Parsing For Agile Modeling

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Jan Kurš
von Tschechien

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Institut für Informatik

Parsing For Agile Modeling

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Jan Kurš
von Tschechien

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Institut für Informatik

Von der Philosophisch-naturwissenschaftlichen Fakultät
angenommen.

Bern, 25.10.2016

Der Dekan:
Prof. Dr. Gilberto Colangelo

Abstract

Agile modeling refers to a set of methods that allow for a quick initial development of an importer and its further refinement. These requirements are not met simultaneously by the current parsing technology. Problems with parsing became a bottleneck in our research of agile modeling.

In this thesis we introduce a novel approach to specify and build parsers. Our approach allows for expressive, tolerant and composable parsers without sacrificing performance. The approach is based on a context-sensitive extension of parsing expression grammars that allows a grammar engineer to specify complex language restrictions. To insure high parsing performance we automatically analyze a grammar definition and choose different parsing strategies for different parts of the grammar.

We show that context-sensitive parsing expression grammars allow for highly composable, tolerant and variable-grained parsers that can be easily refined. Different parsing strategies significantly insure high-performance of parsers without sacrificing expressiveness of the underlying grammars.

Contents

1 Introduction 8

1.1 Agile Modeling . . . 9

1.2 Parsing Obstacles of Agile Modeling . . . 12

1.3 Thesis . . . 13

1.4 Our Contribution . . . 13

2 Overview of Parsing Technologies 15

2.1 Parsing in the Wild . . . 15

2.1.1 Expressive Power . . . 15

2.1.2 Composability . . . 17

2.1.3 Tolerant Grammars and Semi-Parsing . . . 18

2.1.4 Performance . . . 19

2.1.5 Parsing Frameworks . . . 20

2.2 Existing Limitations . . . 22

2.3 Our Solution . . . 23

3 Parsing Expression Grammars and PetitParser 24

3.1 Parsing Expression Grammars . . . 24

3.1.1 PEG Analysis . . . 25

3.1.2 Parser Combinators . . . 26

3.2 PetitParser . . . 29

4 Context Sensitivity in Parsing Expression Grammars 32

4.1 Motivating Example . . . 33

4.2 Parsing Contexts . . . 34

4.2.1 Context-Sensitive Extension . . . 35

4.2.2 Indentation Stack . . . 35

4.3 Parsing Contexts in Parsing Expression Grammars . . . 37

4.3.1 Parser Combinators . . . 40

4.3.2 CS-PEG analysis . . . 40

4.4 Implementation . . . 43

4.4.1 Performance . . . 44

4.5 Case Studies . . . 47

4.5.1 Python . . . 47

4.5.2 Markdown . . . 49

4.6 Related Work . . . 52

4.7 Conclusion . . . 54

5 Semi-Parsing with Bounded Seas 55
5.1 Motivating Example 56
5.1.1 Why not use Regular Expressions? 56
5.1.2 A Naïve Island Grammar 57
5.1.3 An Advanced Island Grammar 57
5.2 Bounded Seas 59
5.2.1 The Sea Boundary 60
5.2.2 The Context Sensitivity of Bounded Seas 61
5.3 Bounded Seas in Parsing Expression Grammars 62
5.3.1 The *Water* Operator 63
5.3.2 The *NEXT* function 67
5.3.3 BS-PEG analysis 71
5.4 Implementation 72
5.4.1 Performance 73
5.5 Java Parser Case Study 73
5.5.1 Without Nested Classes 75
5.5.2 With Nested Classes 75
5.5.3 With Return Types 76
5.5.4 Performance 77
5.6 Related Work 79
5.7 Conclusion 82

