Fast Packrat Parsing in a Live Programming Environment: Improving Left-recursion in Parsing Expression Grammars

Friedrich Eichenroth, Patrick Rein, Robert Hirschfeld

Technische Berichte Nr. 135

des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Digital Engineering an der Universität Potsdam | 135

Friedrich Schöne | Patrick Rein | Robert Hirschfeld

Fast Packrat Parsing in a Live Programming Environment

Improving Left-recursion in Parsing Expression Grammars

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2022
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Digital Engineering an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
https://doi.org/10.25932/publishup-49124
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-491242

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-503-3

Language developers who design domain-specific languages or new language features need a way to make fast changes to language definitions. Those fast changes require immediate feedback. Also, it should be possible to parse the developed languages quickly to handle extensive sets of code.

Parsing expression grammars provides an easy to understand method for language definitions. Packrat parsing is a method to parse grammars of this kind, but this method is unable to handle left-recursion properly. Existing solutions either partially rewrite left-recursive rules and partly forbid them, or use complex extensions to packrat parsing that are hard to understand and cost-intensive.

We investigated methods to make parsing as fast as possible, using easy to follow algorithms while not losing the ability to make fast changes to grammars.

We focused our efforts on two approaches. One is to start from an existing technique for limited left-recursion rewriting and enhance it to work for general left-recursive grammars. The second approach is to design a grammar compilation process to find left-recursion before parsing, and in this way, reduce computational costs wherever possible and generate ready to use parser classes.

Rewriting parsing expression grammars is a task that, if done in a general way, unveils a large number of cases such that any rewriting algorithm surpasses the complexity of other left-recursive parsing algorithms. Lookahead operators introduce this complexity. However, most languages have only little portions that are left-recursive and in virtually all cases, have no indirect or hidden left-recursion. This means that the distinction of left-recursive parts of grammars from components that are non-left-recursive holds great improvement potential for existing parsers.

In this report, we list all the required steps for grammar rewriting to handle left-recursion, including grammar analysis, grammar rewriting itself, and syntax tree restructuring.

Also, we describe the implementation of a parsing expression grammar framework in Squeak/Smalltalk and the possible interactions with the already existing parser Ohm/S. We quantitatively benchmarked this framework directing our focus on parsing time and the ability to use it in a live programming context. Compared with Ohm, we achieved massive parsing time improvements while preserving the ability to use our parser it as a live programming tool.

The work is essential because, for one, we outlined the difficulties and complexity that come with grammar rewriting. Also, we removed the existing limitations that came with left-recursion by eliminating them before parsing.

Contents

Listings

1 Introduction & Motivation

Language scientists and developers want to explore and test experimental language features. Besides the use case for those language experts, some applications require *domain-specific languages* (DSLs) that have the benefit of serving one specific purpose. Non-experts often develop these.

To meet the needs of those user groups, they need a fast and straightforward way to create languages. It should be easy to create or modify new languages because some design requirements can only become apparent when testing them out.

Any language tool should have a very short runtime to be useful and should be usable in a live environment. Fast parsing is helpful for tools that do syntax highlighting or code hinting. Also, it is mandatory when we need to parse large sets of source codes. Having a parser that we can consider live is significant because, during the construction of a language, a developer wants to add and test a lot of minor changes fast without any delays.

Our goal should be to define languages using grammars in an easy and fast manner. Grammars should be able to describe all aspects of programming languages and easily human-readable. Based on those language defining grammars, we want to parse strings using an easy to understand algorithm to eliminate sources of error and to be accessible. Also, we want to be able to make changes to grammars that get applied quickly.

1.1 Contributions

In this report, we establish what is necessary to rewrite arbitrary grammars to make them easily parsable. For this purpose, we provide several algorithms that outline the necessary steps during rewriting. Altogether, we show that rewriting grammars is a non-trivial problem that requires considering numerous cases.

Furthermore, we describe a grammar parsing framework that uses a combination of existing algorithms and careful preprocessing to make parsing faster without limiting users in their use of grammars. We implement the framework in Squeak/Smalltalk. At last, we show that our framework is usable in a live environment and benchmark it against existing parsing algorithms.

1.2 Structure of this Report

First, we lay out the essential foundations for this report in chapter 2. We outline the general direction, draft a problem statement, and define underlying formalisms.

The significant portion of this work consists of two mostly independent parts. In chapter 3, we discuss the necessary steps and aspects one needs to consider when rewriting parsing expression grammars. In chapter 4 and chapter 5, we describe a parser generator framework in its design and the resulting implementation.

We conduct a quantitative evaluation of our implementation in the form of benchmarks and report this in chapter 6. Also, we discuss various aspects of the implementation there.

We give an overview of similar approaches and other solutions to our problem statement in chapter 7. Chapter 8 concludes the report, re-examines our problem statement, and points to possible future work.

2 Background

In this chapter, we lay the foundations for the following chapters. We give an overview of language parsing and grammar formalisms. Using *parsing expression grammars*, we introduce the theoretical basis and definitions for this report. We also describe the challenges that occur while parsing and why individual parsers are fast or slow.

2.1 Parsing Expression Grammar

Parsing Expression Grammars (PEGs) were first introduced in 2002 [4] and accurately defined in "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation" [6]. They have the purpose of providing a recognition-based formal foundation for describing programming languages in a human-readable manner.

Furthermore, they have the benefit that it is possible to parse any string in linear time, except in some well-known cases using a method, called packrat parsing.

2.1.1 Features

The syntax of PEGs is similar to *Extended Backus-Naur Form* (EBNF) [1, 24]. Like EBNF, it is possible to denote unbounded repetitions of expressions and optional expressions.

What makes PEGs different from EBNF is the possibility to use the lookahead operators `&` and `!` The operator `&` denotes that a specific part of a string should be parsable by the annotated expression without actually going forward within the input string, meaning consuming no characters. The operator `!` excludes the ability to parse a string by an annotated expression. Like EBNF grammars, PEGs contain prioritized alternatives-expressions. Those alternatives are exclusive and prioritized in descending order from left to right.

The language class corresponding to PEG is similar to the language class corresponding to *Context-Free Grammars* (CFGs). However, it is neither a superset nor a subset. Most languages that can be parsed by CFGs can be parsed by PEGs, as well. There is still a difference since CFG parsing has a worst-case runtime of $\mathcal{O}(n^3)$ [3, 10] compared to the linear runtime we can achieve with PEGs.

Also, parsing using PEGs results in a single unambiguous syntax tree. That is due to the use of prioritized alternatives instead of unordered choice in CFGs.

2.1.2 Formal Definition of Parsing Expression Grammars

Each PEG consists of combined parsing expressions that define a set of rules which describes the parsing possibilities of that grammar. We have the following three definitions.

Definition 2.1 (Expression)**.** An expression is a parse instruction that can be primitive, created by modifying another expression or created by combining a collection of expressions. Primitive expressions can match strings directly. The suffix- and prefix-expressions modify their encapsulated expression. Collection-expressions combine expressions of their encapsulated expressions.

In Table 2.1 we show all available expressions.

Table 2.1: Expressions syntax used by a PEG

Expression	Syntax	Type
String	`"abc"`	primitive
Character class	`[A-Z]`	primitive
Any	`.`	primitive
Optional	e`?`	suffix
Zero or more	e`*`	suffix
One or more	e`+`	suffix
Lookahead (And)	`&`e	prefix
Not	`!`e	prefix
Sequence	e_1 e_2	collection
Alternatives	e_1 `|` e_2	collection
Apply	`a`	apply

e, e_1 and e_2 are expressions.

Definition 2.2 (Rule)**.** A rule is a named expression that can be referenced through its identifier. To reference those rules within expressions, we use the apply-expression that can reference the name of a rule. We can use apply-expressions similar to non-terminals in CFGs.

We denote a rule definition like the following.

```
ruleName := [A-Z] | "abc" | b
```

This definition describes a rule named `ruleName` consisting of three choices, namely a character class expression, a string expression, and an apply-expression.

Definition 2.3 (Grammar)**.** A set of rules where each apply-expression points to a rule within this set is called a grammar.

2.2 Packrat Parsing

Packrat parsing was first described in "Packrat Parsing: Simple, Powerful, Lazy, Linear Time" [5]. The author designed it for lazy programming languages, but the technique is not limited to those.

Packrat parsing is a top-down parsing algorithm with backtracking that achieves linear time. Packrat parsing uses a dynamic programming approach. That means it recursively matches the expressions of a grammar and creates a memoization data structure that allows storing results of every rule invocation for every position of a parsed string. That way, the algorithm only needs to invoke every rule once per position in the string. That results in a time and space complexity of $\mathcal{O}(nm)$, with n being the string length and m being the total rule count of the used grammar.

A packrat parser returns for every successful parse a syntax tree which follows the invocation tree of successfully matched expressions.

2.2.1 Limitations of Packrat Parsing

Despite the algorithms' elegance and simplicity, it has a few shortcomings. One is its inability to parse left-recursive grammars. Another is the performance decrease that memoization can introduce in some cases. We take a closer look at those issues in this section.

2.2.1.1 Left-recursion

Left-recursion is a problem that has its origin in the prioritization in the alternatives-expression of PEGs. Since alternatives are parsed by priority, the algorithm can fall into a recursive invocation loop without making any progress on parsing the input string.

The following rule illustrates the problem.

```
a := a "b" | "a"
```

A packrat parser that invokes the parse of rule `a` would select the first choice of the expression and again parse rule `a`, resulting in an infinite call stack.

Banning left-recursive rules from PEGs is not an option, as we demonstrate in the following example.

Example 2.1 (Necessity of Left-recursion and Hierarchy)**.** Listing 2.1 contains a simple math grammar for arithmetical formulas.

In the grammar, the precedence of operators is incorporated using hierarchically ordered rules, and the left-associativity of operators is incorporated using left-recursive rules. Both precedence and left-associativity are necessary. To see why, we can use the grammar to construct the syntax tree for the string `"1-2*3+4"` depicted in Figure 2.1.

If we evaluate the tree bottom-up, we recognize that the tree represents the correct arithmetical semantics. The operations are left-associative, meaning operations that come earlier in the string are lower in the syntax tree. Also, operations with higher

Listing 2.1: Simple Math Grammar

```
expr    := addexpr
addexpr := addexpr "+" mulexpr |
           addexpr "-" mulexpr |
           mulexpr
mulexpr := mulexpr "*" digit |
           mulexpr "/" digit |
           digit
digit   := [0-9]
```

precedence are lower in the syntax tree than operations with lower precedence. This leads to the correct order of evaluation of operations.

Let us construct two more grammars that do not incorporate precedence and left-associativity, respectively.

Listing 2.2: No-hierarchy Math Grammar

```
expr  := expr "+" digit | expr "-" digit |
         expr "*" digit | expr "/" digit |
         digit
digit := [0-9]
```

If we parse the string `"1-2*3+4"` according to the grammar in Listing 2.2 not incorporating precedence, we get the parse tree in Figure 2.2.

If we evaluate this tree the same way as the previous tree, we get the same semantics as for the formula $((1-2)*3)+4$, which is not our goal.

If we use the grammar in Listing 2.3, we get the syntax tree in Figure 2.3, which has the same semantics as the formula $1-((2*3)+4)$ if we evaluate it bottom up.

As we can see, hierarchy is necessary to represent the precedence of mathematical operations. Left-recursiveness is needed to evaluate left-associative formulas in the correct order.

2.2.1.2 Types of Left-recursion

There are multiple types of left-recursion that need to be addressed in order to implement parsers that support left-recursion. We have direct left-recursion, indirect-left recursion, and hidden left-recursion, which we show in the following three examples. We forego on an example of hidden indirect left-recursion, and note that this case can also exist.

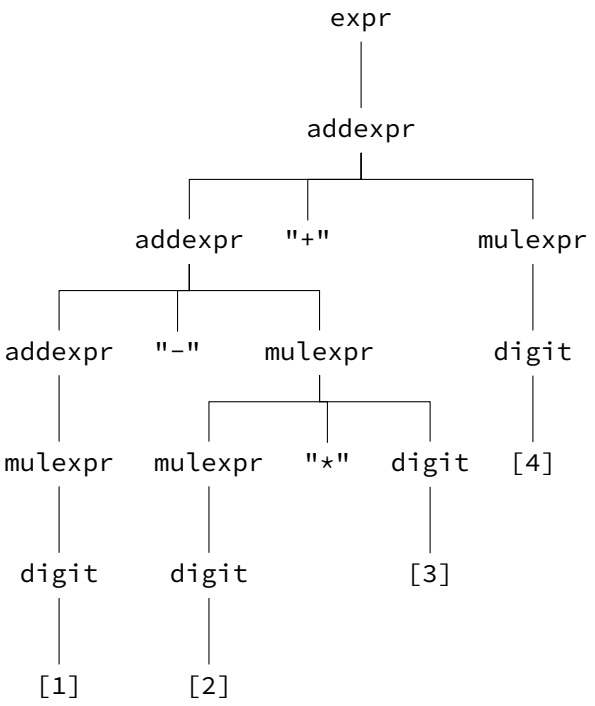


Figure 2.1: The syntax tree for the Simple Math Grammar

Listing 2.3: Non Left-recursive Math Grammar

```
expr    := addexpr
addexpr := mulexpr "+" addexpr |
           mulexpr "-" addexpr |
           mulexpr
mulexpr := digit "*" mulexpr |
           digit "/" mulexpr |
           digit
digit   := [0-9]
```

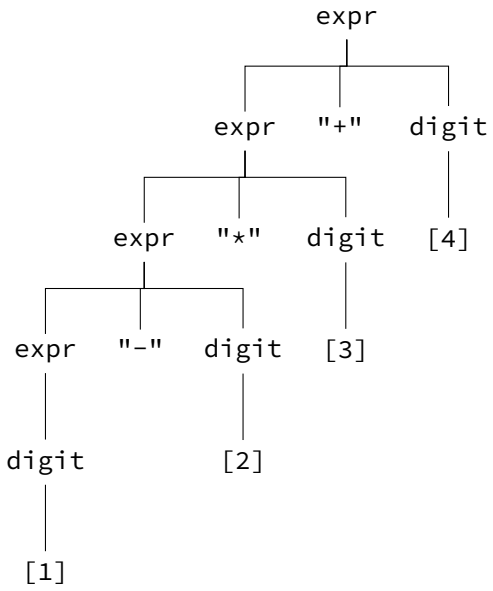


Figure 2.2: The syntax tree for the No-hierarchy Math Grammar

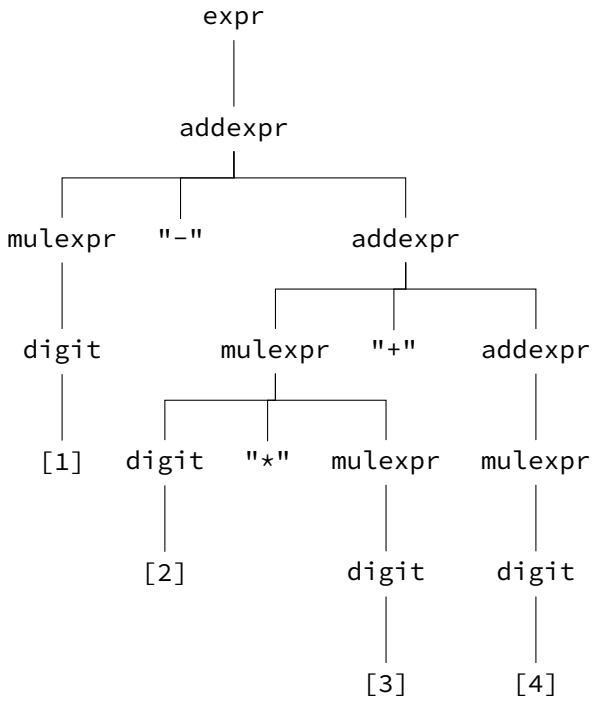


Figure 2.3: The syntax tree for the Non Left-recursive Math Grammar

Direct Left-recursion Direct left-recursion is the simplest form of left-recursion. It occurs when a rule contains an apply-expression that references the rule itself.

