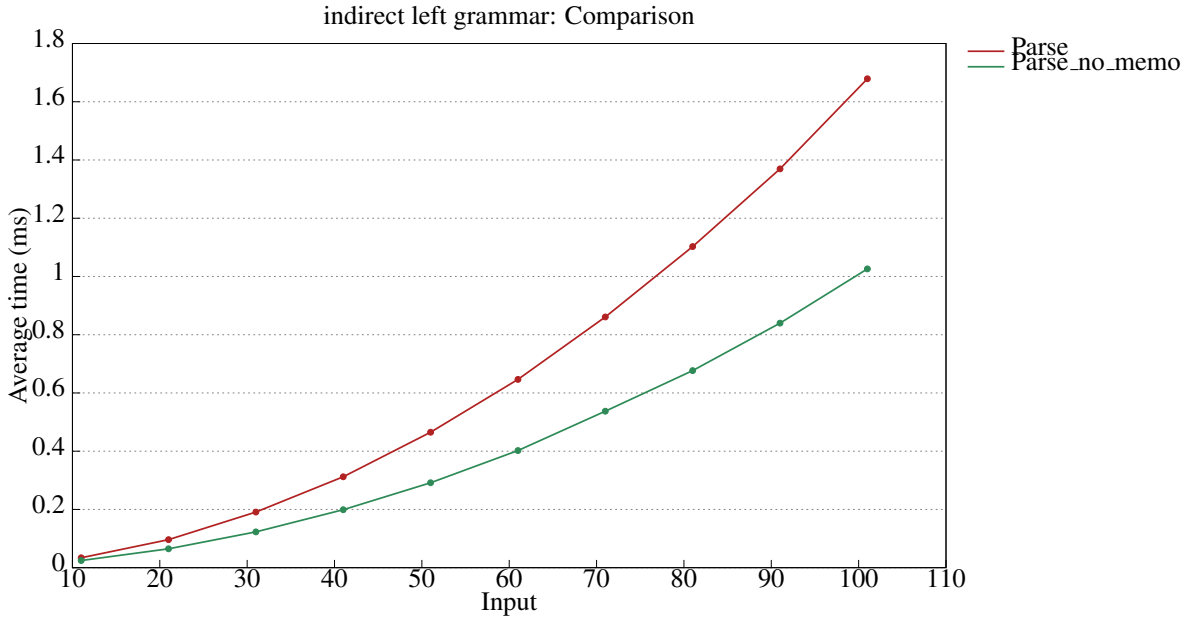


Figure 12. Comparison between memoized and non-memoized parsing for indirect left-recursive grammar (older version of application)