6 Adaptable Parsing Strategies 83
6.1 Motivating Example 84
6.1.1 Composition Overhead 86
6.1.2 Superfluous Intermediate Objects 86
6.1.3 Backtracking Overhead 87
6.1.4 Context-Sensitivity Overhead 87
6.2 A Parser Combinator Compiler 88
6.2.1 Adaptable Strategies 88
6.3 Parser Optimizations 90
6.3.1 Regular Optimizations 92
6.3.2 Context-Free Optimizations 94
6.3.3 Context-Sensitive Optimizations 96
6.4 Performance analysis 100
6.4.1 PetitParser compiler 100
6.4.2 Benchmarks 101
6.4.3 Parsing Strategies Impact 103
6.4.4 Scanner Impact 105
6.4.5 Memoization Impact 107
6.4.6 Java Parsers Comparison 108
6.4.7 Smalltalk Parsers Comparison 108
6.5 Related Work 109
6.6 Conclusion 110

7 Ruby Case study 112
7.1 Ruby Structure 112
7.1.1 The Dangling End Problem 113
7.1.2 Measurements 115
7.2 Ruby Method Calls 116

7.2.1 Measurements 117

7.3 Performance 119

7.4 Conclusion 121

8 Conclusion 123

A Formal development of PEGs 136

B Bounded Seas Examples 141

B.1 Example of Dynamic *NEXT* computation 141

B.2 Example of Static *NEXT* computation 142

B.3 Overlapping Seas Example 147

C Implementation 152

C.1 Bounded seas 152

D Layout Sensitivity in the Wild 159

D.1 Haskell 159

D.2 Python 160

D.3 F# 161

D.4 YAML 162

D.5 OCaml 162

D.6 CoffeeScript 163

D.7 Grace 163

D.8 SRFI 49 — Indentation-Sensitive Scheme 164

D.9 Elastic Tabstops 164

E Scanner 166

E.1 Scanners in PEG-based parsers 166

E.1.1 Tokens and Scannable Parsing Expressions 166

E.1.2 Scannable Choices 167

E.1.3 Scanner 167

E.2 Regular Parsing Expressions 170

E.3 Regular Parsing Expression Languages 172

E.4 Finite State Automata 175

E.4.1 Construction of finite state automata from regular parsing expressions ($\mathcal{FSA}$) 178

E.4.2 Determinization of the automata with epsilons and priorities ($\mathcal{D}$) 181

F Measurements 183

F.1 Summary 183

F.2 Strategies Details 187

F.2.1 Expressions 190

F.2.2 IS Expressions 191

F.2.3 CF Python 192

F.2.4 Python 193

F.2.5 Smalltalk 194

F.2.6 Java 195

F.2.7 Java Sea 196

F.3 Scanner Impact 197

F.3.1 Expressions 199

F.3.2 Smalltalk 200
F.4 Memoization Details 201
F.5 Smalltalk Parsers 202
F.6 Java Parsers 203

G CommonMark Grammar Definition 204

1 Introduction

It is widely accepted that software developers spend more time reading code than writing it [LVD06]. Reading code not only promotes program comprehension, but helps developers understand the impact of their changes on the existing system. Nevertheless there are numerous questions developers ask that cannot simply be answered by reading code, such as *"which code implements this feature?"*, or *"what is the impact of this change?"*, and for which dedicated analyses are needed [SMDV06, NL12].

Dedicated platforms exist to model and analyze software systems, such as Moose [NDG05] and Rascal [KvdSV09]. A prerequisite for using such tools, however, is that a *model importer* exists for the programming language (or languages) in which the system is developed. Constructing a model importer from scratch is a major effort, and the large up-front investment hampers initiatives for many commercial tool builders [LV01, BBC+10]. Adapting an existing parser for the host language is often not an option especially for proprietary and legacy languages, or for sources mixed from different languages.

We use the term *agile modeling* to refer to a set of methods that allow for a quick initial development of an importer and its further refinement. The goal of agile modeling is to *build a coarse model of a software system in the morning, analyze it in the afternoon, and refine it the next day.* The repeated refinement eventually leads to a complete model built through a series of more and more fine-grained models. The methods of agile modeling utilize specific properties of programming languages, their similarities and expert knowledge to achieve the goal of agile modeling.

Agile modeling raises many questions, such as *"what entities are in the source code?"*, *"what are efficient ways to discover the mapping between code and model entities?"*, *"how can we leverage expert knowledge in the least intrusive way?"*, or . *"what is a suitable parsing technology to quickly develop a parser?"*. This work focuses on this last question — the parsing aspect of agile modeling.