Example 2.2 (Direct Left-recursion)**.** We assume that we have a grammar with the following rule.

```
a := a "b" | "a"
```

The rule contains said apply-expression named `a`, which invokes the rule itself without consuming any characters resulting in an infinite call loop.

Indirect Left-recursion Indirect left-recursion occurs when a rule does not invoke itself directly but is part of a larger cycle that does not consume any characters while cycling.

Example 2.3 (Indirect Left-recursion)**.** The grammar shown in Listing 2.4 is indirect left-recursive. Rule `a` invokes rule `b`, rule `b` invokes rule `c`, which invokes rule `a` closing the cycle without consuming any characters.

Listing 2.4: Indirect Left-recursive Grammar

```
a := b "b" | "a"
b := c "c" | "b"
c := a "a" | "c"
```

Hidden Left-recursion Hidden left-recursion can be much harder to detect. Until now, we only considered left-recursion when the cycling apply-expressions were invoked first in their sequence-expression. Hidden left-recursion happens when not all apply-expressions are the first to invoke in their respective sequence-expression.

Example 2.4 (Hidden Left-recursion)**.** When we use the grammar in Listing 2.5 to parse a string with the suffix `"b"` by starting with rule `a` we can get an infinite call loop. Rule `a` invokes rule `b` which results in a parse success without consuming any characters. That leads to the invocation of rule `a` again. Therefore, this rule is left-recursive.

Listing 2.5: Hidden Left-recursive Grammar

```
a := &b a "a" | "a"
b := "b"
```

2.2.1.3 Performance

Despite linear runtime through memoization, the memoization component can slow down the parser more than it improves its performance in real use cases. The authors of the paper "DCGs + Memoing = Packrat Parsing - But is it worth it" [2] showed that memoization almost always leads to a performance loss. Also, they state that using no memoization is very close to optimum performance.

They found out that memoizing only a few rules can gain a significant speedup. Finding those good memoization candidates, however, is difficult. Rules that are higher up in a grammar hierarchy could be appropriate because they have the potential to prevent parsing of large subtrees twice and, therefore, prevent a lot of extra invocations. Rules that are lower in the grammar hierarchy could also be good candidates because they have a much higher chance of getting invoked multiple times.

To address the vast amount of allocated memory, in "Packrat parsing with Elastic Sliding Window" [11], the authors present an approach to minimize the allocation memory size for memoization. Their allocation size depends on the backtracking activity.

2.2.2 Left-recursive Packrat Parsing

The Warth-algorithm is an extension to regular packrat parsing. First described in "Packrat Parsers Can Support Left Recursion" [22], this algorithm can parse left-recursive grammars but has no guaranteed linear runtime anymore. The asymptotic worst-case runtime is $\mathcal{O}(n^2)$, as we demonstrate in example 2.5.

The algorithm detects if it enters a cycle without consuming any characters of the input string. As soon as the algorithm detects such cycles, it tries to invoke other rules than the already chosen path. After getting a successful parse, the algorithm reapplies the result for the rule that starts the cycle on the rule itself. The change in rule invocation order results in the parse tree growing upward instead of downwards for the duration of left-recursive parses.

This method requires a lot of additional parser state that it continually edits. That means that the extension complicates the algorithm, and the method loses the elegance of the regular packrat parsing algorithm.

Example 2.5 (Super-linear Parse Time Grammar)**.** The authors of the Warth-algorithm gave this example [22].

We consider the grammar in Listing 2.6. Using this grammar and the Warth-algorithm we can parse strings of `"1"`s in $\mathcal{O}(n^2)$ time. This is because, after every `"1"`, the parser reenters a cycle and again parses the whole remaining string.

2.3 Existing Parsers in Squeak/Smalltalk

We choose Squeak/Smalltalk as our implementation environment. Squeak is a live programming environment using the Smalltalk language. To have a baseline for

Listing 2.6: Super-linear Parse Time Grammar

```
start := ones "2" | "1" start | ""
ones  := ones "1" | "1"
```

later evaluation and to analyze what makes specific parsers slow or fast, we give an overview of some existing parsers.

2.3.1 Ohm

Ohm is a feature-rich parsing framework that is intended to be an interface definition that can be implemented in multiple programming languages [23].

All Ohm features can be accessed over its grammar definition language. Ohm supports inheritance, including overriding and extending rules. It allows the use of left-recursive rules by implementing the Warth-algorithm. To allow an easy higher-level language description, it enables the differentiation of lexical and syntactical rules. The difference between those two is that syntactical rules try to parse spaces in between expressions without adding them to the resulting syntax tree. That allows specifying lexing and hierarchical parsing using the same grammar while not worrying about delimiting when describing the hierarchical structure.

There currently exist two implementations of Ohm. Ohm/JS is the original implementation [23] and Ohm/S is the Squeak/Smalltalk reimplementation [17]. The Ohm package for Squeak contains an unofficial Squeak/Smalltalk grammar.

The rich feature set of Ohm has adverse effects on the practical runtime. The cost-intensiveness differs between features; we have a closer look at those in section 2.4.

2.3.2 Squeak Parser

The Squeak compiler uses the class `Parser` to transform source code to traversable ASTs to generate bytecode later on.

This parser is handwritten and is only able to parse Smalltalk code in the Squeak dialect, which is not left-recursive; therefore, Squeak does not handle left-recursive rules. There exists no official Squeak/Smalltalk grammar, but there is an official Smalltalk-80 grammar [8], which has fewer features than the Squeak dialect.

The parser isolates syntactical parsing from lexing and uses a character array for the lookup of character categories. It is at least two orders of magnitude faster than Ohm for parsing Smalltalk code.

2.3.3 Pharo `RBParser`

Pharo is a Squeak fork using the class `RBParser` for compilation. It has comparable parse times as the Squeak parser and uses a similar parsing strategy compared to

its counterpart. It tokenizes the string before syntactical parsing using a lookup array.

2.4 Fast and Slow Parts of Existing Parsers

As mentioned above, the compiler parser is around two orders of magnitude faster than the Ohm parser. We have a closer look at the reasons for that. Since the Squeak and the Pharo parser are so similar, we focus on the Squeak parser.

2.4.1 Left-recursion Handling

One particular performance costly feature is left-recursion handling. The Squeak parser class does not include it, but Ohm does. Experiments show that deactivating it for non-left-recursive grammars in Ohm by modifying the parsing code results in an improvement of the parsing speed by around 30%.

The reason for the slowness of left-recursion handling is the constant modification of parse state and the checks on the rule invocation stack to detect invocation cycles.

This slowdown holds massive potential for improvement since most grammars only have a few small or no parts that are left-recursive. Mostly these parts are the rules that describe arithmetic, algebraic, or similar operations.

2.4.2 Apply Expression Processing

In the grammars of the Ohm package, the most frequent expressions are apply-expressions by far. They hold another chance for improvement.

In Ohm, apply-expressions for the same rule are different objects. They cache the invoked rule after looking it up. Still, caching does only really help if the same apply expression gets invoked from the same parent expression using the same object pointer.

To minimize lookups, one could replace all apply-expression objects that reference the same rule by one single object. One could also do the lookup before the parsing, for example, while editing, to minimize parsing runtime even more.

2.4.3 Separation of Syntactical Parsing and Lexing

The Squeak parser uses a lookup table in which it can check the category of every character fastly. This check can be done because there is a guarantee that any character is part of a distinct character group.

Ohm is unable to do this because characters can be part of multiple lexical rules in arbitrary grammars. We have no certainty that any character is just part of one specific group. Grammars can even have significant intersections between lexical rules.

This lookup step improves parsing speed because, during parsing, it can walk forward in the input string until the category of a character changes. Ohm has to build multiple parse trees that identify the lexical group for every character. We show the significance of this in the following example.

Example 2.6 (Parsing with Different Strategies)**.** We assume that we want to parse the identifier string `"nameOfMethod123"`. We can define typical rules that can be of use to parse identifiers as in the grammar in Listing 2.7.

Listing 2.7: Identifier Grammar

```
identifier := letter alnum*
letter     := [A-Z] | [a-z]
alnum      := letter | digit
digit      := [0-9]
```

(a) Parsing `"a"` **(b)** Parsing `"1"`

Figure 2.4: Rule parse invocation trees for the Identifier Grammar

In Figure 2.4 we see the parse invocation trees for a letter and a digit using the identifier grammar. Parts of the trees that a parser discards during the process are gray. The Squeak parser can look up any character in the lookup array, and as soon as it knows it is in the state of parsing an identifier, it can skip all characters that have the letter or the digit category. It does not need to create and remove tree node instances, and it does not require such a significant number of rule invocations.

2.5 Liveness

Immediate feedback is a beneficial property of software systems. This property requires liveness. Following the definition in "A Perspective on the Evolution of Live Programming" [20], we can define liveness as the permission for a programmer to edit a program while it is running as well that execution is continued right after editing without noticeable interruption.

3 Rewriting Parsing Expression Grammars

We identified left-recursion as a performance-wise costly part of parsers. We now pursue a theoretical approach to eliminate left-recursion.

We try to analyze and rewrite arbitrary grammars in such a way that we do not have to change the simple packrat parsing algorithm, which does not allow left-recursion. This algorithm has the advantage that it is easy to understand, fast, and grammar-based; therefore, it would be beneficial if it could be used it for any grammar.

We want to work out the challenges of this problem in this chapter. We build up techniques to rewrite left-recursion in PEGs and finally layout a conclusion for rewriting rules.

3.1 The Idea

Algorithms for CFGs that remove left-recursion already exist [7, 15]. Since PEGs and CFGs are very similar in their expressiveness [13], it could be possible to do the same for PEGs. When we rewrite an arbitrary grammar to a non-left-recursive grammar, it means that the new grammar contains no set of rules that constitute a left-recursive cycle in any way.

Let us consider a simple grammar that only consists of one rule.

Listing 3.1: Simple Left-recursive Grammar

```
A := A α1 | A α2 | β1 | β2
```

Let us also consider the following simple non-left-recursive grammar.

Listing 3.2: Simple Non Left-recursive Grammar

```
A  := β1 A′? | β2 A′?
A′ := α1 A′? | α2 A′?
```

Both grammars are able to parse all strings that have a prefix that matches the expressions β_1 or β_2, followed by a string that matches any number of the expressions α_1 or α_2.

In contrast to the first one, the second grammar is non-left-recursive, and therefore, an original packrat parser could be used to parse it. Due to the different grammar structures, however, the resulting syntax tree for the second grammar would not match the rules of the first grammar. That is why we need to rewrite the syntax tree such that it matches the original grammar.

3.2 Rewriting Criteria

The resulting syntax trees must match the original grammar rules in such a way that we can map every syntax tree node to a rule from the original grammar. Also, any resulting syntax tree should equal its corresponding syntax tree of existing methods to parse left-recursive PEGs [12, 22].

That means that if we get any intermediate trees that result from a parse of a rewritten grammar, which we need to rewrite to match the original grammar, they should be unambiguous. In other words, there should be an injective function that assigns a final tree to every intermediate tree.

We use the packrat parsing algorithm due to its simplicity; therefore, the rewriting algorithm should not exceed the parsing algorithms complexity by a large scale, meaning they should not be overly difficult to understand. Otherwise, it would be desirable to use existing left-recursive parsing methods, and the rewriting approach would be useless.

3.3 Rewriting Different Types of Left-recursion

We can split the general problem of left-recursion rewriting into multiple, more specific problem statements. The different problems partly create new issues we need to describe and solve as well.

First, we develop a generalized schema for direct-left recursion rewriting. To rewrite indirect left-recursion, we first need to find an algorithmic way to detect them, which we will explain before describing the rewriting technique. Lastly, we will describe the necessary steps to rewrite hidden left-recursion, including the detection of it.

3.3.1 Direct Left-recursion

We define a generic direct left-recursive grammar in the subsequent Listing 3.3.

We need to note that the order of choices plays a significant role here. We can determine that parsing β_k excludes any α_ℓ with $\ell > k$ from the parsing possibilities that follow. This fact can be demonstrated using the following counterexample.

Listing 3.3: Generic Direct Left-recursive Grammar

```
A := A α1 | β1 | A α2 | β2 | ... | A αn | βn
```

Example 3.1 (Counterexample Expression Order)**.** Let S be a string that matches the sequence-expression $\beta_2\alpha_3\alpha_1$. The parse tree in Figure 3.1 is not valid. At the beginning of the string, a parser can choose β_2. The parse tree includes the choice (A α_3), which matched at the beginning. A parser would only choose (A α_3) if choice β_2 had been checked earlier because of its higher priority. Therefore (A α_3) cannot be applied here.

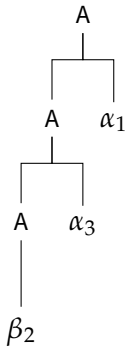


Figure 3.1: Example Syntax Tree that is Impossible

To get a non-left-recursive grammar that represents the same language as our Generic Direct Left-recursive Grammar, we need to combine every non-left-recursive choice with all left-recursive expressions that have a higher priority. We demonstrate this in Listing 3.4.

Listing 3.4: Generic Rewritten Direct Left-recursive Grammar

```
A   := β1 A'1? | β2 A'2? | ... | βn A'n?
A'1 := α1 A'1?
A'2 := α1 A'2? | α2 A'2?
A'3 := α1 A'3? | α2 A'3? | α3 A'3?
 ⋮
A'n := α1 A'n? | α2 A'n? | ... | αn A'n?
```

We start with a grammar with just one rule that is an alternatives-expression with $2n$ choices from which n choices are left-recursive, and n choices are not. We get a resulting grammar with $n + 1$ rules and a total of $\mathcal{O}(n^2)$ choices.

3.3.2 Finding Indirect Left-recursive Cycles

We speak of a left-recursive cycle when invoking a rule twice without making any progress within the parsed string, which means that there is a reference cycle in the involved rules.

To find indirect left-recursion of a grammar, we need to detect cycles in a graph generated from this grammar, as we describe later.

Definition 3.1 (Cycle)**.** Given a graph $G = (V, E)$. An ordered list of nodes $n_1 n_2 \dots n_k$ with $n_1, n_2, \dots, n_k \in V$ is a cycle if, and only if for any pair n_i, n_{i+1} it holds, that $(n_i, n_{i+1}) \in E$, and $(n_k, n_1) \in E$.

We only consider cycles that do not consist of smaller cycles. That means, here, all cycles contain any node at most once.