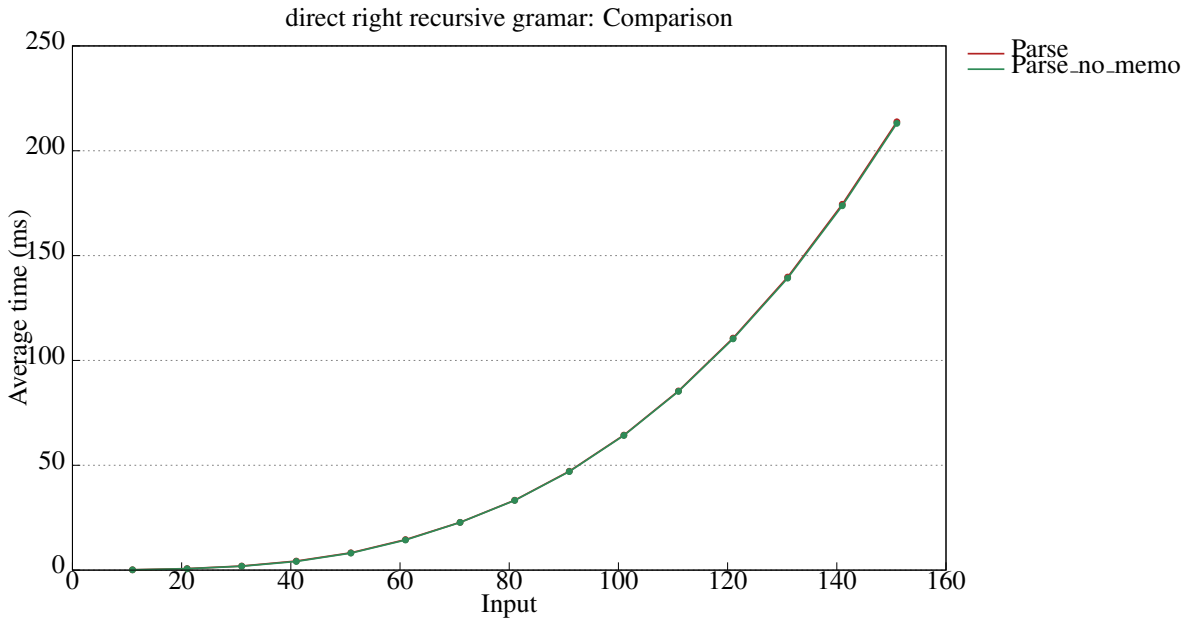


Figure 13. Comparison between memoized and non-memoized parsing for direct right-recursive grammar

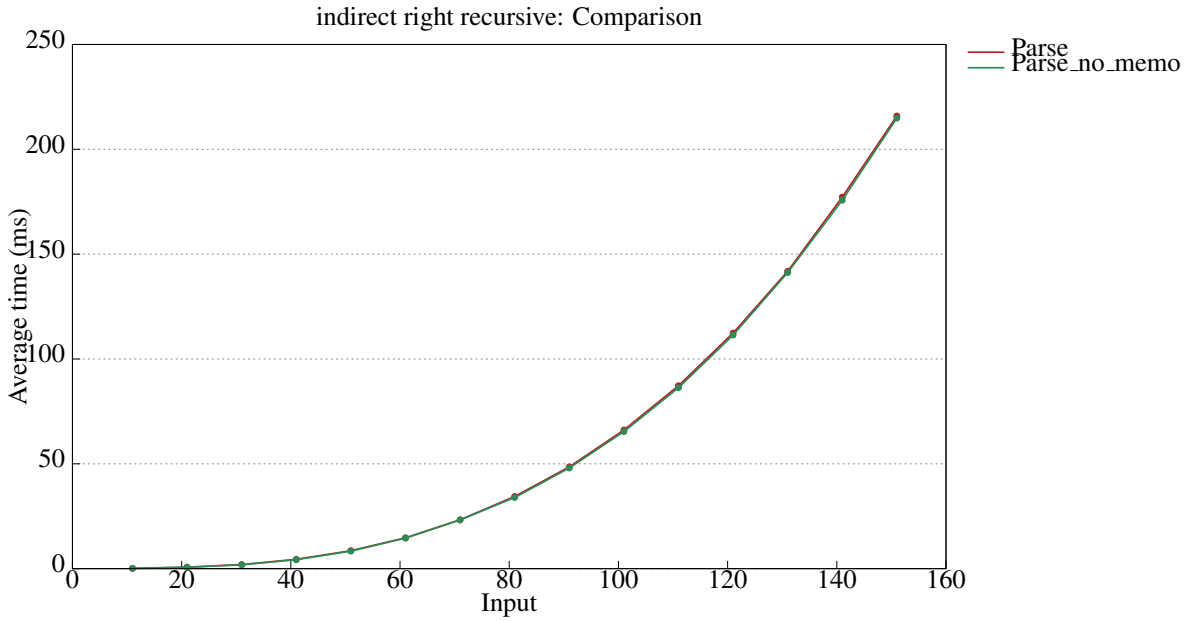


Figure 14. Comparison between memoized and non-memoized parsing for indirect right-recursive grammar

```
C:\Users\max-h\Desktop\ANTLR\Erule>java -cp .;C:\Users\max-h\Desktop\ANTLR\antlr-4.11.1-complete.jar org.antlr.v4.gui.Te
stRig Erule s -tree input.txt
(s (s (s (s (s a) b s c) b (s (s s b (s a) c) b (s (s a) b (s (s (s (s (s a) b (s (s (s a) b (s a) c) b s c) c) b (s a)
c) b (s a) c) b (s s b (s (s s b (s (s (s a) b (s a) c) b s c) c) b (s (s a) b (s (s a) b (s s b (s s b (s (s a) b (s a)
 c) c) c) c) c) c) c) c) c) c) b s c) b (s a) c)
PS C:\Users\max-h\Desktop\ANTLR\Erule> Measure-Command {grun Erule s -tree input.txt}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 177
Ticks             : 1778868
TotalDays         : 2.058875E-06
TotalHours        : 4.9413E-05
TotalMinutes      : 0.00296478
TotalSeconds      : 0.1778868
TotalMilliseconds : 177.8868
```