In this chapter we inspect in detail agile modeling and its goals; we investigate the requirements for a parser that can support agile modeling; and in the end we briefly describe our contribution to parsing from the perspective of agile modeling.

1.1 Agile Modeling

By agile modeling we refer to the ability to (i) quickly build a coarse model of a software system suitable for an initial analysis; and (ii) easily refine the model for subsequent and more detailed analyses. To achieve its goals, agile modeling trades precision for time. We believe that initial and quick construction of a model is feasible for the following reasons:

- Models do not need to be precise because an imprecise model can suffice for many analyses. It takes less effort to specify an imprecise model importer than a complete and precise one.
- Model constraints are known in advance. Each analysis operates on entities and relations that are known beforehand. Furthermore, many programming languages share common concepts and features such as control structures or programming abstractions.
- Model importers can be trained. In any given project, a considerable quantity of existing and valid code is available as *"training data"*. The common features together with readily available knowledgeable developers can be used to guide a training process.

Additionally, we believe that the refinement of a model for more detailed analyses is feasible for the following reasons:

- New details are added to an already existing model that is known and verified. For each refinement step, there is not a combinatorial explosion of possibilities.
- Models are hierarchical and their components are self-contained. Adding a new detail does not affect currently existing parts of a model, but fills in the empty spaces.
- Modern parsing technology allows for modular and composable grammar specifications. Most of analyzed inputs are in a context-free form and new grammar rules do not affect currently existing ones.

Unfortunately, when we have tried to prove our ideas we face problems in an unexpected area; current parsing technology does not fulfill the flexibility and performance requirements of agile modeling. Let us first describe agile modeling in detail and focus on the parsing obstacles in the following section.

Agile Modeling in Detail

Agile modeling is a process of refinement. Each refinement step builds on the previous iteration. The initial iteration, *i.e.*, an initial build of a coarse model, builds on general knowledge about programming languages.

Let us inspect a single iteration of agile modeling. Consider a *depth inheritance analysis* (DIT) [SJC02]. It operates on classes and superclass relations — this is our *meta-model*. The concrete results are obtained from the sources with concrete classes and superclasses — this is our *model*. Depending on a programming language, classes and superclass relations are somehow encoded in source code, *e.g.*, using a `class` keyword followed by an identifier and an `extends` keyword followed by another identifier.

In *"traditional"* modeling, as depicted in Figure 1.1, the relations between a meta-model and sources are provided by an expert. The expert is fully responsible for discovering these relations as well as for providing the parser that extracts the concrete model from sources.

If there is another analysis to perform, for example *weighted methods per class* (WMC) [SJC02], the meta-model has to be extended with methods and their relation to a class. The expert has to extend the existing parser to extract methods as well.