Definition 3.2 (Minimal Cycle)**.** Given a cycle $c = n_1 n_2 \dots n_k$. c is minimal if for any pair n_i, n_j with $i \neq j$ it holds, that $n_i \neq n_j$.

A non-minimal cycle is the concatenation of multiple minimal cycles. Therefore, only considering minimal cycles during rewriting is enough. We can build all other cycles in rewritten grammars using the rules from minimal cycles.

3.3.2.1 Transform the Grammar to a Graph

To find cycles, we need to transform a grammar into a graph, and this way, use existing graph algorithms. To do that, we use expressions as nodes and add directed edges between those. We only create edges between nodes if the corresponding expressions have a referencing relationship between them. The edge creation rules depend on the expression types according to Table 3.1.

Table 3.1: Grammar Graph Edge Creation Rules

Expression node		Adjacent expressions nodes
Optional	e?	the wrapped expression
Zero or more	e*	the wrapped expression
One or more	e+	the wrapped expression
Lookahead And	&e	the wrapped expression
Lookahead Not	!e	the wrapped expression
Sequence	$e_1\ e_2$	the first wrapped expression
Alternatives	e_1 | e_2	every wrapped expression
Apply	A	the referenced rule expression

3.3.2.2 Algorithm

We can use a *depth-first search* (DFS) to traverse the graph and mark visited nodes to prevent recursive cycles. On every cycle detection, we copy the cycle from the expression invocation stack to a result set.

Using this simple approach, multiple results for cycles that have various entry points can be generated. Therefore we need to normalize every cycle deterministically to detect possible cycle duplicates. We opted to rewrite every cycle such that the node with the lowest index is the first node. Since nodes only occur once in a minimal cycle, this method provides unambiguous normalization results.

With DFS, we cannot be sure to reach every node from any node. The trivial approach would be to start the DFS from any node, which could be time expensive. We can optimize our algorithm by first finding all strongly connected components with Tarjan's algorithm [21]. That would make it possible to start the DFS just once in every component. Also, we can add a recursion abortion if the recursive call leaves the strongly connected component. We provide a pseudocode for this algorithm in Listing 3.5.

3.3.3 Indirect Left-recursion

We are now able to detect any minimal cycles in graphs and, therefore, any indirect left-recursive cycles in grammars. To demonstrate how to rewrite those grammars, we define an indirect left-recursive grammar with m rules, each rule consisting of one alternatives-expression with two choices in Listing 3.6.

As an example, we transform the first rule of the grammar, which results in the grammar in Listing 3.7. Since every rule is a potential entry for the cycle, we need to rewrite every rule using this schema. It might happen that we never use most of the new rules to enter the cycle. In that case, the additional rules would not affect method invocations or runtime.

Starting with a cycle of m rules, each consisting of an alternatives expression with one indirect left-recursive and one non-left-recursive choice, we would get $m + m^2$ rules from which m rules have m choices, and m^2 rules have one choice. Thus the total number of choices would be $\mathcal{O}(m^2)$. Here, the exemplary introduction to this new rewriting schema shall suffice. For a complete and formal rewriting schema, we would need to define a universal grammar. If for any rule, more than two choices exist, we would need to combine the rules similar to the combinations in subsection 3.3.1, resulting in an even higher number of choices.

Furthermore, we need to rewrite all minimal cycles, which can potentially result in an even more complex grammar with a huge rule count. The next example outlines that fact.

Example 3.2 (Rewriting Multiple Minimal Cycles)**.** We define the grammar in Listing 3.8 with six indirect left-recursive rules, and Figure 3.2 displays the four minimal cycles that result from the grammar definition. When rewriting this specific grammar, the result is a large number of new rules. Four minimal cycles, each consisting of three rules, results in $4 \cdot 4 \cdot 4 = 64$ new rules.

Listing 3.5: Minimal Cycle Detection Pseudocode

```
Input: Graph G = (V, E)
Output: List of minimal cycles

C := G.components                              // tarjans algorithm
R := {}                                        // result set
S := []                                        // initialize the stack
for c ∈ C do
    n := C.any                                 // random node in C
    visit(n)
end for

R.remove_duplicates()                          // normalize cycles and
                                               // remove duplicates
return R

function visit(n)
    S.push(n)
    for n' ∈ n.neighbors do
        if n'.component = n.component then
            if n' ∈ S then
                R.add(S.top_until(n')          // write top of S into R
                                               // until n' appears
            else
                visit(n')
            end if
        end if
    end for
    S.pop()
end function
```

Listing 3.6: Generic Indirect Left-recursive Grammar

$$
\begin{aligned}
&A_1 \texttt{ := } A_2\ \alpha_1 \texttt{ | } \beta_1 \\
&A_2 \texttt{ := } A_3\ \alpha_2 \texttt{ | } \beta_2 \\
&A_3 \texttt{ := } A_4\ \alpha_3 \texttt{ | } \beta_3 \\
&\ \vdots \\
&A_m \texttt{ := } A_1\ \alpha_m \texttt{ | } \beta_m
\end{aligned}
$$

Listing 3.7: Generic Rewritten Indirect Left-recursive Grammar

$$
\begin{aligned}
&\mathtt{A}_1 \ \mathtt{:=}\ \beta_m\ \mathtt{A}^1_{m-1} \mid \beta_{m-1}\ \mathtt{A}^1_{m-2} \mid \ldots \mid \beta_1\ \mathtt{A}^1_m \\
&\mathtt{A}^1_1 \ \mathtt{:=}\ \alpha_1\ \mathtt{A}^1_m\mathtt{?} \\
&\mathtt{A}^1_2 \ \mathtt{:=}\ \alpha_2\ \mathtt{A}^1_1\mathtt{?} \\
&\mathtt{A}^1_3 \ \mathtt{:=}\ \alpha_3\ \mathtt{A}^1_2\mathtt{?} \\
&\ \ \vdots \\
&\mathtt{A}^1_m \ \mathtt{:=}\ \alpha_m\ \mathtt{A}^1_{m-1}\mathtt{?} \\
&\ \ \vdots
\end{aligned}
$$

Listing 3.8: Multiple Minimal Cycles Grammar

```
a := b "a" | ...
b := c "b" | d "b" | ...
c := e "c" | ...
d := a "d" | e "d" | ...
e := b "e" | f "e" | ...
f := d "f" | ...
```

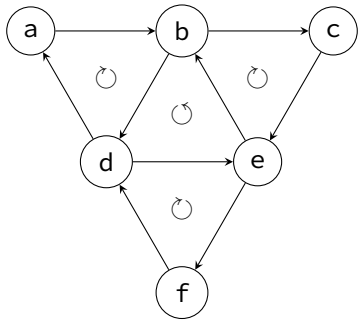


Figure 3.2: Example for a Left-recursive Grammar Graph

3.3.4 Finding Hidden Left-recursive Cycles

To describe the difficulty of finding hidden left-recursion, consider the simple potentially hidden left-recursive grammar in Listing 3.9.

Listing 3.9: Potentially Hidden Left-recursive Grammar

```
A := B A α | β
B := ...
⋮
```

If, somehow, evaluating rule `B` consumes no characters, the evaluation of rule `A` would result in a left-recursive invocation. The algorithm described in subsection 3.3.2 does not take such hidden left-recursive rules into account.

To find out if invoking rule `B` can result in a parse success without consuming any characters, we can again create an expression graph with edges between nodes representing wrapping relationships between these expressions. Starting at the nodes without outgoing edges of this graph and reducing it successively, we can specify for every node whether it can parse an empty string or not. For this process, we use the reduce rules specified in Table 3.2.

Table 3.2: Specifications if Expressions can Parse Successfully by Consuming ϵ

Expression		can parse by consuming ϵ
String	`"abc"`	if $length = 0$
Character class	`[A-Z]`	$\times$
Any	`.`	$\times$
Optional	e`?`	✓
Zero or more	e`*`	✓
One or more	e`+`	if child can parse by consuming ϵ
Lookahead And	`&`e	✓
Lookahead Not	`!`e	✓
Sequence	$e_1\ e_2$	if all children can parse by consuming ϵ
Alternatives	e_1 `|` e_2	if any child can parse by consuming ϵ
Apply	`A`	if referenced rule can parse by consuming ϵ

When we know for each expression if it can parse successfully without consuming any characters, we can build a new graph using a similar algorithm than the

graph creation algorithm specified in subsection 3.3.2. The only edge creation case that changes is the one for sequence-expressions:

$$e = e_1 \,|\, e_2 \,|\, \ldots \,|\, e_n$$

For this expression, we add an edge to the graph from e to e_k for all k, which satisfy that e_ℓ can parse successfully without consuming any characters for every $\ell < k$. With this modification of the graph in place, using the algorithm in Listing 3.5 we again can detect all left-recursive cycles in the grammar.

3.3.5 Hidden Left-recursion

Hidden left-recursive cycles can be found as described in the previous section. To rewrite hidden left-recursive rules, it is necessary to extend the detection of rules that can parse successfully without consuming characters, even more, to make it more differentiated.

To motivate this extension, consider the following example.

Example 3.3 (Motivate Hidden Left-recursion Detection Algorithm)**.** Again, consider the grammar in Listing 3.9. If rule `B` can only parse ϵ and does not contain any lookahead, we could just let go of the apply-expression in rule `A`. If, however, this rule contains a lookahead, it could parse successfully by consuming no characters but is still critical at this place of the grammar, and thus cannot be removed. If the rule can both parse successfully by consuming nonempty strings and parse successfully by consuming ϵ, we need to make sure any rewritten grammar consumes those nonempty strings as well.

We extend Table 3.2 to determine if an expression contains a lookahead, can parse empty strings, and can parse nonempty strings. The reducing rules of this extension are visible in Table 3.3.

To demonstrate which cases are needed to distinguish between, again, take the grammar from Listing 3.9 with the unspecified rule `B`. Depending on whether or not `B` can parse epsilon or can derive a lookahead rule, we need to use different rewriting strategies.

We use the markers $\triangleright$, ϵ, and $\not\epsilon$ to mark the properties *contains lookahead*, *can parse* ϵ, and *can parse* $\not\epsilon$, respectively. Since every rule has at least one of the three properties, we have the cases $\overset{\triangleright}{\texttt{B}}$, $\overset{\epsilon}{\texttt{B}}$, $\overset{\not\epsilon}{\texttt{B}}$, $\overset{\triangleright\epsilon}{\texttt{B}}$, $\overset{\triangleright\not\epsilon}{\texttt{B}}$, $\overset{\epsilon\not\epsilon}{\texttt{B}}$ and $\overset{\triangleright\epsilon\not\epsilon}{\texttt{B}}$. We describe the essential rewriting measures for each of those cases.

3.3.5.1 `B` only contains lookahead ($\overset{\triangleright}{\texttt{B}}$)

In this case, we can not parse an empty string using `B`, but still, `B` does not consume any characters. The choice (`B A` α) needs to start with `B` even when the rule is rewritten since the lookahead at this very position is necessary. We, therefore, rewrite the rule as if it is direct left-recursive and prepend the lookahead rule after rewriting.

Table 3.3: Specifications what Expressions can Consume

Expression	contains lookahead	can parse ϵ	can parse $\notin$
String	×	if $length = 0$	if $length \neq 0$
Character class	×	×	✓
Any	×	×	✓
Optional	wrapped	✓	wrapped
Zero or more	wrapped	✓	wrapped
One or more	wrapped	wrapped	wrapped
Lookahead And	✓	×	×
Lookahead Not	✓	×	×
Sequence	any wrapped	all wrapped	any wrapped
Alternatives	any wrapped	any wrapped	any wrapped
Apply	referenced rule	referenced rule	referenced rule

"wrapped" means true, if and only if the attribute is true for the wrapped expressions.

Listing 3.10: Rewritten Simple Hidden Left-recursive Grammar for Case ▷

```
A := ▷B β A′ | β
A′ := α A′?
▷B := ...
```

Starting with the rule

$$\texttt{A} := \overset{\triangleright}{\texttt{B}}\ \texttt{A}\ \alpha \mid \beta$$

we build the following new grammar.

3.3.5.2 B can only parse empty strings ($\overset{\epsilon}{\texttt{B}}$)

We have rule

$$\texttt{A} := \overset{\epsilon}{\texttt{B}}\ \texttt{A}\ \alpha \mid \beta$$

with the choice ($\overset{\epsilon}{\texttt{B}}\ \texttt{A}\ \alpha$) being effectively the same as ($\texttt{A}\ \alpha$). To get a syntax tree that matches the original grammar, we cannot just remove $\overset{\epsilon}{\texttt{B}}$ from the rule, but still, need to handle the rule like a left-recursive one.

We rewrite the rule resulting in the grammar in Listing 3.11. B below A denotes that the node representing rule A in the syntax tree needs a node representing rule B as left sibling even though it is not part of the rewritten rule anymore.

Listing 3.11: Rewritten Simple Hidden Left-recursive Grammar for Case ϵ

$$\texttt{A} := \beta\ \underset{\texttt{B}}{\texttt{A}'} \mid \beta$$
$$\texttt{A}' := \alpha\ \underset{\texttt{B}}{\texttt{A}'}?$$
$$\texttt{B} := \ldots$$

3.3.5.3 B can only parse nonempty strings ($\overset{\not\epsilon}{\texttt{B}}$)

In this case, $\overset{\not\epsilon}{\texttt{B}}$ cannot parse anything resulting in no consumption of characters. Therefore this case is not left-recursive. We do not rewrite the rules.

3.3.5.4 B contains lookahead and can parse empty strings ($\overset{\triangleright\epsilon}{\texttt{B}}$)

There is no way that the rule consumes any characters. This combined characteristic leads to the need for us to create multiple choices for different parse trees B can derive, each parse tree only having one of both properties.

Let $\overset{\triangleright}{\texttt{B}}$ denote the rule $\overset{\triangleright\epsilon}{\texttt{B}}$ without the choice that leads to parsing ϵ as well as no choices with lower priority. This rule split transformation might make it necessary to transform multiple indirect connected rules.

Note that a parsing path that is resulting in no character consumption always has the smallest priority from all parsing paths. That holds since a parser would not check any other parsing paths once a successful parse consumes no characters.

To rewrite the grammar, we combine the rewriting of the cases $\overset{\triangleright}{\mathtt{B}}$ and $\overset{\epsilon}{\mathtt{B}}$. We introduce new choices to A, resulting in rule

$$\mathtt{A} \texttt{ := } \overset{\triangleright}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$$

Using this rule, we get the following rewritten grammar.

Listing 3.12: Rewritten Simple Hidden Left-recursive Grammar for Case $\triangleright\epsilon$

$$\begin{aligned}
&\mathtt{A} \texttt{ := } \overset{\triangleright}{\mathtt{B}}\ \beta\ \mathtt{A}' \mid \beta\ \underset{\mathtt{B}}{\mathtt{A}''} \mid \beta \\
&\mathtt{A}' \texttt{ := } \alpha\ \mathtt{A}'\texttt{?} \\
&\mathtt{A}'' \texttt{ := } \alpha\ \underset{\mathtt{B}}{\mathtt{A}''}\texttt{?} \\
&\overset{\triangleright}{\mathtt{B}} \texttt{ := } \ldots \\
&\overset{\epsilon}{\mathtt{B}} \texttt{ := } \ldots
\end{aligned}$$

3.3.5.5 B contains lookahead and can parse nonempty strings ($\overset{\triangleright\notin}{\mathtt{B}}$)

This case allows no possible parse for an empty input string. We again split the choices resulting in a rule that can have the following format.

$$\mathtt{A} \texttt{ := } \overset{\triangleright}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\notin}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$$

This rule is only exemplary since there can occur much more choices depending on how mixed up the possible parse paths for rule B are that derive lookahead and nonempty parses. We again combine the rewriting schemes of the previous cases and get the following grammar.