Figure 15. ANTLR lex and parse time of 61 character test input for e-rule grammar

```
C:\Users\max-h\Desktop\ANTLR\Odd_nr_a>java -cp .;C:\Users\max-h\Desktop\A
NTLR\antlr-4.11.1-complete.jar org.antlr.v4.gui.TestRig Odd_nr_a s -tree
a
^Z
(s (a a))
PS C:\Users\max-h\Desktop\ANTLR\Odd_nr_a> grun Odd_nr_a s -tree

C:\Users\max-h\Desktop\ANTLR\Odd_nr_a>java -cp .;C:\Users\max-h\Desktop\A
NTLR\antlr-4.11.1-complete.jar org.antlr.v4.gui.TestRig Odd_nr_a s -tree
aa
^Z
(s (a a))
PS C:\Users\max-h\Desktop\ANTLR\Odd_nr_a> grun Odd_nr_a s -tree

C:\Users\max-h\Desktop\ANTLR\Odd_nr_a>java -cp .;C:\Users\max-h\Desktop\A
NTLR\antlr-4.11.1-complete.jar org.antlr.v4.gui.TestRig Odd_nr_a s -tree
aaa
^Z
(s (a a) (s (a a)) (a a))
PS C:\Users\max-h\Desktop\ANTLR\Odd_nr_a> grun Odd_nr_a s -tree

C:\Users\max-h\Desktop\ANTLR\Odd_nr_a>java -cp .;C:\Users\max-h\Desktop\A
NTLR\antlr-4.11.1-complete.jar org.antlr.v4.gui.TestRig Odd_nr_a s -tree
aaaa
^Z
line 2:0 missing 'a' at '<EOF>'
(s (a a) (s (a a) (s (a a)) (a a)) (a <missing 'a'>))
PS C:\Users\max-h\Desktop\ANTLR\Odd_nr_a> grun Odd_nr_a s -tree

C:\Users\max-h\Desktop\ANTLR\Odd_nr_a>java -cp .;C:\Users\max-h\Desktop\A
NTLR\antlr-4.11.1-complete.jar org.antlr.v4.gui.TestRig Odd_nr_a s -tree
aaaaa
^Z
line 2:0 mismatched input '<EOF>' expecting 'a'
(s (a a) (s (a a) (s (a a) (s (a a)) (a a)) a) (a <missing 'a'>))
PS C:\Users\max-h\Desktop\ANTLR\Odd_nr_a>
```

Figure 16. ANTLR parse attempts for odd number of a grammar

```
PS C:\Users\max-h\Desktop\ANTLR\Indirect left> antlr4 .\IndirectLeft.g4
error(119): IndirectLeft.g4::: The following sets of rules are mutually left-recursive [a, b]
```

Figure 17. ANTLR grammar compilation attempt for indirect left-recursive grammar

```
C:\Users\max-h\Desktop\ANTLR\Direct right>java -cp .;C:\Users\max-h\Desktop\A
NTLR\antlr-4.11.1-complete.jar org.antlr.v4.gui.TestRig DirectRight s -tree
a
^Z
line 2:0 mismatched input '<EOF>' expecting 'a'
(s a s)
PS C:\Users\max-h\Desktop\ANTLR\Direct right> grun DirectRight s -tree

C:\Users\max-h\Desktop\ANTLR\Direct right>java -cp .;C:\Users\max-h\Desktop\A
NTLR\antlr-4.11.1-complete.jar org.antlr.v4.gui.TestRig DirectRight s -tree
aa
^Z
line 2:0 mismatched input '<EOF>' expecting 'a'
(s a (s a s))
PS C:\Users\max-h\Desktop\ANTLR\Direct right> grun DirectRight s -tree

C:\Users\max-h\Desktop\ANTLR\Direct right>java -cp .;C:\Users\max-h\Desktop\A
NTLR\antlr-4.11.1-complete.jar org.antlr.v4.gui.TestRig DirectRight s -tree
aaa
^Z
line 2:0 mismatched input '<EOF>' expecting 'a'
(s a (s a (s a s)))
PS C:\Users\max-h\Desktop\ANTLR\Direct right>
```