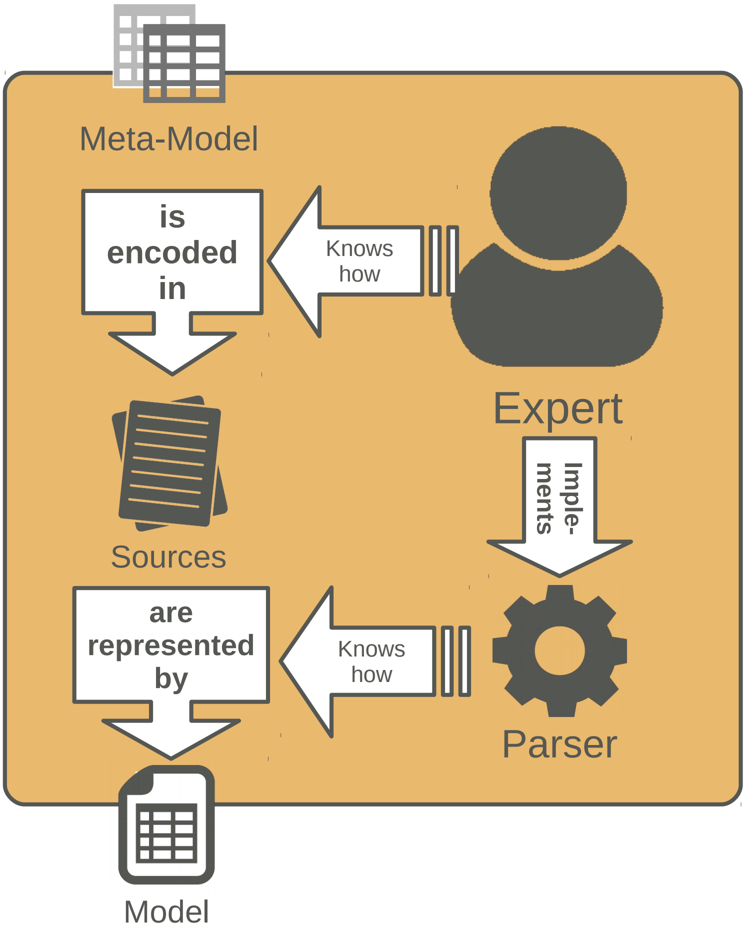


Figure 1.1: Traditional modeling schema, the burden is on expert who implements the parser based on her knowledge.

The traditional approach imposes a big burden on an expert. In *agile* modeling, as depicted in Figure 1.2, this burden is lowered by using heuristics, automation and general or previous knowledge. The heuristics try to guess the relations between a meta-model and sources and generate the parser. An expert verifies the results. If the results are incorrect, heuristics try another guess until the correct results are obtained. The whole process is semi-automated and is directed by an expert to reach the correct results faster.

For example, we experimented with automated extraction of language keywords based on their occurrences in source code. By using a completely language-agnostic analysis, Guggisberg found around fifty percent of Java, Shell and Haskell keywords [Gug15]. This information can be used by heuristics of agile modeling without questioning the expert, thus saving her time.

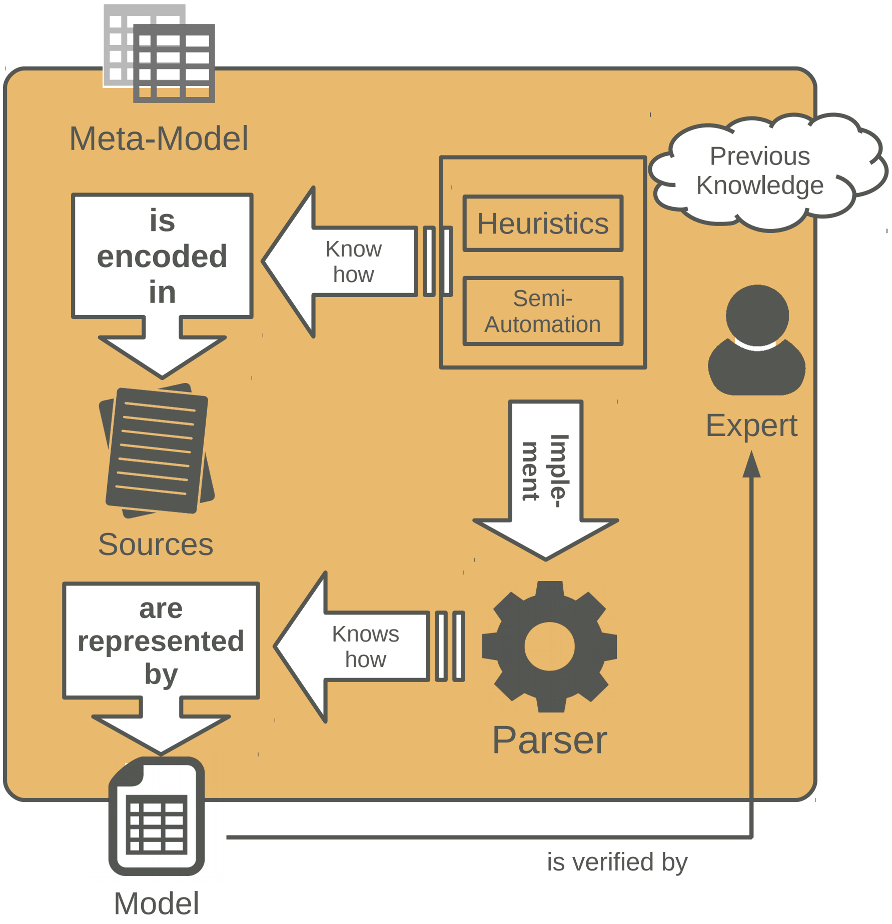


Figure 1.2: In agile modeling heuristics and automation are used to implement a parser. The burden on expert is lowered, because she only verifies results or guides decisions of heuristics.