Listing 3.13: Rewritten Simple Hidden Left-recursive Grammar for Case $\triangleright\notin$

$$\begin{aligned}
&\mathtt{A} \texttt{ := } \overset{\triangleright}{\mathtt{B}}\ \beta\ \mathtt{A}' \mid \overset{\notin}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta \\
&\mathtt{A}' \texttt{ := } \alpha\ \mathtt{A}'\texttt{?} \\
&\overset{\triangleright}{\mathtt{B}} \texttt{ := } \ldots \\
&\overset{\notin}{\mathtt{B}} \texttt{ := } \ldots
\end{aligned}$$

3.3.5.6 B contains no lookahead and can parse empty and nonempty strings ($\overset{\epsilon\not\epsilon}{\mathtt{B}}$)

This one is similar to the case $\overset{\epsilon}{\mathtt{B}}$ with the difference that this case is more generalized. Having still no lookahead, the parse result for B can be both empty and not empty.

In the previous case, we were able to remove $\mathtt{B}_\epsilon$ and annotate A such that we know to modify the syntax tree later. Here we again need to create multiple choices for different parse trees that B can derive.

The new split up rule is

$$\mathtt{A} := \overset{\not\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$$

Rewriting the rules and annotating A results in the following grammar.

Listing 3.14: Rewritten Simple Hidden Left-recursive Grammar for Case $\epsilon\not\epsilon$

```
A  := B^{/ε} A α | β A'_B | β
A' := α A'_B?
B^ε := ...
B^{/ε} := ...
```

3.3.5.7 B contains lookahead and can parse empty and nonempty strings ($\overset{\triangleright\epsilon\not\epsilon}{\mathtt{B}}$)

The last case is the combination of all previous ones. It is similar to the case $\overset{\triangleright\not\epsilon}{\mathtt{B}}$ with the difference that there is an additional choice with the lowest priority of the split choices resulting in an empty string parse. We can now get a rule

$$\mathtt{A} := \overset{\triangleright}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\not\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$$

Like in the case for $\overset{\triangleright\not\epsilon}{\mathtt{B}}$, the first two choices are not necessarily in order, and we could have even more choices after splitting. When rewriting the grammar using our split rule, we get the grammar in Listing 3.15.

3.3.5.8 Overview

We give an overview of all different cases for the hidden left-recursive rules in Table 3.4.

Listing 3.15: Rewritten Simple Hidden Left-recursive Grammar for Case $\triangleright\epsilon\not\epsilon$

```
A  := ▷B β A′ | ⊀B β A | β A″_B | β
A′ := α A′?
A″ := α A″_B?
▷B := ...
ϵB := ...
⊀B := ...
```

Table 3.4: Overview of Hidden Left-recursive Rewrites for $\mathtt{A} := \mathtt{B}\ \mathtt{A}\ \alpha \mid \beta$

Case	Split rule	New rules
$\overset{\triangleright}{\mathtt{B}}$	$\mathtt{A} := \overset{\triangleright}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$	$\mathtt{A} := \overset{\triangleright}{\mathtt{B}}\ \beta\ \mathtt{A}' \mid \beta$
		$\mathtt{A}' := \alpha\ \mathtt{A}'?$
$\overset{\epsilon}{\mathtt{B}}$	$\mathtt{A} := \overset{\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$	$\mathtt{A} := \beta\ \mathtt{A}'_{\mathtt{B}} \mid \beta$
		$\mathtt{A}' := \alpha\ \mathtt{A}'_{\mathtt{B}}?$
$\overset{\not\epsilon}{\mathtt{B}}$	$\mathtt{A} := \overset{\not\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$	$\mathtt{A} := \overset{\not\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$
$\overset{\triangleright\epsilon}{\mathtt{B}}$	$\mathtt{A} := \overset{\triangleright}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$	$\mathtt{A} := \overset{\triangleright}{\mathtt{B}}\ \beta\ \mathtt{A}' \mid \beta\ \mathtt{A}''_{\mathtt{B}} \mid \beta$
		$\mathtt{A'} := \alpha\ \mathtt{A}'?$
		$\mathtt{A''} := \alpha\ \mathtt{A}''_{\mathtt{B}}?$
$\overset{\triangleright\not\epsilon}{\mathtt{B}}$	$\mathtt{A} := \overset{\triangleright}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\not\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$	$\mathtt{A} := \overset{\triangleright}{\mathtt{B}}\ \beta\ \mathtt{A}' \mid \overset{\not\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$
		$\mathtt{A}' := \alpha\ \mathtt{A}'?$
$\overset{\epsilon\not\epsilon}{\mathtt{B}}$	$\mathtt{A} := \overset{\not\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$	$\mathtt{A} := \overset{\not\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta\ \mathtt{A}'_{\mathtt{B}} \mid \beta$
		$\mathtt{A}' := \alpha\ \mathtt{A}'_{\mathtt{B}}?$
$\overset{\triangleright\epsilon\not\epsilon}{\mathtt{B}}$	$\mathtt{A} := \overset{\triangleright}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\not\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \overset{\epsilon}{\mathtt{B}}\ \mathtt{A}\ \alpha \mid \beta$	$\mathtt{A} := \overset{\triangleright}{\mathtt{B}}\ \beta\ \mathtt{A}' \mid \overset{\not\epsilon}{\mathtt{B}}\ \beta\ \mathtt{A} \mid \beta\ \mathtt{A}''_{\mathtt{B}} \mid \beta$
		$\mathtt{A'} := \alpha\ \mathtt{A}'?$
		$\mathtt{A''} := \alpha\ \mathtt{A}''_{\mathtt{B}}?$

3.4 Syntax Tree Rewriting

To create a syntax tree that matches the rules of an original grammar we need to restructure the syntax tree of a rewritten grammar.

We modify left-recursive rules to be non-left-recursive, but they remain recursive in some way. Those rewritten rules result in recursion paths that differ from recursion paths in syntax trees derived from the original grammar. All rules that were not left-recursive do not lead to different tree structures and, therefore, do not need to be rewritten.

We mark those rewritten rules with an upper dash or with a superscript number, for example, `A'` or $\texttt{A}^1$.

Using these marks we can identify the rewritten rule paths because they consist of directly connected marked nodes with the same base rule identifier and one unmarked node as their parent. Such paths will be presented in example 3.4, example 3.5, and example 3.6.

To recreate a syntax tree in which the nodes match the rules of the original grammar, we need to rotate the recursive paths that resulted from rewritten rules while preserving additional connected nodes. The rewriting for indirect left-recursions, already incorporates the change of rule order for the rewritten grammar.

However, we need to fix the node names so that they match the rule names of the original grammar. Furthermore, we need to fix the entry-point node of any rotated path. Also, we need to reinsert missing apply-expressions that we remove during rewriting.

We have the following steps for syntax tree modification.

1. Find paths in the syntax tree that are non-left-recursive but should be left-recursive.
2. Rotate the paths, so they become left-recursive.
3. Fix the node names of the paths.
4. Insert missing subtrees.

Processing those rule modification steps results in linear runtime since each of those steps needs only a constant number of tree traversals.

We demonstrate the tree-rewriting steps by walking through several examples.

Example 3.4 (Simple Math Grammar Rewriting)**.** We use the math grammar in Listing 2.1. Using the rewriting schema for direct left-recursive grammars we get the non-left-recursive grammar in Listing 3.16.

Parsing the input string `"1-2*3+4"` according to this grammar, we get the syntax tree in Figure 3.3.

In the tree, we have two paths that we need to rotate. Both are marked with outlined nodes and dotted edges. After rotation, we do not need to do any additional steps. In this case, we do not need to fix any rule names, and we do not need to reinsert missing parse trees. We get in fact the tree in Figure 2.1.

Listing 3.16: Rewritten Simple Math Grammar

```
expr     := addexpr
addexpr  := mulexpr addexpr′?
addexpr′ := "+" mulexpr addexpr′? |
            "-" mulexpr addexpr′?
mulexpr  := digit mulexpr′?
mulexpr′ := "*" digit mulexpr′? |
            "/" digit mulexpr′?
digit    := [0-9]
```

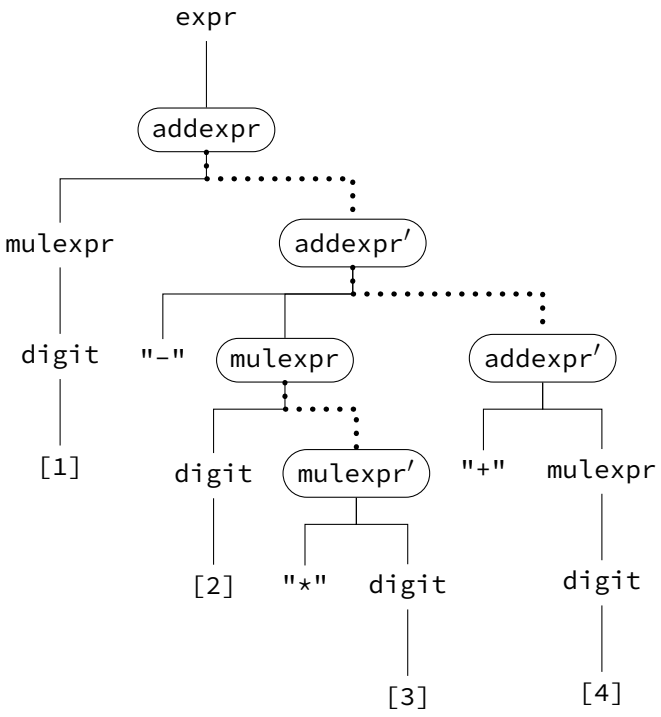


Figure 3.3: The parse tree for the Rewritten Simple Math Grammar

Example 3.5 (Indirect Left-recursion Rewriting)**.** To find a real use case for an indirect left-recursive grammar poses a quite hard problem, even though we need to cover this case. To have an example, regardless, we use a generalized grammar.

We consider the grammar in Listing 3.17.

Listing 3.17: Indirect Left-recursive Example Grammar

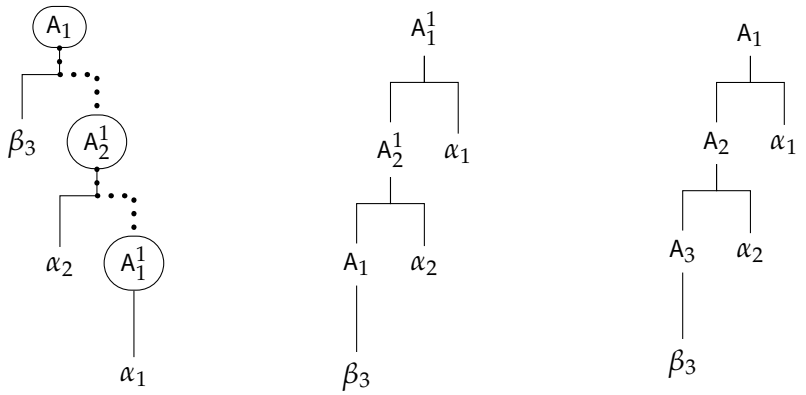
$$
\begin{aligned}
\mathtt{A}_1 &:= \mathtt{A}_2\ \alpha_1 \mid \beta_1 \\
\mathtt{A}_2 &:= \mathtt{A}_3\ \alpha_2 \mid \beta_2 \\
\mathtt{A}_3 &:= \mathtt{A}_1\ \alpha_3 \mid \beta_3
\end{aligned}
$$

Following our rewriting specifications for indirect left-recursions, we get the following non-left-recursive grammar.

Listing 3.18: Rewritten Indirect Left-recursive Example Grammar

$$
\begin{aligned}
\mathtt{A}_1 &:= \beta_3\ \mathtt{A}_2^1 \mid \beta_2\ \mathtt{A}_1^1 \mid \beta_1\ \mathtt{A}_3^1 \\
\mathtt{A}_1^1 &:= \alpha_1\ \mathtt{A}_3^1? \\
\mathtt{A}_2^1 &:= \alpha_2\ \mathtt{A}_1^1? \\
\mathtt{A}_3^1 &:= \alpha_3\ \mathtt{A}_2^1? \\
\\
\mathtt{A}_2 &:= \beta_1\ \mathtt{A}_3^2 \mid \beta_3\ \mathtt{A}_2^2 \mid \beta_2\ \mathtt{A}_1^2 \\
\mathtt{A}_1^2 &:= \alpha_1\ \mathtt{A}_3^2? \\
\mathtt{A}_2^2 &:= \alpha_2\ \mathtt{A}_1^2? \\
\mathtt{A}_3^2 &:= \alpha_3\ \mathtt{A}_2^2? \\
\\
\mathtt{A}_3 &:= \beta_2\ \mathtt{A}_1^3 \mid \beta_1\ \mathtt{A}_3^3 \mid \beta_3\ \mathtt{A}_2^3 \\
\mathtt{A}_1^3 &:= \alpha_1\ \mathtt{A}_3^3? \\
\mathtt{A}_2^3 &:= \alpha_2\ \mathtt{A}_1^3? \\
\mathtt{A}_3^3 &:= \alpha_3\ \mathtt{A}_2^3?
\end{aligned}
$$

We assume that we want to parse a string that matches the sequence-expression $(\beta_3\ \alpha_2\ \alpha_1)$. By using the rewritten grammar, we get the syntax tree in Figure 3.4a with the recursive-path marked with outlined nodes and dotted edges. We need to restructure the tree by rotating that path, which results in the tree shown in Figure 3.4b. At last, we fix the node names and change the name of the path's lowest node to match the original grammar. It is required that the rule corresponding to the node is $\mathtt{A}_1$ in the rewritten grammar so that the parser can find the correct entry point for this subtree.



(a) Syntax tree **(b)** Reordered tree **(c)** Fixed nodes

Figure 3.4: The parse tree for the Rewritten Indirect Left-recursive Example Grammar

Example 3.6 (Hidden Left-recursion Rewriting)**.** Again we construct an arbitrary example because it is quite hard to find a real grammar that covers all cases. Consider the grammar in Listing 3.19.

Listing 3.19: Hidden Left-recursive Example Grammar

```
A := B A α1 | α2
B := β! | !β2 B β2 | β3 | ""
```

Applying our rewriting schema for hidden left-recursion results in the grammar in Listing 3.20.

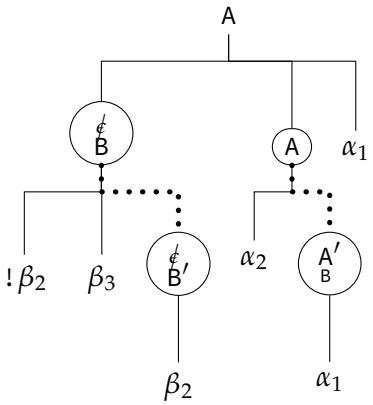
With both grammars, one is able to parse a string that matches the sequence-expression $!\beta_2\beta_3\beta_2\alpha_2\alpha_1\alpha_1$. Using the rewritten grammar results in the syntax tree in Figure 3.5a. Rotating both left-recursive paths, results in the second tree of Figure 3.5. At the node $\underset{\text{B}}{\text{A}}$, it is essential to insert the absent parse tree for rule B and at last, fix the rule names, so they match the first grammar. We get the resulting rewritten syntax tree in Figure 3.5d.