Figure 18. ANTLR parse attempt for direct right-recursive grammar

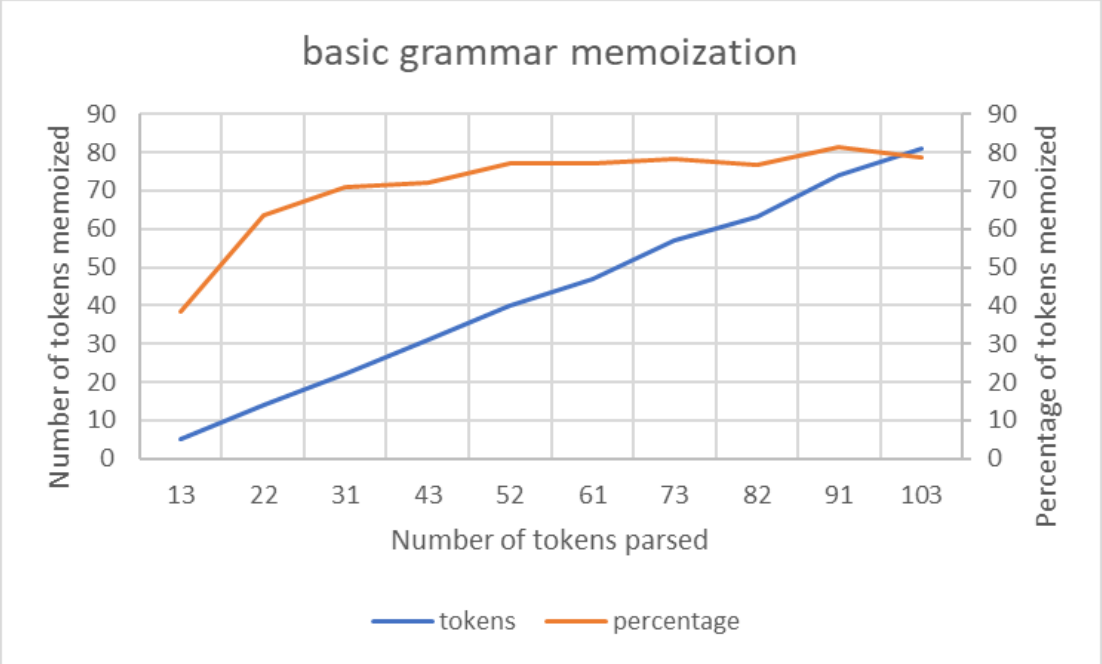


Figure 19. Memoization statistics for basic grammar

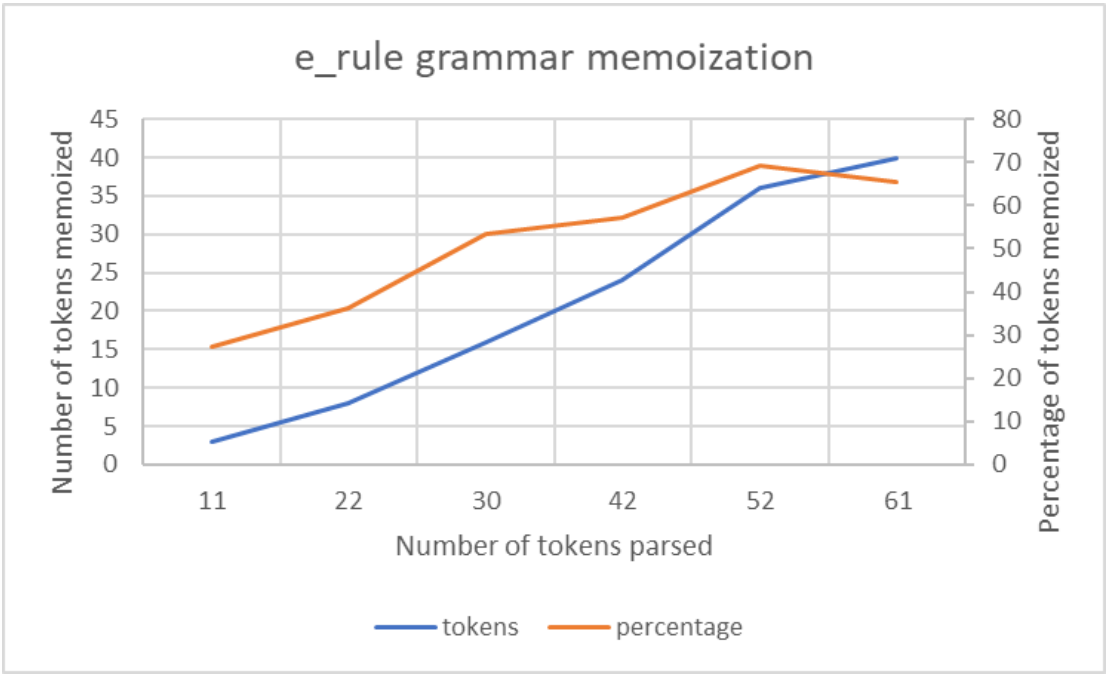