If there is another analysis to be performed, the meta-model is extended with new entities required by the new analysis. The same tools are then used to obtain new entities from the sources building on top of the information harvested from the previous iteration.

An early example of agile modeling is a *parsing by example* approach utilized in CodeSnooper [Kob05]. CodeSnooper shows fragments of input to an expert who highlights program structures such as blocks or classes. A grammar is then inferred from these examples and a constructed parser attempts to recognize as much code as possible. If the recognition attempt fails, the CodeSnooper tool prompts an expert with further examples. Unfortunately, CodeSnooper utilizes an LALR parser generator, which has limited composability and quickly causes problems with grammar ambiguities.

1.2 Parsing Obstacles of Agile Modeling

Agile modeling has specific requirements on the parsing technology used [NK15]. Based on our own experience with agile modeling, we identify the following obstacles for a parser that can support agile modeling.

Expressiveness obstacles There are no limitations (within the domain of software systems) on syntax of the analyzed source. Any existing syntax has to be supported including uncommon features such as indentation sensitivity or various forms of context-sensitivity.

For example, where does the `doc` heredoc from Listing 1.1 end? If it is Ruby, it depends on the identifier after leading `'<<'`, which happens to be context-sensitive. Once an expert decides to interpret the text of heredoc as Markdown, is the `Position` class part of the list, or not? It depends on its indentation.

Tolerance obstacles Because initial phases of agile modeling produce a coarse grained model, it is very likely that fragments of sources will not be specified and therefore a parser has to tolerate unknown input.

What if an expert starts with the class hierarchy analysis of the code in Listing 1.1? Does she need to define grammar rules for instance variables, method definitions and method bodies?

Composability obstacles The process of refinement is an essential part of agile modeling. An inferred parser can be at any time extended with a new functionality that should not break the functionality already in place. A parser must be refinable without imposing any significant effort.

What if an expert decides to explore a method call graph for the code in Listing 1.1? Can she easily extend the existing parser with code that parses method calls? Can she embed a new parser to parse Markdown inside the heredoc strings?

Efficiency obstacles The agile modeling process is by its own nature based on a trial and error method and each trial can be verified after the parsing phase. There is a possibly large code base to be parsed and it has to be parsed in reasonable time. Though it is not the primary focus, a parser for agile modeling must provide good performance.

The first three obstacles relate to flexibility, while the last challenge relates to performance. Yet, performance and flexibility tend to contradict each other. Performance optimizations are usually based on specific expectations that are not always valid in agile modeling (*e.g.*, non-deterministic grammars, context-sensitive fragments, *etc.*). In addition, performance optimizations tend to tie code fragments together, which limits composability.

```
class Square < Shape
    @length = 1

    def draw(canvas)
        raise "Error" if canvas.nil?
        canvas.drawRectangle(
            Position(0,0),
            Position(length,length)
        )
    end

    @doc = <<heredoc
      Include the following:
      - @color instvar
      - inner class `Position`:
            class Position
              @x = 0
              @y = 0
            end
    heredoc
end
```

Listing 1.1: Example of a ruby code with embedded Markdown

1.3 Thesis

In this thesis we argue that the seemingly contradictory needs of a parser for agile modeling can be satisfied in a single parsing framework. We formally state our thesis as follows:

> *Flexible grammars with efficient parsers can be achieved with context-sensitivity and dynamic adaptation of parsing strategies.*

We show that context-sensitive grammars can provide the flexibility required from a parser that can support agile modeling and that performance comparable to non-flexible parsers can be achieved using adaptive parsing strategies in a single parser.