3.5 Conclusion

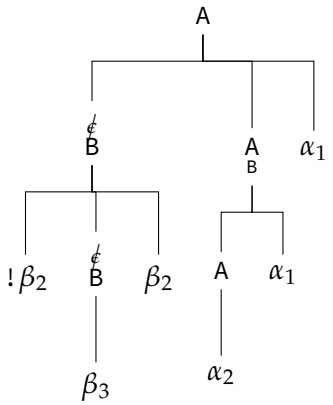
At this point, the report takes a turn. By outlining the required rewriting steps and algorithms, we established that rewriting PGEs is not trivial, and we can state that the complexity exceeds the complexity of already existing parsing solutions for

Listing 3.20: Rewritten Hidden Left-recursive Example Grammar

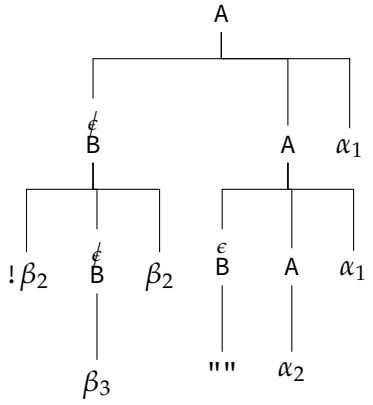
```
A  := B̸ A α1 | α2 A'_B | α2
A' := α1 A'_B?
B̸  := β1 | !β2 β3 B̸' | β3
B̸' := β2 B̸'?
B̸  := ""
```



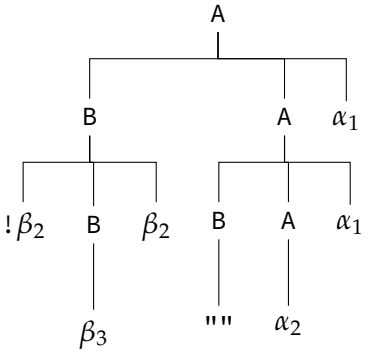
(a) Syntax tree



(b) Reordered tree



(c) Inserted subtree



(d) Finished syntax tree

Figure 3.5: The parse tree for the Rewritten Hidden Left-recursive Example Grammar

left-recursive grammars. That is due to the number of cases we need to take into account for any parse on arbitrary grammars, and the difficulty of finding those cases beforehand.

The complexity gets introduced because of PEGs' lookahead operators. Those are indispensable since they are necessary for the elegant expressiveness of PEGs.

Rewriting grammar seems not to be feasible since it is way too complex to do in a general way. We now will shift our focus to a more practical approach.

4 Parser Generator Approach

We shift our attention towards an engineering-focused approach. We try to eliminate elements of Ohm/S that we consider slow and make use of aspects of the Squeak parser class that are fast. We also aim to do as much computing before the parsing process, even when more expensive, to make the parsing as fast as possible. This shift of computing is advantageous as we assume that we use parsers for parsing much more often than we edit the underlying grammar.

The intention is to generate a function object class from a grammar that does the parsing. We want to convert every expression and, therefore, every rule into a parsing method of the generated class. A generated class has no expensive lookups for values, objects, or rules. It only consists of static code that we compile.

4.1 Compatibility with Ohm

We want to achieve compatibility with Ohm as much as possible to use existing tooling for live language development, including the grammars provided by the Ohm package.

Some of Ohm's features, like rule-extensions, can be simulated by transforming them during generation. Other features, such as inheritance and the difference between syntactical and lexical rules, need to be incorporated into the generated classes to work correctly. To support inheritance of grammars, we can make use of inheritance of classes. Any class that extends a parent class can use inherited rules by calling the corresponding methods. To cover the distinction between syntactical and lexical expressions, we use different generators for each of them. Syntactical expression methods need to call a spacing method without adding space-nodes to the resulting syntax tree. The expressions that are affected by this distinction of syntaxtical and lexical expressions are the apply-expression, the zero-or-more-expression, the one-or-more-expression, and the sequence-expression.

4.2 Left-recursive Rules

As we established before, the handling of left-recursion is expensive since a lot of additional computing is necessary to handle cycle detection and cycle handling.

We also stated that typically, by far, not all rules are left-recursive. We only need to handle left-recursion while parsing left-recursive rules; that is why we need a way to distinguish non-left-recursive rules from left-recursive rules. We

have a closer look at this distinction in section 4.3. When categorizing rules into left-recursive and non-left-recursive ones, we can handle left-recursive rules by using the Warth-algorithm and non-left-recursive rules like the original packrat parser handles them. We need to incorporate the different handling modes into the generated code, so we use different generators for left-recursive rules than for non-left-recursive rules.

The Warth-algorithm requires multiple data structures holding some state during the parse. Those are part of the parser instance, and the parser initializes them at the beginning of a parse, just like the memoization data structure.

4.3 Detecting Left-recursive Rules

We can detect rules that potentially cause left-recursion as described in section 3.3. We depict the single steps that we need to carry out in Figure 4.1.

The first step is to build a derivation graph from the set of rule definitions. Note that we need to incorporate all rule definitions, including rule definitions from grammars the current grammar inherited from, to precisely detect all potential cycles. We explain why in example 4.1 and example 4.2. With a derivation graph, we need to reduce it to know which expressions can parse successfully without consuming any characters. Using the resulting reduced graph, it is possible to build the left-recursion graph. From the left-recursion graph, we extract strongly connected components using Tarjan's algorithm [21]. Every rule expression that is now in a connected component with a size larger than one is potentially left-recursive.

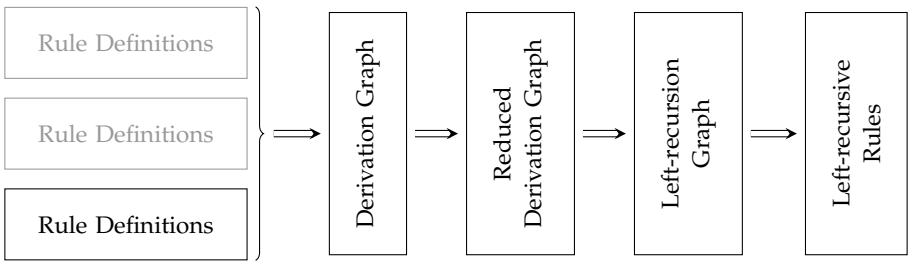


Figure 4.1: The procedure to extract all left-recursive rules from rule definitions

We might need to generate rules in parser classes despite them being already part of an ancestor class. We show within two examples why this is unavoidable.

Example 4.1 (The Necessity of Incorporating Parent Rule Definitions)**.** Consider the following two grammars and assume that the class implementing the second grammar inherits from the class implementing first grammar.

Listing 4.1: Inheritance Parent Grammar A

```
a := b | "a"
b := a "b"
```

Listing 4.2: Inheritance Child Grammar A

```
b := "b"
```

We notice that in the parent grammar, both rules are left-recursive. Considering the child grammar that inherits rule `a` and overrides rule `b`, we notice that neither of these rules is left-recursive anymore, and the implementing classes need to handle them differently.

Example 4.2 (The Necessity of Incorporating Parent Rule Definitions)**.** We take the following two grammars and again assume that the class implementing the second grammar inherits from the class implementing the first.

Listing 4.3: Inheritance Parent Grammar B

```
a := b | "a"
b := "b"
```

Neither of the rules in the parent grammar is left-recursive; however, in the child grammar, both rules become left-recursive due to the overridden rule `b`. We need to handle both rules with different methods in both parsers.

4.4 Tree Flattening

A straightforward approach to build a parser is to generate a syntax tree node for every expression. We, however, want the syntax trees flattened for two reasons.

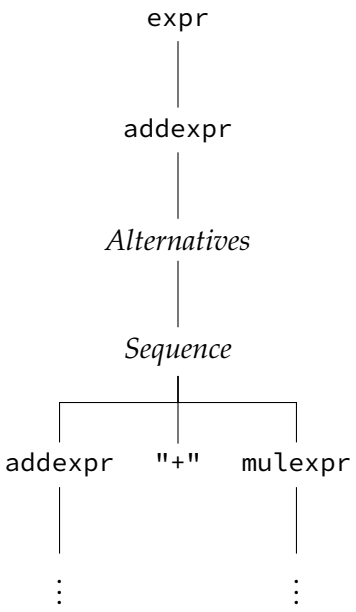
First, the traversal through a tree that has fewer nodes and only named nodes is more comfortable. We illustrate that in the following example.

Example 4.3 (Flattened Tree)**.** We again use the math grammar from Listing 2.1. The natural parse tree that we want to get from our parser using that grammar is visible in Figure 2.1. A tree having a node for every expression is depicted in Figure 4.2. This tree contains nodes that are of no use for us and have no semantic meaning, for example the *Alternatives*-node.

Listing 4.4: Inheritance Child Grammar B

```
b := a "b"
```

Figure 4.2: The unflattened parse tree of the Simple Math Grammar



The second reason is compatibility with Ohm. Since we want to make use of Ohm tooling and grammars, it is helpful to generate parsing trees that have the same structure as Ohm parsing trees, which are flattened as well.

We choose to flatten the tree during parsing; this has the advantage that we can memoize parts of a tree that are already flattened, and we do not have to flatten multiple times when the identical subtrees occur at two points in the syntax tree.

To realize this flattening while parsing, we use shadow nodes. These nodes are, in contrast to ordinary nodes, intermediate and undesired in our final syntax tree. Node objects and shadow node objects differ in their implementation as they both have a differently implemented method `nodeList`. For ordinary nodes, this method returns a list containing just the node itself. For shadow nodes, this method returns the child list of the node. When the parser creates a new node with children, it calls `nodeList` on all its children and concatenates the resulting lists. This way, the parser shadow nodes reach their children through to their parents and vanish during the parsing process.

4.5 Failure Reporting

For failure reporting, we want to use the same strategy as Ohm. We report the set of failures that occur farthest to the right because they are probably close to the actual failure [4]. We know that only leaf expressions throw failures because all other failing expressions derive from failing leaf-expression. These leaf-expressions are primitive expressions; in other words: string-expressions, character-class-expressions, or any-expressions.

There are three options to do failure reporting. For one, we could add the failures to the return statements of parsed expressions. This way, they traverse the parse invocation tree until they reach the initial parse invocation. The second option is to throw exceptions for failed parse invocations. The third option would be to save failures globally in the parser object.

The last option seems more feasible since the first two options introduce the need for failure handling in every expression method template instead of only in the templates for primitive expression methods.

4.6 Memoization

Memoization can be expensive due to the allocation of a large portion of heap memory, sometimes being even more expensive than not memoizing [2].

Experiments show that in the case of parsing Smalltalk, using a generated packrat parser class leads to an apparent performance gain of at least 25% and therefore we will incorporate memoization into our parser generator.

5 Parser Generator Implementation

The implementation of the parser generator consists of multiple parts. We discuss the most critical generator parts in this chapter.

We start with the parser class itself, which is generated for every grammar. We present such a parser in a general way so we can apply it on arbitrary grammars. Next, we describe the generation process of this parser, especially the use of the generator and which decisions the generator has to make.

Compatibility with Ohm introduces the need for a converter from Ohm grammars to make use of them.

5.1 Parser

To give an understanding of the parser, we describe its usage, functionality, and features. Those include control flow description and implementation detail decisions as well as supported capabilities of the parser.

5.1.1 Interface

Any parser class provides an entry point method that triggers parsing:

```
parse: <string> startingFrom: <ruleIdent>
```

This method invokes the parse of the given string using the starting rule that is referenced by the rule identifier.

The method returns a match result object that contains information if the parse succeeded or failed. For successful parses, it also includes the syntax tree in the form of a root node. For failed parses, it contains a failure object representing the rightmost failure set, which is a list of elements that would lead to a further right parse as described in section 4.5.

5.1.2 Structure

While parsing, several different rule types are involved, and multiple methods of the parser object call each other. We give an overview of the order of method calls and the motivation for this implementation approach.

5.1.2.1 Method Names

Each expression has a corresponding method in its analogous parser class. Those expression methods are named depending on their containing rule.

To explain how we name expression methods, we note that expressions always follow a tree-like structure. The expression that is the rule itself gets the rule name. Subexpression names consist of two parts, their parent expression name and a number that refers to its position within its sibling expressions. That means that expression method names indicate all their ancestor expressions and the level in the expression hierarchy. For an expression method that is named `<name>` its k children are named `<name>_1`, `<name>_2`, ..., and `<name>_<k>`.

Using this naming schema, one can generate all subexpression names recursively by only knowing the parent expression name, and it does not create any naming conflicts.

Expression method names also get the prefix `"parse_"` to indicate that they are not auxiliary methods. Expression methods get the position in the parse string as argument, so they have the information where in the string they have to invoke the rule.

Example 5.1 (Expression Method Naming)**.** We assume that we have a grammar with the rule

```
simpleIdent := [a-z] ([a-z] | [0-9] | "_")*
```

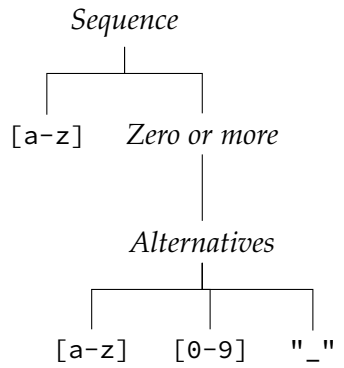


Figure 5.1: The expression tree of the `simpleIdent` rule

The tree structure of this rule is visible in Figure 5.1. Using this rule we get the method names in Listing 5.1.

Apply-expressions follow a different naming schema since they are just referencing rules by name. We generate a corresponding apply-expression method for every rule. These apply-expression methods have the prefix `"apply_"`. We will now have a closer look at the combination of apply-expression methods and other expression methods.

Listing 5.1: `simpleIdent` Method Heads

```
1 parse_simpleIdent: <pos>
2 parse_simpleIdent_1: <pos>
3 parse_simpleIdent_2: <pos>
4 parse_simpleIdent_2_1: <pos>
5 parse_simpleIdent_2_1_1: <pos>
6 parse_simpleIdent_2_1_2: <pos>
7 parse_simpleIdent_2_1_3: <pos>
```

5.1.2.2 Apply

As stated above, apply-expression methods follow a different naming convention than other expressions. To keep the recursive naming convention of rules consistent, we introduce a new method. Apply wrapper methods follow the naming schema of the other expressions and only call their corresponding apply-expression.

This has the advantage that they are easy to generate because we can produce apply wrapper methods and apply methods independently.

We can use an apply wrapper method to call the corresponding apply-expression method, which does the memoization and, if necessary, left-recursion handling and calls the actual parse method of its rule expression.

Example 5.2 (Apply Methods and Apply Wrapper Methods)**.** Consider the following rule.

```
literal := simpleIdent | number
```

This rule results in the corresponding expression method names, as shown in the following listing.