Figure 20. Memoization statistics for e rule grammar

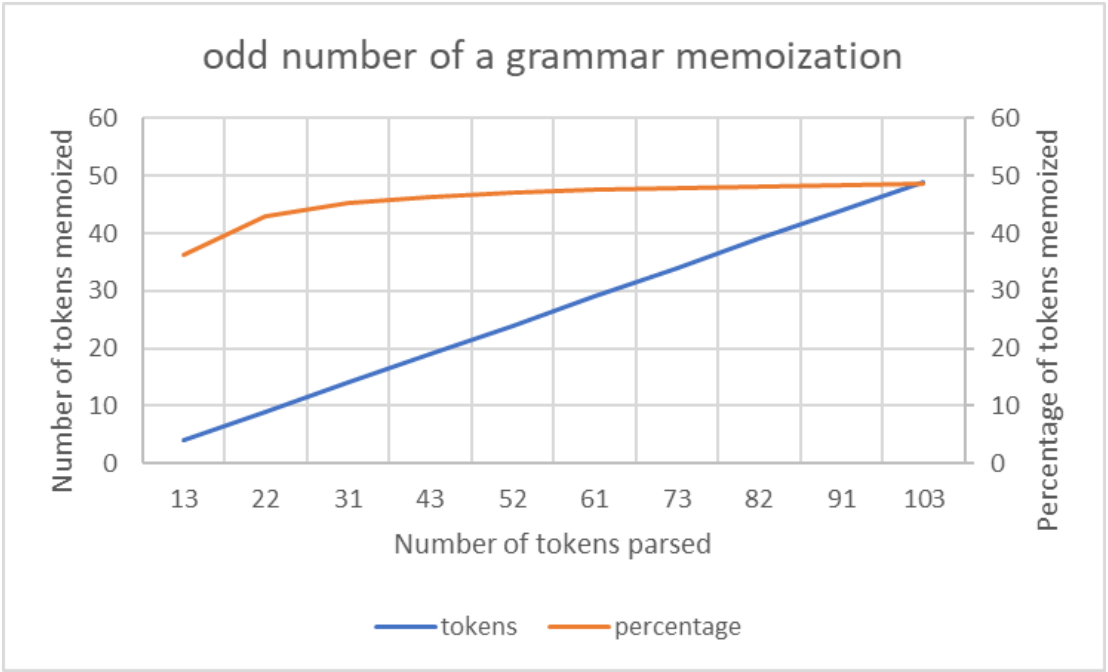


Figure 21. Memoization statistics for odd number of a grammar

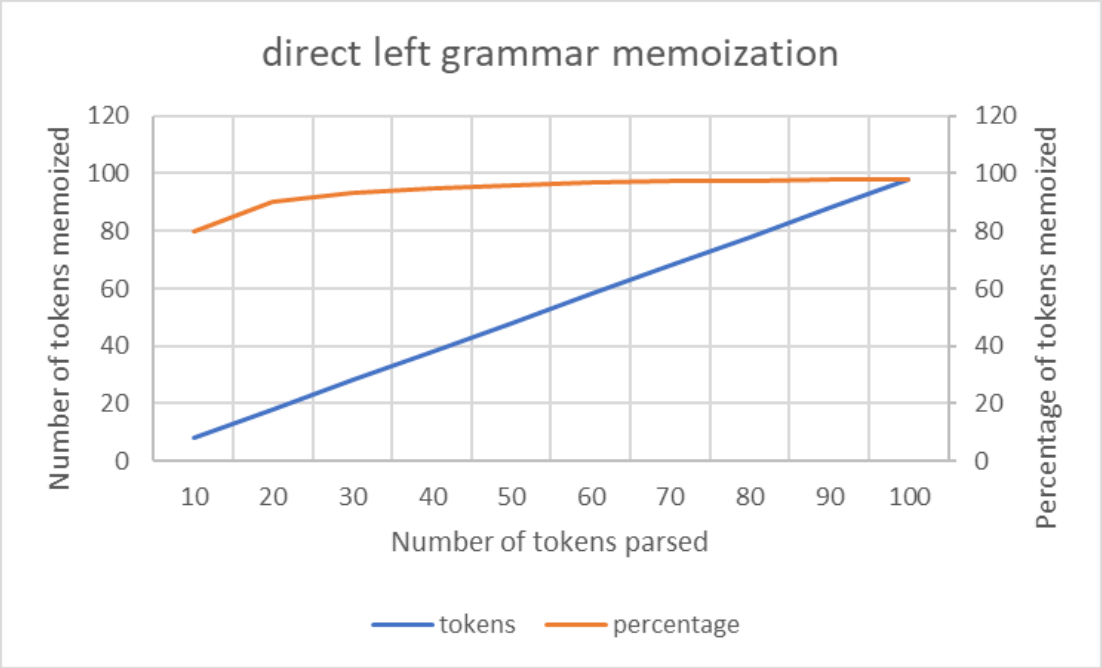