1.4 Our Contribution

We suggest two context-sensitive extensions of parsing expression grammars and a parser compiler. These context-sensitive extensions are parsing contexts [KLN14b] and bounded seas [KLN14a]. The context-sensitive extensions improve the flexibility of parsing expression grammars to overcome the first three obstacles of a parser for agile modeling. The parser compiler [KVG$^+$16] is an ahead-of-time source-to-source optimizer that addresses the last challenge of a parser for agile modeling.

As a proof of concept introduced in this thesis we extend PetitParser [RDGN10, KLR$^+$13] — a PEG-based parser combinator framework. In our work we show that with parsing contexts and bounded seas it is possible to quickly prototype a coarse-grained imprecise parser. We also show that it is possible to incrementally refine the parser by adding more features without any significant engineering overhead. Last but not least, we show that with dynamic parsing strategies we achieve significant performance speedup, which allows us to quickly extract data from analyzed sources. The resulting performance is comparable to a top-down hand-written and optimized parser.

2
Overview of Parsing Technologies

In this section we review available parsing technology from the perspective of challenges for agile modeling. We choose the most suitable ones and identify their limitations. Last but not least, we briefly introduce our solution.

2.1 Parsing in the Wild

In this section we review parsing formalisms and technologies related to the expressiveness, composability, tolerance to an unknown input and performance of a parser.

2.1.1 Expressive Power

There is a well-established hierarchy of grammars, the Chomsky hierarchy [Cho57], depicted in Figure 2.1. The least powerful languages in Chomsky hierarchy, regular languages, are equivalent to finite automata while the most powerful ones, recursively enumerable languages, are equivalent to the universal Turing machine. Let us inspect these parsing formalisms roughly from the least to the most powerful ones.

Regular Expressions A regular expression is an expression used to identify a subset of strings matching that expression. Regular expressions can describe infinite languages, but they are not suitable for describing structured languages. Common applications include data validation, data scraping (especially web scraping), data wrangling, simple parsing, the production of syntax highlighting systems *etc.* Regular expressions are often used in a traditional parsing to split input into tokens.

Deterministic Context-Free Grammars Deterministic context-free languages (DCFLs) such as LL(k) or LR(k) [AU72] are a popular class of languages because of their balance between expressiveness and parseability. Many programming languages are designed to fit into this class. The DCFLs can be recognized by a parser in a form of deterministic pushdown automata. Semantic and syntactic predicates [PQ94] and

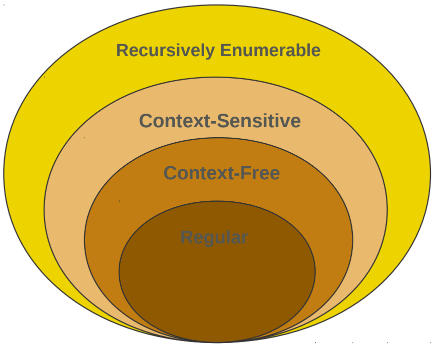


Figure 2.1: Chomsky Hierarchy

attributes [Knu68] or affixes [Kos91] extend the expressiveness of LL(k)- and LR(k)-based parsers beyond the context-free boundary.

Context-Free Grammars Any context-free languages [Cho57] are recognized by more powerful parsing algorithms such as Generalized LR (GLR) [Tom85], Generalized LL (GLL) [SJ10], CYK [You67], and Earley parsing [Ear70]. Parsing Expression Grammars (PEGs) [For04] provide an alternative formalism for describing syntax, and the class of parsing expression languages includes context-free and some context-sensitive languages. LL(*) grammars are an extension of LL(k) grammars with unlimited lookahead, reaching the expressiveness of PEGs, in some cases beyond GLR and PEGs [PF11].

Context-sensitive Grammars Definite Clause Grammars (DCGs) [PW86] offer a way of expressing grammars in a logic programming language such as Prolog. In pure Prolog normal DCGs can only express context-free grammars, however context-sensitive languages can be expressed in DCGs if extended with extra arguments.