Listing 5.2: `literal` Method Heads

```
1 parse_literal: <pos>
2 parse_literal_1: <pos>
3 parse_literal_2: <pos>
```

Both subexpression methods `parse_literal_1` and `parse_literal_2` are apply wrappers calling their actual apply method, respectively. The source codes of the first apply wrapper method and its corresponding apply-expression method are given in Listing 5.3 and Listing 5.4.

The apply method accesses the memoization data structure at line 2 and parses the `simpleIdent` rule at line 5 in case no memoization entry is found.

Listing 5.3: `simpleIdent` Apply Wrapper Method

```
1 parse_literal_1: pos
2     ↑ self apply_simpleIdent: pos
```

Listing 5.4: `simpleIdent` Apply Method

```
1  apply_simpleIdent: pos
2      ↑ (memoization at: pos ifAbsent: [↑ nil])
3          at: #parse_simpleIdent ifAbsentPut: [
4              | childResult |
5              childResult := self parse_simpleIdent: pos.
6              childResult
7                  ifNil: [nil]
8                  ifNotNil: [
9                      PEGNode
10                         newOn: #simpleIdent
11                         from: pos
12                         to: childResult endPos
13                         children: childResult nodeList ] ]
```

If `simpleIdent` would be involved in a left-recursive cycle, we would generate a different method for the apply, independently from the apply wrapper method. The generated method would have a different code invoking the methods that handle left-recursion according to Warths' algorithm.

5.1.2.3 Lexical and Syntactical Rules

As mentioned in section 4.1, we want the ability to differentiate between lexical and syntactical expressions. Lexical expression methods should parse the expressions according to their definitions. Syntactical rule methods, on the other hand, need to try to parse spaces in between their subexpressions without including them in the syntax tree. We show the distinction with the following example for two zero or more expressions.

Example 5.3 (Lexical vs. Syntactical Expression Methods)**.** We have the following lexical rule definition.

```
exp := subexp*
```

We also have the same rule definitions with the difference that it is syntactical.

```
Exp := subexp*
```

The parse methods for both the lexical and the syntactical rule parse methods are in Listing 5.5 and Listing 5.6 respectively.

Listing 5.5: `exp` Parse Method

```
1  parse_exp: pos
2      | currentPos currentResult children |
3      currentPos := pos.
4      children := OrderedCollection new.
5      [
6          currentResult := self parse_exp_1: currentPos.
7          currentResult ifNotNil: [
8              currentPos := currentResult endPos.
9              children add: currentResult]
10     ] doWhileTrue: [(currentResult == nil) not].
11
12     ↑ PEGNode
13         newOn: #_list
14         from: pos
15         to: currentPos
16         children: (children collect: #nodeList) flatten
```

As we can see, line 6 of the `Exp` parse method contains the method call to skip any spaces. The skip spaces method calls the method called `apply_spaces`, which means that the grammar needs to have a rule named `spaces`.

5.1.2.4 Superclass

To provide certain functionality like an entry point method, space skipping, left-recursion, and failure handling, we use a superclass that holds the necessary auxiliary methods. This superclass is given by the class `PEGParser` from which all parser classes inherit directly or indirectly. This class also provides data structures necessary during parsing and sets them up at parse begin.

5.1.3 State

To be able to parse, the parser needs to keep some data structures that are provided by a superclass. It generates them on the invocation of the parse method. We will describe them one at a time.

5.1.3.1 Memoization

For memoization, we use an array of identity dictionaries called `memoization`. Identity dictionaries are dictionaries that reference by the identity of a key instead

Listing 5.6: `Exp` Parse Method

```
parse_Exp: pos
    | skipPos currentPos currentResult children |
    currentPos := pos.
    children := OrderedCollection new.
    [
        skipPos := self skipSpaces: currentPos.
        currentResult := self parse_exp_1: skipPos.
        currentResult ifNotNil: [
            currentPos := currentResult endPos.
            children add: currentResult]
    ] doWhileTrue: [(currentResult == nil) not].

    ↑ PEGNode
        newOn: #_list
        from: pos
        to: currentPos
        children: (children collect: #nodeList) flatten
```

of the value of a key. The array has the size of the parsed string, so for each position, the parser can store parse results for every rule identifier. For example, it would store the result for parsing the rule `simpleIdent` at a position p under the key `#parse_simpleIdent` in the p-th identity dictionary.

In theory, it would have also been possible to use an identity dictionary containing arrays. However, experiments have shown that this is slower than the method we use.

5.1.3.2 Left-recursion

Handling left-recursion requires two data structures, as described by the authors of the Warth-algorithm [22]. Those data structures are named `heads` and `leftRecursionStack`. `heads` is an array with the size of the parse string, containing head objects that mark the beginnings of left-recursion cycles. `leftRecursionStack` is a list containing the left-recursion objects ordered by invocation time.

5.1.3.3 Failure Reporting

The object `rightmostFailure` contains a set of failures that are farthest to the right at any given moment during the parse. The set can contain more than one failure because it is possible to have multiple unsuccessful parse paths that end at the same position in the string. The parser overrides it if a failure occurs that has a higher position in the string than the current failure set.

5.1.4 Thread Safety

The parser classes that we generate are not thread-safe in their current state. That is due to the multiple data structures that introduce state to the parse. To make a parser instance thread-safe, we could create a state class that wraps those data structures. However, it is not necessary because, for thread safety, we can also create multiple instances of the same parser class, which has the same effect as having various state objects.

5.2 Parser Generator

The generator is the main component of our software. It does the essential computing steps for our parsers and is the part that differs from other PEG parsers.

In this section, we describe the necessary steps and nuances of the parser generation process.

5.2.1 Interface

A parser generator object is an instance of the class `PEGParserGenerator`. To generate a parser, we need to add rules that are part of the grammar. We add those rules by calling the following method.

```
defineRule: <ruleName> withExpr: <exprGenerator>
```

The expressions that we add come in the form of expression generator objects. To receive these expression generator objects, we can use methods of the parser generator object providing them. A list of all methods that return an expression generator is depicted in Table 5.1. We explain expression generators in subsection 5.2.2.

We can generate the entire parser class by calling the following method.

```
generate: <grammarName> inheritFrom: <parentClass>
```

To demonstrate the usage of the parser generator, we generate a parser that can parse the math grammar defined in Listing 2.1 by using the parser generator interface as follows.

We define a parser generator object in line 1. After that, we successively add expressions with identifiers at lines 3, 4, 10, and 16. We use the expression methods that return expression method generators between lines 3 and 16.

At line 18, we generate a parser class with the name `SimpleMath`.

5.2.2 Expression Generators

Expression generator instances generate single expression methods. Those are generator objects that hold none, one or more generators as children. The generators without children generate expression methods for primitive expressions. Those

Listing 5.7: Simple Math Grammar Parser Generation

```
1  g := PEGParserGenerator new.
2
3  g defineRule: #expr withExpr: (g apply: #addexpr).
4  g defineRule: #addexpr withExpr: (
5      g alt: {
6          g seq: {g apply: #addexpr . g string: '+' . g apply: #
               mulexpr} .
7          g seq: {g apply: #addexpr . g string: '-' . g apply: #
               mulexpr} .
8          g apply: #mulexpr
9      }).
10 g defineRule: #mulexpr withExpr: (
11     g alt: {
12         g seq: {g apply: #mulexpr . g string: '*' . g apply: #digit
               } .
13         g seq: {g apply: #mulexpr . g string: '/' . g apply: #digit
               } .
14         g apply: #digit
15     }).
16 g defineRule: #digit withExpr: (g range: 0to :9).
17
18 g generate: #SimpleMath inheritFrom: PEGParser.
```

Table 5.1: Expression Generator Methods of the Parser Generator

Expression	Lexical Methods	Syntactical Methods
String	`string: <string>,`	
	`str: <string>`	
Character class	`range: <start> to: <end>`	
Any	`any`	
Optional	`opt: <generator>`	
Zero or more	`lexstar: <generator>,`	`synstar: <generator>`
	`star: <generator>`	
One or more	`lexplus: <generator>,`	`synplus: <generator>`
	`plus: <generator>`	
Lookahead And	`and: <generator>`	
Lookahead Not	`not: <generator>`	
Sequence	`lexseq: <generators>,`	`synseq: <generators>`
	`seq: <generators>`	
Alternatives	`lexalt: <generators>,`	`synalt: <generators>`
	`alt: <generators>`	
Apply	`lexapply: <ruleName>,`	`synapply: <ruleName>`
	`apply: <ruleName>`	

with one child are prefix- and suffix-expression generators. Collection-expression methods have generators that hold multiple children. Apply and apply wrapper expression generators have no children as well.

Each generator has the method

```
compile: <name> into: <class>
```

This method calls the same method on all its child generators recursively. This recursive approach requires that each generator object itself generates the method names for its children using its method name, adding suffixes. Each generator also generates the source code for the corresponding expression method and compiles it into the parser class.

The generation of source code works in different ways, depending on the type of expression that we generate.

5.2.2.1 Primitive Expressions

Primitive expression methods need to compare individual values against characters in the string to parse. It is helpful if those values are directly part of the method object. We generate the method string from a source code template. The template contains symbols, which are singleton objects in Squeak, that we use as placeholders for the actual expression attributes. After compiling said method template strings into method objects, we can check the method objects literals for the symbols and replace the actual values directly in the method object.

Example 5.4 (String Expression Generator)**.** In Listing 5.8, there is the method template for a rule named `string`.

Listing 5.8: String Expression Method Template

```
1 parse_string: pos
2     #matchString withIndexDo: [:character :index |
3         (character == (string at: (pos + index - 1) ifAbsent: [nil
              ]))
4         ifFalse: [
5             self reportFailureOf: #matchString atPos: pos.
6             ↑ nil] ].
7     ↑ PEGNode newOn: #_terminal from: pos to: (pos + #matchString
          size)
```

After compilation of the method, the literal `#matchString` is replaced by the actual match string in the method object.

5.2.2.2 Prefix- and Suffix-expressions

In the case of the prefix- and suffix-expressions, we use a method string template again.

In contrast to the previous case, we do not replace values after the compilation but before it. The placeholders come in the form of format string markers. It is necessary to fill in the child-expression name of each prefix- and suffix-expression method to call the child method.

We demonstrate such a code generation in the following example.

Example 5.5 (Optional Expression Generator)**.** Listing 5.9 contains the method template for an optional rule.

We replace placeholder `{1}` by the actual name of the child-expression method before the compilation of the method.

5.2.2.3 Collection-expressions

Collection-expression methods are a little more complicated to generate than the previous ones. For those, we need to concatenate the method source string using multiple template strings depending on the number of child expressions. In the concatenated string, we format the child-expression method names similar to the replacement for prefix- and suffix-expressions.

5.2.2.4 Apply and Apply Wrapper Expressions

We generate apply and apply wrapper methods very similarly to prefix- and suffix-expression methods. The difference is that we do not have child-expression genera-

Listing 5.9: Optional Expression Method Template

```
parse_opt: pos
    | childResult |
    childResult := self {1}: pos.
    ↑ childResult
        ifNil: [
            PEGNode
                newOn: #_terminal
                from: pos
                to: pos
                children: OrderedCollection new]
        ifNotNil: [
            PEGShadowNode
                newOn: #optional
                from: pos
                to: childResult endPos
                children: childResult nodeList]
```

tors that need a generated name. Instead, we have rule name references. We format these rule names into the method template, as well.

Example 5.6 (Apply Wrapper Generator)**.** In Listing 5.10, find the simple method source code template before replacing the name of the actual rule.

Listing 5.10: Lexical Apply Wrapper Method Template

```
parse_someRule_1: pos
    ↑ self {1}: pos
```

We replace `{1}` by `apply_<ruleName>` with `<ruleName>` being the identifier for the rule method the apply wrapper expression references. We independently generate the apply-expression method.

5.2.2.5 List of Generators

To give an overview of all available generators, we provide a list in Table 5.2.

Table 5.2: Complete List of all Expression Method Generators

Type	Generator Name
primitive	`PEGExprStringGenerator`
	`PEGExprRangeGenerator`
	`PEGExprAnyGenerator`
suffix	`PEGExprOptionalGenerator`
	`PEGExprLexicalZeroOrMoreGenerator`
	`PEGExprSyntacticalZeroOrMoreGenerator`
	`PEGExprLexicalOneOrMoreGenerator`
	`PEGExprSyntacticalOneOrMoreGenerator`
prefix	`PEGExprLookaheadGenerator`
	`PEGExprNotGenerator`
collection	`PEGExprLexicalSequenceGenerator`
	`PEGExprSyntacticalSequenceGenerator`
	`PEGExprAlternativesGenerator`
apply & apply wrapper	`PEGExprApplyGenerator`
	`PEGExprLeftRecursiveApplyGenerator`
	`PEGExprLexicalApplyWrapperGenerator`
	`PEGExprSyntacticalApplyWrapperGenerator`

5.2.3 Generation

The generator uses a method object for the generation of the actual parser. The method object class name is `PEGBuildParser`; we will refer to it as the parser builder.

We give a visual overview of the parser generation process in Figure 5.2. At first, the parser builder creates an empty parser class to which it can add the generated methods. After creating the class, it uses the rule definitions to build a derivation graph to find left-recursive rules. As described in section 4.3, this process also needs to take into account the rules of all ancestor grammars. Therefore we need a system to associate the parent classes with the respective rules. That is why we add a class instance variable `ruleDefinitions` that is a set of all rules of the corresponding grammar. On class generation, we add the set of those rules as static variable value to the class instance. From the derivation graph, the method extracts the left-recursive rule identifiers and the non-left-recursive rule identifiers and stores them in class instance variables as well, as it did for all superclass parsers.

The parser generator creates a left-recursive apply-expression method for each left-recursive identifier of a parser class that is not part of the parent class left-recursive identifier set. It also creates a non-left-recursive apply-expression method for each non-left-recursive identifier of a parser class that is not part of the non-left-recursive identifier set of the parent class.

If a parser class identifies a rule as left-recursive, but the parent class does not, that means that either the related rule is new or the inheritance changed the left-recursiveness of it. Either way, we need to generate a new left-recursive apply method.

The generation of all other expression methods, including the apply wrapper methods, happens by recursively calling the expression method generators and passing the parser class as a parameter.