Figure 22. Memoization statistics for direct left grammar

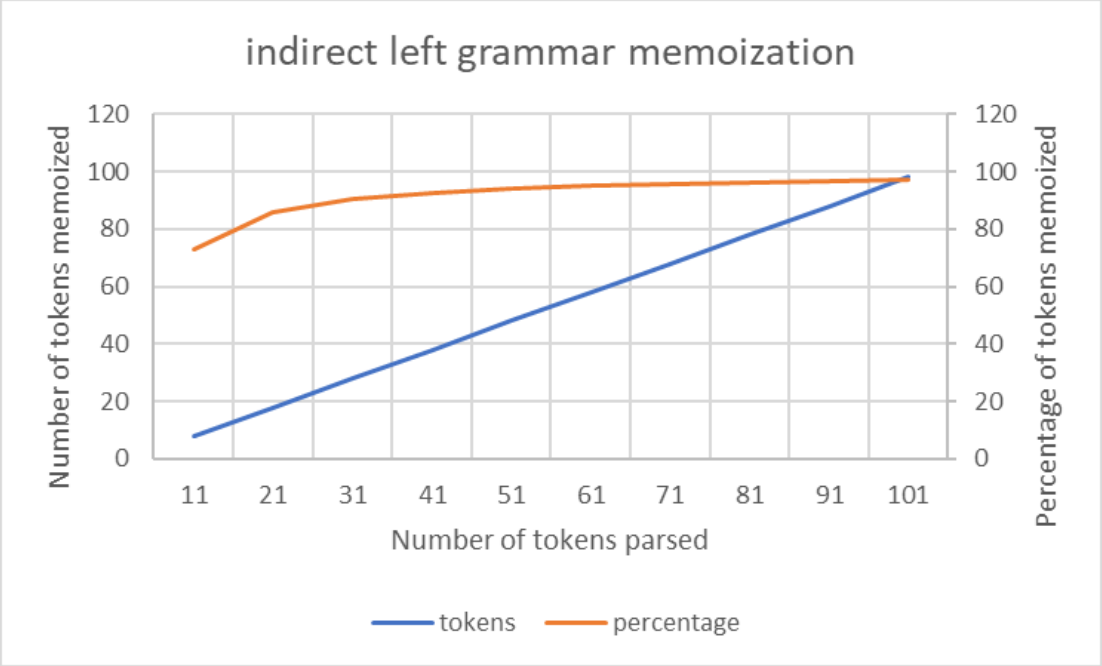


Figure 23. Memoization statistics for indirect left grammar