The class of context-sensitive languages is equivalent to a linear bounded non-deterministic Turing machine [HU69]. The mildly context-sensitive formalisms [Jos85] such as tree adjoining grammars [JS97], linear context-free rewriting systems [VSWJ87], or multiple context-free grammars [SMFK91] generate a subset of context-sensitive languages. Adaptable grammars, either for CFGs [Chr09] or for PEGs [RVB$^+$12], can describe context-sensitive languages as well.

Parser Combinators [HM96, LM01] are used to build expressive parsing frameworks [Swi01]. Such frameworks can handle context-sensitive languages by a recursive descent of higher order functions [Bur75] with the full power of a functional language available to define new combinators for special applications, *e.g.*, layoutsensitivity [Lan66, AA14]

Layoutsensitivity, a special case of context-sensitivity, has been integrated into a popular languages such as Python, Haskell or F#. Recently, formal approaches have been suggested to integrate support for indentation into the existing parsing frameworks [ERKO12, Ada13]. These approaches do not offer the full expressiveness of a context-sensitive parser and focus only on the indentation aspect.

2.1.2 Composability

To achieve good composability, a parsing formalism has to describe nested structures and be closed under union. If a parsing formalism cannot describe nested structures, it cannot describe programming languages. If languages described by a parsing formalism are not closed under union, it is hard to add new sentences into the language and hence extend it with new constructs.

We will demonstrate the issues with composability on a Ruby example. Ruby contains instance methods (`defn`):

```
def getColor
    return color
end
```

There are also static methods (`defs`);

```
def self.RED
    return Color.new(255,0,0)
end
```

while methods can contain other methods:

```
def getColor
    def printColor
        puts color
    end
    printColor
    return color
end
```

Regular Expressions Regular languages are closed under union, concatenation and intersection. Let us suppose there is a rule to describe `defn` :

```
def [a-z]+.?end
```

and a rule to describe `defs` :

```
def self\.[a-z]+.?end
```

The rules can be composed to describe either `defn` or `defs` :

```
(def self\.[a-z]+.*?end)|(def [a-z]+.*?end)
```

On the other hand, method definitions based on regular expressions can never describe methods that contain other methods because regular expressions cannot refer to themselves.

Context-free Grammars Deterministic context-free languages [AU72] are not closed under union while context-free languages and parsing expression languages are both closed under union.

For example in case of CodeSnooper [Kob05], which utilized a deterministic parser, it is possible to describe methods in methods:

```
defn → 'def' id stmt+ 'end'
stmt → defn | defs | ...
```

On the other hand, the union of two deterministic languages `defn|defs` causes a conflict, because both rules start with `'def'`. A deterministic parser with the one-token lookahead cannot decide if it sees `defs` or `defn` and raises an error. Unless we use a non-deterministic parsing formalism (or increase the lookahead), refactoring of the grammar is needed to parse union of `defs` and `defn`.

Scannerless parsing Traditional scanner-parser pipelines [AU72] impose compositional challenges due to lexical ambiguities (*e.g.*, token conflicts of embedded languages). Scannerless grammars [SC89, Vis97b] and packrat parsing [For02b, For04] solve this problem by integrating lexical and context-free syntax into one grammar. Scannerless generalized LR, packrat-based LL grammars or PEGs support easy and seamless grammar composition.

To demonstrate issues with a scanner-parser pipeline, consider Ruby code parsed with the scanner-parser pipeline. In such a case, methods cannot be named `'def'` because of the conflict with the `'def'` keyword. Actually, even the highlighter, which is based on a lexical definition, of the following code snippet cannot distinguish between a keyword and a method name:

```
def def
    puts 'I would like to be a method named "def"'
end
```

This would not be an issue when using a scannerless parsing formalisms.

Left Recursion Top-down technologies such as LL and PEGs suffer from problems with left recursion, yet recently solutions have been proposed to tackle this problem [FHC07, WDM08, Tra10]. Parser combinator frameworks [HM96, LM01] offer great composability, but their top-down nature implies the same problems (and solutions) regarding left recursion.

For example, problems with left recursion prevent or complicate the following definitions of arithmetic expressions (*e.g.*, `'1+2-3'`):

```
expr → expr + expr |
       expr - expr |
       [0-9]+
```

The grammar, when parsed with a top-down parser, has to be either refactored or a mechanism that can avoid an infinite recursive descent has to be implemented.