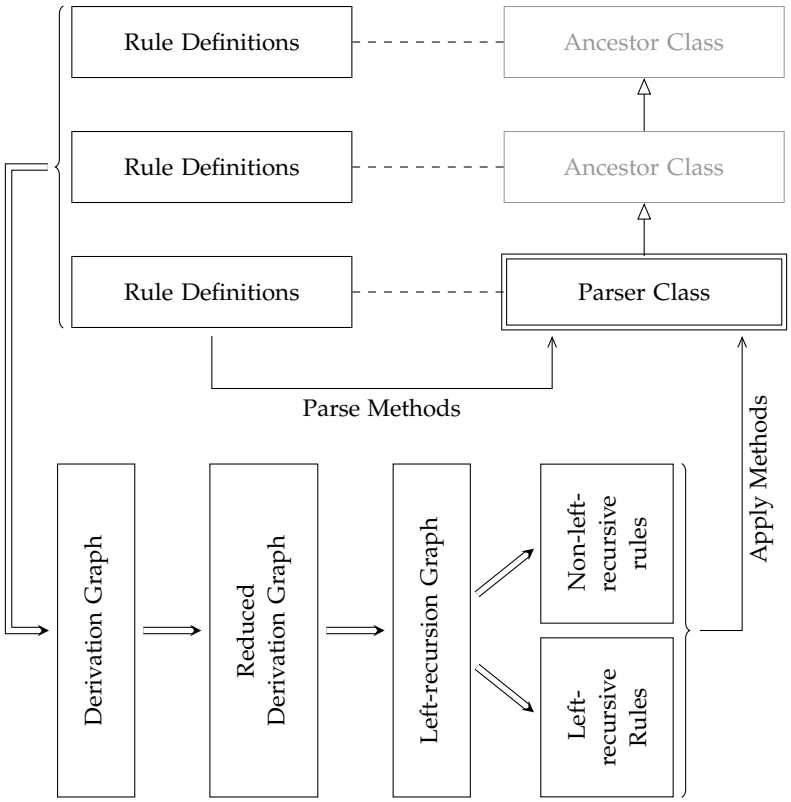


Figure 5.2: Overview of the Parser Generation Process

5.3 Ohm Grammar Converter

To make use of already existing Ohm grammars, we need to translate Ohm grammar objects to parser generators that again generate parser classes.

For that, we use the interface of the parser generator class. We can incorporate the inheritance of Ohm grammars by always generating the whole superclass stack for any Ohm grammar object to make sure it exists. We can do this by recursively converting the respective parent grammars before converting the grammar

itself. The grammar converter walks through the rules and expressions of an Ohm grammar and converts them recursively to expression method generators. Walking recursively through them works because there are no reference cycles of object pointers in Ohm expressions. For each rule, the converter processes, whether the rule is syntactical or lexical, and generates the subexpression method generators according to the stored type.

Ohm has four kinds of rules: `OhmRuleDefine`, `OhmRuleOverride`, `OhmRuleInline` and `OhmRuleExtend`. We consider the first three kinds of rules to be equal for the translation process. They only handle checks for intentional overrides. The fourth kind of rule is different. It creates an extra alternative for an existing rule, meaning that we need to prepend the expression to an existing alternatives-expression or create a new alternatives-expression with two options.

6 Evaluation

To evaluate the parser generator, we conduct multiple benchmarks. In the first part of this chapter, we describe and motivate our benchmark setups and document the results. In the second part, we discuss the comparability of our benchmarks and have a brief look at the shortcomings and weaknesses of the generated parser.

6.1 Benchmarking

We benchmark using different grammars, focusing on two aspects: We perform non-left-recursive parsing by using a grammar without left-recursive rules. Additionally, we parse a left-recursive grammar that contains a lot of left-recursive rules.

As the non-left-recursive grammar, we choose the Smalltalk grammar from the Ohm/S package and parse typical Smalltalk methods. To handle the most typical methods, we parse all method source strings within the clear Squeak image.

To benchmark a left-recursive grammar, we define a grammar that can parse arithmetical formulas. For the dataset of strings to parse, we generate formula strings using this grammar to parse them afterward with the same grammar.

We choose these datasets so that parses would take at least several seconds to run to minimize the error produced by the Squeak virtual machine. We also choose them to mimic real use cases for the parser.

We also need to evaluate the duration of the generation and compilation of a parser to get an estimation if our grammar tooling is useful in a live programming environment. In this case, we choose to convert the Smalltalk grammar that is part of the Ohm package, as well.

We measured all times without garbage collection and conducted all experiments 20 times to minimize random variable influence. We calculate the mean and standard deviation for the measurement results.

6.1.1 System Description

We conduct all our experiments on a Windows 10 machine with an Intel i5-6299U processor, running a Squeak image in the Squeak virtual machine.

A comprehensive list of system specifications is available in Table 6.1.

For the experiments, we stopped all other user-invoked processes in the operating system and deactivated Windows virus protection. Also, we deactivated the network connection of the machine to eliminate background computation related to network activity.

Table 6.1: System and Environment Specs

Hardware	Attribute	
CPU	Vendor	Intel
	Model	i5-6299U
	Clock rate	2.30Ghz
	Number of cores	4
RAM	Memory	8GB
	Type	DDR3
	Speed	1600 MHz
Mainboard	Vendor	Lenovo
	Productno.	20FNCTO1WW
Software	**Attribute**	
OS	Name	Windows 10
	Version	1903
	Build	18362.418
	Architecture	x64
Environment	Name	Squeak/Smalltalk
	Version	5.2-18229
	Architecture	x64
Ohm/S	Repository	`https://github.com/hpi-swa/Ohm-S`
	Commit	3fc87d7

To get inline-compilation for the loop inside the benchmark block, we use the method `to:do:` of the `Number` class.

6.1.2 Non-Left-recursive Grammar

As mentioned above, we use a Smalltalk grammar to parse all method source strings of all classes in a plain Squeak image build 5.2-18229. The image consists of 2713 classes with a total of 50697 methods. With this benchmark, we mimic the use case of recompiling the whole Squeak image during development.

As a benchmark preparation step, we extract all method strings and add them into a list of tuples in which each tuple consists of a class reference and a method string. The preparation should minimize the lookup times for method source strings during parse time measurements.

In Listing 6.1, we show as an example the parsing invocation to get one of the twenty results for the Squeak parser class.

Listing 6.1: Smalltalk Benchmark Parser Invocation

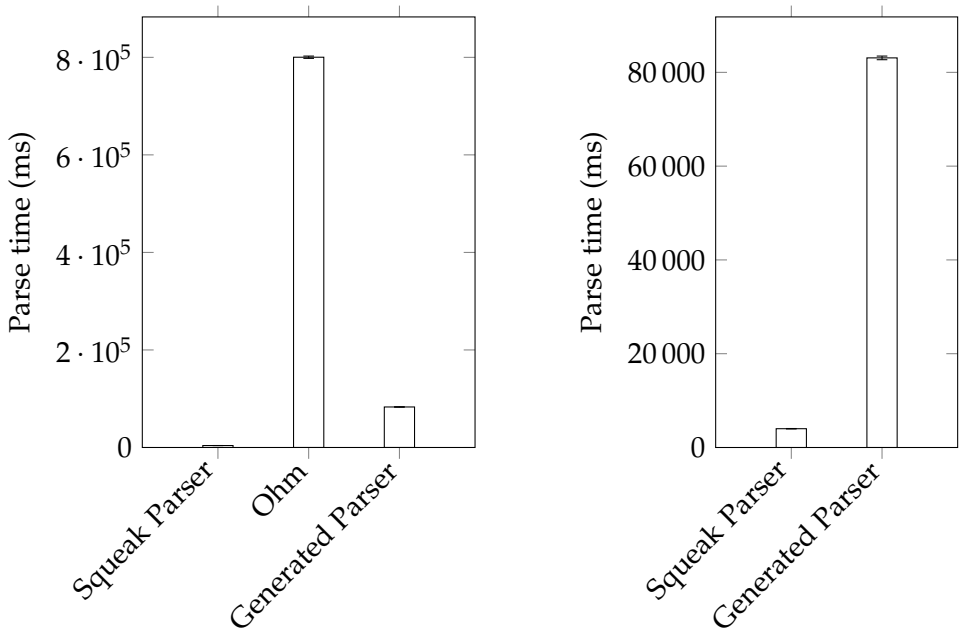
```
1 [1 to: methodStrings size do: [:i |
2     | parser item |
3     parser := Parser new.
4     item := methodStrings at: i.
5     parser parse: (item at: 2) class: (item at: 1)
6     ]] timeToRunWithoutGC
```

The results for all three parsers are reported in Table 6.2 as well as in Figure 6.1.

Table 6.2: Non-left-recursive Parse Times

Parser	Mean	Standard Deviation
Squeak Parser	4007.6 ms	42.7 ms
Ohm	800230.5 ms	2358.8 ms
Generated Parser	83083.0 ms	389.1 ms

The figure shows the average parse times in milliseconds as well as the standard deviation intervals.



(a) All Parsers **(b)** Parsers without Ohm

Figure 6.1: Non-left-recursive Benchmark with Smalltalk Grammar

6.1.3 Left-recursive Grammar

To benchmark left-recursion, we define the following highly left-recursive mathematical grammar. It supports the semantics of the arithmetic operations addition, subtraction, multiplication, and division as well as exponentiation.

We generated 100,000 mathematical formulas randomly. We produced them by using the grammar to be benchmarked as a generative grammar. Starting with rule `expr`, we built a tree top-down. Sequence-expression generators generated the concatenation of the generation of all their subexpressions. Zero-or-more-expressions used their subexpressions for every additional iteration with a $\frac{2}{3}$ chance. One-or-more-expressions did the same with the difference that they used the subexpression at least once. String-expressions generated just their match string. Range-expression generation resulted in a character picked uniformly at random from all characters within the range. Alternatives-expressions could not choose an option uniformly at random due to the recursive nature of the grammar; this would have resulted in infinitely large trees. For alternative-expressions, we picked the last option, which in our case is always a hierarchy jump, with a $\frac{2}{3}$ chance, and another option uniformly at random for the remaining $\frac{1}{3}$ cases.

We got strings with an average size of around 30.8 and a median size of 15, for example, the following formula string.

```
"2/.8675583/-(19.08806)/29/9231^-8*900542*.9659627"
```

Listing 6.2: Math Grammar

```
expr := addexpr

addexpr := addexpr_plus | addexpr_minus |
   mulexpr
addexpr_plus := addexpr '+' mulexpr
addexpr_minus := addexpr '-' mulexpr

mulexpr := mulexpr_times | mulexpr_divide |
   expexpr
mulexpr_times := mulexpr '*' expexpr
mulexpr_divide := mulexpr '/' expexpr

expexpr := expexpr_power | priexpr
expexpr_power := priexpr '^' expexpr

priexpr := priexpr_paren | priexpr_pos |
   priexpr_neg | number
priexpr_paren := '(' expr ')'
priexpr_pos := '+' priexpr
priexpr_neg := '-' priexpr

number := number_frac | number_int
number_frac := digit* '.' digit+
number_int := digit+

digit := [0-9]
```

We present the results in Table 6.3.

In Figure 6.2, you can see the average parse times in milliseconds as well as standard deviation.

Table 6.3: Left-recursive Parse Times

Parser	Mean	Standard Deviation
Ohm	80731.6 ms	548.8 ms
Generated Parser	18558.3 ms	732.6 ms

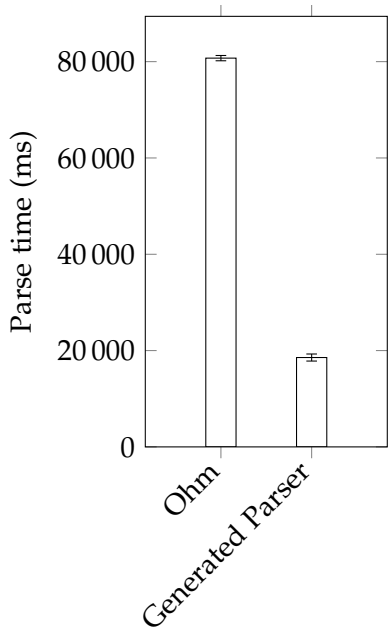


Figure 6.2: Math Benchmark

6.1.4 Grammar Generation

To check if our grammar generation is viable for a live programming environment, we need to get a perception of the timeframe of grammar generation.

During the development of custom grammars, one does change the grammar quite often. As described in section 4.3, any rule can influence the left-recursiveness of all other rules. Therefore, the whole left-recursion detection pipeline needs to

run for every change. To have an instantly available parser, we always need to compile the parser class as well.

To test the runtime of grammar conversion, we choose to convert the moderately sized Smalltalk grammar provided by Ohm. The grammar incorporates inheritance with two ancestor grammars, which we include in our measurements. That means that all parser generation steps are executed for all three involved classes.

After 20 runs, we achieved a mean conversion time of 162.5 ms and a standard deviation of 7.1 ms.

6.2 Discussion

In this section, we discuss the decisions we had to make for the benchmark setups. We go into the used grammars and our methodology, including possible errors and shortcomings. We also motivate our decisions despite those issues. Furthermore, we describe potential improvements for our parser generator implementation.

6.2.1 Data

We tested one non-left-recursive grammar in the form of the Smalltalk grammar parsing the methods of the Squeak image and one highly left-recursive grammar by means of a self-defined math grammar that incorporates arithmetic operations.

We could have used more non-left-recursive grammars of other programming languages, but parsing Smalltalk is the closest use case and most practical within Squeak.

For the highly left-recursive language, we constructed arbitrary strings to parse. The strings might be too long to represent real used formulas. We could have used a large dataset like GitHub repositories to extract real used formulas; this, however, would have come with two downsides. First, there is the possibility that we could not have been able to parse all extracted formulas with our self-constructed grammar. Second, it is very context-dependent, and therefore hardly generalizable how long and how complex formulas are.

We also did not test a grammar that has only left-recursive rules, because those have no real use case for formal language descriptions.

We also have no grammar that incorporates hidden left-recursion or big indirect left-recursive cycles because they are hard to find and to construct. Also, most programming language definitions do not have indirect or hidden left-recursion.

6.2.2 Methodology

In addition to measuring the actual parsing algorithm times, our measurements include the overhead times each parser needs to set up at the beginning of a parse. For the generated parsers and the Ohm parsers, that is the time it takes to set up the data structures for the memoization and left-recursion handling. For the

Squeak parser, it is the time to initiate a new instance because the parser generates state during parses but does not remove it.

We measured all times while excluding the times that the garbage collector needs. That is not entirely representative because some parse strategies might require more garbage collector time than others due to the different amounts of temporary data that we gather during parsing.

We calculated the standard deviation for all measurements but no confidence intervals; this is because the standard deviation is microscopic, and the measurement results for different parsers are very far apart. We measured each case 20 times on one machine. That should be enough, considering the small standard deviation. The number of runs should be high enough to eliminate random, non-controllable variables mostly.

We also can not rule out that just-in-time compilation altered the measured results partially. A large number of runs with different data should have proactively eliminated that problem because, after a short time, any code should reach a state that is close to a stable state. Also, experiments showed that there where no performance increases using any of the three parsers even when parsing the same input a couple of thousand times consecutively.

We also did not measure the performance of the grammars themselves. The grammars are not optimized for the fastest parsing. It might be possible to parse the same languages with different grammars faster. That would, on the other hand, result in different syntax trees and change the semantic meaning of nodes.

6.2.3 Potential Parser Shortcomings

Despite putting much effort into the parser generator, there remain potential downsides.

Flattening the syntax trees can be expensive for large expressions with deep nesting of subexpressions. For those expressions, the parser flattens the arrays multiple times resulting in a large number of memory allocations and list traversals. A possible solution for this issue could be to flatten the tree post parsing. This would allow for allocating memory only once while flattening and moving every object only once.

We also did not address name conflicts with the named subexpressions. Creating two rules, named `myrule` and `myrule_1` can lead to a name conflict within the generated parser. The parser generator does not check this case, which can lead to unexpected behavior of generated parsers.

7 Related Work

The main challenge for creating a PEG parser is to make the parsing process as fast as possible while handling left-recursion correctly. There are different approaches. Some eliminate left-recursion entirely while relying on some assumptions about the input grammar. Others use different algorithms or grammar specifications. We present the approaches closest to our problem statement.