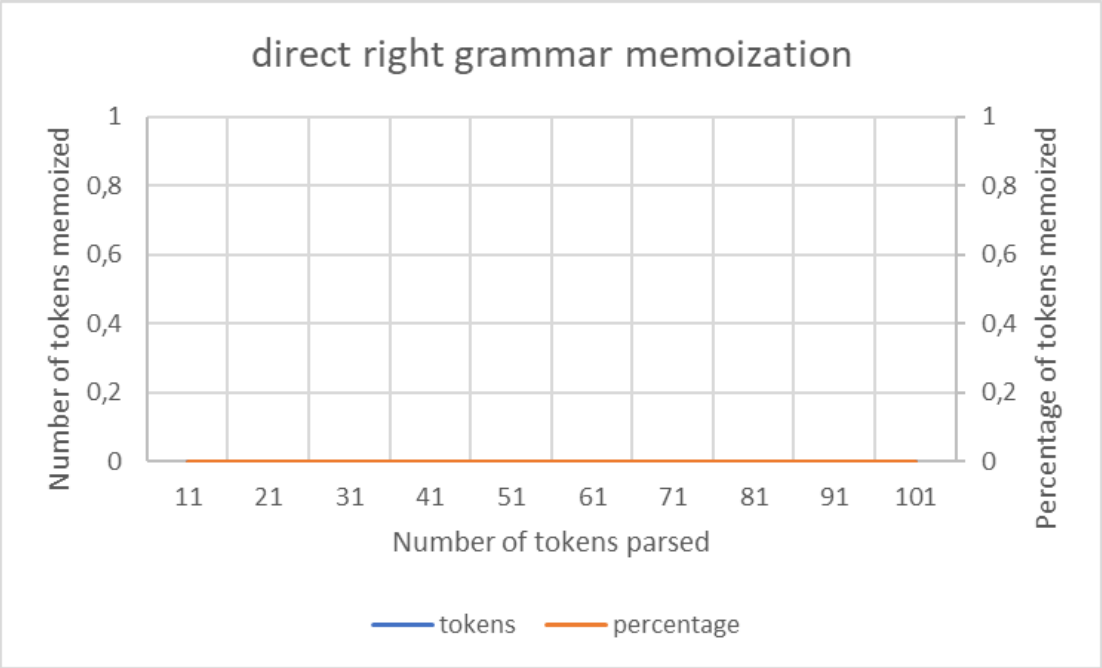


Figure 24. Memoization statistics for direct right grammar

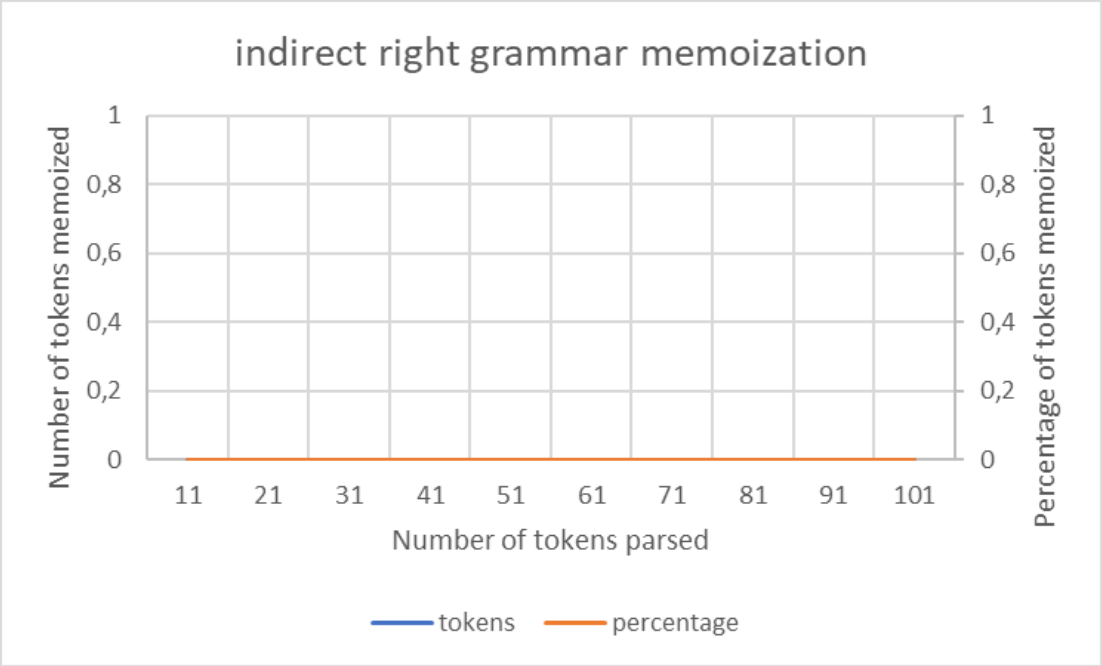


Figure 25. Memoization statistics for indirect right grammar