7.1 Modifying PEG Capabilities

All the following papers emphasized that left-recursion is a desirable feature for PEGs and therefore try to handle it as well as possible. In contrast to our generated parsers, however, they modified the input grammar semantics to have more accessible parsing instructions.

Parsing Expression Grammars Made Practical The authors of this paper propose a PEG library that supports all left-recursion types. [12] They introduce an annotation to mark left-recursion and precedence of rules in PEGs. Furthermore, they use a concept called expression cluster to parse the annotated left-recursive rules. The core of their method includes reapplying rules on themselves, just like the Warth-algorithm.

One can argue that this method of additional annotations creates a very human-readable form of grammars, and the creators can gain a thorough understanding of their grammars. Unfortunately, the users need to make the annotations manually instead of generating them automatically.

A Programming Language Where the Syntax and Semantics Are Mutable at Runtime In this thesis, the author develops a programming language called *Kathadin*. [19] The *Kathadin* interpreter executes programs according to the language specification, which can be modified during runtime. The interpreter only defines a minimal language, which allows the definition of more complex language features.

The created parser uses longest-match parsing instead of prioritized alternatives like PEG specifies. The author argues that the PEG parse strategy is greedy and that longest-match conforms to this strategy.

Kathadin uses annotations as well for left-recursive, right-recursive, and non-recursive definitions. The parser is a modified packrat parser that uses these annotations. It stores annotated rules in a data structure during parsing. This way, it can abort the non-recursive rule-applies. For left-recursive rules, the parser also uses a seed growing approach by reapplying the parsed rule to itself until it fails.

Left Recursion in Parsing Expression Grammars In this paper, the idea of bounded left-recursion is presented. [14] The authors propose to limit the left-recursion rule invocations for every single rule and therefore forcing a guaranteed termination. They also find for every number of left-recursive invocations the invocation number that is lowest with the longest match. This approach solves left-recursion on the one hand but limits the expressiveness of the grammar on the other hand.

Practical Dynamic Grammars for Dynamic Languages The author presents the PEG parsing framework *PetitParser*. It is a parser generator framework for Pharo/Smalltalk. [18] It uses the packrat parsing algorithm to parse PEGs. It allows one to reuse, compose, and transform grammars while utilizing the existing Smalltalk language for grammar definition.

The author states that the performance of *PetitParser* is better than table-based parsers for carefully written grammars. Its performance comes from memoizing only selected rules which the user has to choose.

7.2 Restricting PEG Capabilities

The following two works use the approach to rewrite PEGs. As we showed, direct left-recursive rewriting, in contrast to indirect and hidden left-recursion rewriting, is a manageable task. Therefore the authors decided on limiting their parsing frameworks to these limited grammars, which they can rewrite.

Better Extensibility through Modular Syntax In this work, the parser generator *Rats!* is presented. [9] It supports all PEG capabilities of the original definition and implements packrat parsing using memoization, ensuring linear parse time. The parser allows direct left-recursive rules, which it automatically transforms into right-iterative production rules. However, it restricts the use of arbitrary left-recursion. Therefore it has the same expressiveness as any PEG but cannot represent arbitrary semantic information.

Adaptive LL(*) Parsing: The Power of Dynamic Analysis In this paper, the authors describe the parser generator *ANTLR* that uses CFGs with a syntax similar to BNF and EBNF as input. [16] Like *Rats!*, it forbids indirect and hidden left-recursion. It also eliminates direct left-recursion by rule rewriting, which can lead to grammars that are exponential in size.

While restricting their grammar definition capabilities, they allow boolean evaluation using the host language to determine the semantic viability of productions.

8 Conclusion

In this chapter, we describe the achievements of this report. In particular, we highlight the benefits of our parser generator in contrast to other approaches. Also, we describe further research topics that can be of interest to improve PGE parser in the future.

8.1 Conclusion

Even though a lot of effort went into PEG parsing, it is still not a completely solved problem. Parsing of left-recursion constitutes a difficulty with the potential for improvement. Although we were able to narrow the problem of left-recursive parsing down a bit.

8.1.1 Rewriting Parsing Expression Grammars

We depicted the problems that occur when trying to rewrite arbitrary left-recursive PEGs. We developed a method to find any potential left-recursive cycles and described an algorithm that can find all expressions causing left-recursion. We showed that rewriting arbitrary left-recursive grammars can result in vast and convoluted rule sets, which reduces the comprehensibility of those grammars very much.

8.1.2 Constructing a Parser Generator

As a practical approach, we designed and implemented a PEG parser generator that creates parser classes. Generated parser class instances can handle left-recursion, which they can detect during their generation process. This way, users of this parser generator do not need to worry about left-recursive rules and use them without being aware of their left-recursiveness.

8.1.2.1 Performance

We benchmarked the parser for two contrasting cases. The standard deviations for the benchmark results were pretty small, which gives us high confidence in the validity of our measurements.

For the non-left-recursive Smalltalk grammar, we benchmarked against Ohm and the handwritten Squeak parser by parsing all methods in a clear Squeak image. Our parser was 9.6 times faster than Ohm and, therefore, nearly one order of magnitude better. Testing against the very efficient Squeak parser, we were 20.7 times slower.

Problematic for our parser is the use of grammars as we established early on; this can result in large parse trees for lexing alone as well as parse trees that it has to discard during parsing.

For the highly left-recursive self-defined math grammar that represents arithmetic expressions, we benchmarked the parse of a large number of generated strings. We achieved a 4.4 times better parsing speed than Ohm using the same grammar. In this case, preprocessing did not add that much value due to the high number of left-recursive rules. These could not be excluded from left-recursion handling, which is time expensive.

8.1.2.2 Liveness

We tested if we can use our tool in a live programming environment. According to our adopted definition in section 2.5, there should not be any noticeable interruption. The conversion of a more extensive Ohm grammar to a parser generator takes far less than half a second, and we can, therefore, consider it as immediate. Note that, in fact, during this process, we converted three grammars including their inheritance chain, and all constant overheads of graph setup and analytics happened three times.

In our liveness definition, we also speak of the necessity to permit editing the program while it is running and the execution continuing right after editing. All our parses of single strings happen in the timeframe of milliseconds; therefore, there is no real need to modify the parser during parses. It would even be possible to halt the program during parsing and apply changes to its generating grammar. This, however, would lead to unexpected behavior, and therefore should be avoided.

We consider the parser generator, and with it, the generated parsers as suitable for live programming.

8.2 Future Work

Although we were able to create a fast parser generator; there is remaining work to do, and new research opportunities open up, which could furthermore increase a parser generator's performance.

In this section, we discuss some of those potential improvements.

8.2.1 Memoization

We discovered that for the case of parsing Smalltalk methods, memoization could improve the performance, although some research states that it often leads to a performance decrease. To find out when exactly memoization is desirable holds potential for improvement.

The challenge is to find out which rules are worth memoizing. It is necessary to find metrics that measure the probability a parser invokes a rule multiple times at the same position. Existing probability graph models like the Page-algorithm

or Markov-chains or more extensive grammar graph analytics could be useful to solve this problem.

8.2.2 Grammar Analysis

We established that lexing using a lookup table for character grouping is a rapid method. To incorporate this into a grammar-based parser, one could perform grammar analysis to find out which leaf nodes are in unambiguous categories and use those categories to lookup characters faster instead of building syntax subtrees for single characters. The parser could precalculate the subtrees for every character, just adding them to the syntax tree when needed.

This method potentially saves calculation time for whole lexing subtrees, and leaves us only with parsing the syntactical part of any grammar using the Warth-algorithm.

8.2.3 Memoization Datastructure

Currently, our parser generator memoizes via a lookup in an array of identity dictionaries. This method proved to be the most efficient for looking up entries with integer-string-tuples in Squeak.

In the original packrat parsing paper, a two-dimensional array was used [5]. Our parser could also use a bidirectional mapping between natural numbers and rule identifiers such that we can look up the rules in a two-dimensional array. This could be implemented by creating a mapping before the conversion such that we can integrate the mapped numbers into our compilation process.

This method would also be inheritance stable since, for every child class, we only would need to increase the range of mapped integers depending on the new rule count.

8.3 Summary

We showed that grammar rewriting is impractical when reducing parse times due to the high complexity of required rules.

We managed to design and implement a parser generator that uses left-recursion detection before parsing, and we showed that it increases parsing speed. The required preprocessing is efficient enough that the generator can be used in a live programming environment.

Bibliography

[1] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. "Report on the Algorithmic Language ALGOL 60". In: *Communications of the ACM* 3.5 (May 1960). Edited by P. Naur, pages 299–314. ISSN: 0001-0782. DOI: 10.1145/367236.367262.

[2] R. Becket and Z. Somogyi. "DCGs + Memoing = Packrat Parsing but is It Worth It?" In: volume 4902. Springer. 2008, pages 182–196. DOI: 10.1007/978-3-540-77442-6_13.

[3] J. Earley. "An Efficient Context-free Parsing Algorithm". In: *Communications of the ACM* 13.2 (Feb. 1970), pages 94–102. ISSN: 0001-0782. DOI: 10.1145/362007.362035.

[4] B. Ford. "Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking". Master's thesis. Massachusetts Institute of Technology, 2002.

[5] B. Ford. "Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl". In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '02. Pittsburgh, PA, USA: ACM, 2002, pages 36–47. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581483.

[6] B. Ford. "Parsing Expression Grammars: A Recognition-based Syntactic Foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: ACM, 2004, pages 111–122. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964011.

[7] R. Frost, R. Hafiz, and P. Callaghan. "Parser Combinators for Ambiguous Left-Recursive Grammars". In: volume 4902. Springer. 2008, pages 167–181. DOI: 10.1007/978-3-540-77442-6_12.

[8] A. Goldberg and D. Robson. *Smalltalk-80. The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-11371-6.

[9] R. Grimm. "Better Extensibility Through Modular Syntax". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. Ottawa, Ontario, Canada: ACM, 2006, pages 38–51. ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1133987.

[10] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd edition. Pearson Education India, July 2008. ISBN: 978-0321462251.

[11] K. Kuramitsu. "Packrat Parsing with Elastic Sliding Window". In: *Journal of Information Processing* 23.4 (July 2015), pages 505–512. DOI: 10.2197/ipsjjip.23.505.

[12] N. Laurent and K. Mens. "Parsing Expression Grammars Made Practical". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2015. Pittsburgh, PA, USA: ACM, 2015, pages 167–172. ISBN: 978-1-4503-3686-4. DOI: 10.1145/2814251.2814265.

[13] F. Mascarenhas, S. Medeiros, and R. Ierusalimschy. "On the Relation between Context-Free Grammars and Parsing Expression Grammars". In: *Science of Computer Programming* 89 (Apr. 2013). DOI: 10.1016/j.scico.2014.01.012.

[14] S. Medeiros, F. Mascarenhas, and R. Ierusalimschy. "Left recursion in parsing expression grammars". In: *Science of Computer Programming* 96 (2014), pages 177–190.

[15] R. C. Moore. "Removing Left Recursion from Context-free Grammars". In: *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference*. NAACL 2000. Seattle, Washington: Association for Computational Linguistics, 2000, pages 249–255.

[16] T. Parr, S. Harwell, and K. Fisher. "Adaptive LL(*) Parsing: The Power of Dynamic Analysis". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pages 579–598. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660202.

[17] P. Rein, R. Hirschfeld, and M. Taeumel. "Gramada: Immediacy in Programming Language Development". In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2016. Amsterdam, Netherlands: ACM, 2016, pages 165–179. ISBN: 978-1-4503-4076-2. DOI: 10.1145/2986012.2986022.

[18] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. "Practical Dynamic Grammars for Dynamic Languages". In: (2010).

[19] C. Seaton. "A Programming Language Where the Syntax and Semantics Are Mutable at Runtime". Master's thesis. University of Bristol, 2007.

[20] S. L. Tanimoto. "A Perspective on the Evolution of Live Programming". In: *Proceedings of the 1st International Workshop on Live Programming*. LIVE '13. San Francisco, California: IEEE Press, 2013, pages 31–34. ISBN: 978-1-4673-6265-8.

[21] R. Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *Foundations of Computer Science, IEEE Annual Symposium on* 0 (Nov. 1971), pages 114–121. DOI: 10.1109/SWAT.1971.10.

[22] A. Warth, J. R. Douglass, and T. Millstein. "Packrat Parsers Can Support Left Recursion". In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '08. San Francisco, California, USA: ACM, 2008, pages 103–110. ISBN: 978-1-59593-977-7. DOI: 10.1145/1328408.1328424.

[23] A. Warth, P. Dubroy, and T. Garnock-Jones. "Modular Semantic Actions". In: *Proceedings of the 12th Symposium on Dynamic Languages*. DLS 2016. Amsterdam, Netherlands: ACM, 2016, pages 108–119. ISBN: 978-1-4503-4445-6. DOI: 10.1145/2989225.2989231.

[24] N. Wirth. "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?" In: *Communications of the ACM* 20.11 (Nov. 1977), pages 822–823. ISSN: 0001-0782. DOI: 10.1145/359863.359883.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
134	978-3-86956-502-6	**Interval probabilistic timed graph transformation systems**	Maria Maximova, Sven Schneider, Holger Giese
133	978-3-86956-501-9	**Fast packrat parsing in a live programming environment : improving left-recursion in parsing expression grammars**	Friedrich Schöne, Patrick Rein, Robert Hirschfeld
132	978-3-86956-482-1	**SandBlocks : Integration visueller und textueller Programmelemente in Live-Programmiersysteme**	Leon Bein, Tom Braun, Björn Daase, Elina Emsbach, Leon Matthes, Maximilian Stiede, Marcel Taeumel, Toni Mattis, Stefan Ramson, Patrick Rein, Robert Hirschfeld, Jens Mönig
131	978-3-86956-481-4	**Was macht das Hasso-Plattner-Institut für Digital Engineering zu einer Besonderheit?**	August-Wilhelm Scheer
130	978-3-86956-475-3	**HPI Future SOC Lab : Proceedings 2017**	Christoph Meinel, Andreas Polze, Karsten Beins, Rolf Strotmann, Ulrich Seibold, Kurt Rödszus, Jürgen Müller
129	978-3-86956-465-4	**Technical report : Fall Retreat 2018**	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich, Erwin Böttinger, Christoph Lippert
128	978-3-86956-464-7	**The font engineering platform : collaborative font creation in a self-supporting programming environment**	Tom Beckmann, Justus Hildebrand, Corinna Jaschek, Eva Krebs, Alexander Löser, Marcel Taeumel, Tobias Pape, Lasse Fister, Robert Hirschfeld
127	978-3-86956-463-0	**Metric temporal graph logic over typed attributed graphs : extended version**	Holger Giese, Maria Maximova, Lucas Sakizloglou, Sven Schneider
126	978-3-86956-462-3	**A logic-based incremental approach to graph repair**	Sven Schneider, Leen Lambers, Fernando Orejas
125	978-3-86956-453-1	**Die HPI Schul-Cloud : Roll-Out einer Cloud-Architektur für Schulen in Deutschland**	Christoph Meinel, Jan Renz, Matthias Luderich, Vivien Malyska, Konstantin Kaiser, Arne Oberländer

ISBN 978-3-86956-503-3
ISSN 1613-5652