2.1.3 Tolerant Grammars and Semi-Parsing

A *semi-parser*, the term coined by Dean *et al.* [DCMS03], is a parser that parses fragments of input while leaving others unparsed. Depending on a degree of tolerance, there are different approaches to semi-parsing. Zaytsev provides a comprehensive list of semi-parsing techniques [Zay14]. The list ranges from ad-hoc lexical analysis [KLV05] using `grep` or `sed` to practical and precise parsing [ALSU06] that skips only whitespace and comments. The two main application areas of semi-parsing are reverse engineering and error recovery. We call a formal definition of a semi-parser a *tolerant grammar.*

Reverse Engineering Techniques Some semi-parsing techniques are focused on reverse engineering, such as island grammars [Moo01], a method to deal with irregularities in the artifacts that are typical for reverse engineering. Island grammars combine the detailed specification possibilities of grammars (islands) with the liberal behavior of lexical approaches (water). A noise skipping parser GLR* [LT93] is a tolerant version of GLR that nondeterministically skips some word(s) in input and returns the parse with the fewest skipped words.

Island grammars are utilized in the agile parsing [DCMS03] paradigm using the non-greedy operators known from regular expressions[1] and syntactic predicates to extract embedded languages. Skeleton grammars [KL03] represent another semi-parsing approach to deal with different dialects of a programming language. In both cases the semi-grammar is inferred from a baseline grammar.

Error Recovery Techniques Other approaches are focused primarily on error recovery. The fuzzy parser used in Sniff [Bis92], a commercial C++ IDE, recovers on symbol declarations such as classes, functions and variables when dealing with incomplete C++ programs. Permissive grammars [KdJNNV09, JKSV12] are grammars derived from baseline grammars to deal with syntactical errors such as missing closing brackets *etc.* These techniques can also be complemented with bridge parsing [NNEH09], a lightweight recovery algorithm that extends an island grammar with the notion of bridges and reefs , which are used as synchronization tokens during error recovery. Reefs are obtained by layout-sensitive preprocessing of input and corresponding reefs are connected with bridges.

2.1.4 Performance

Naturally, the more expressive a parser is, the worse is its performance. Though the least powerful parsing formalisms such as DCFGs have reached their theoretical limits, there is a space for improvements for more powerful formalisms.

Regular Expressions Regular expressions are equivalent to finite state automata and can be implemented efficiently with linear asymptotical performance $\mathcal{O}(n)$ [Cox07].

Deterministic Context-free Grammars Parsing algorithms for deterministic context-free languages can offer linear asymptotic performance $\mathcal{O}(n)$ (where n is the input size) because they are equivalent to deterministic pushdown automata. There are table-driven parser generators for LL(k) or LR(k) grammars such as: SmaCC [BR], Happy [GM95] or Yacc [Joh75] and Lexx [LS75], or recursive ascent/descent parser generators such as Bison and Flex[Lev09], JavaCC,[2] ANTLR [PQ95], or ScalaBison [BS10].

Even in the class of deterministic parsers with linear asymptotic performance the constant overhead is significant and recursive descent/ascent approaches provide better performance than their table-driven variants [Pen86].

Context-free Grammars PEG parsers when combined with memoization [For02a] run in linear $\mathcal{O}(n)$ time as well. Memoization can be also utilized by other top-down approaches [FS96].

[1] http://www.webcitation.org/6k62faBSX

[2] http://www.webcitation.org/6k60x69Ga

It has been proven that the best performance for general context-free grammars can be $\mathcal{O}(n^{log7})$, *i.e.*, approximately $\mathcal{O}(n^{2.807})$ [Val75]. GLR [Tom85] is tuned for deterministic grammars where it performs in linear time $\mathcal{O}(n)$, or cubic $\mathcal{O}(n^3)$ in the worst case. The cubic $\mathcal{O}(n^3)$ asymptotic time holds for CYK [You67] and Earley [Ear70]. The GLL performance is also $\mathcal{O}(n^3)$ [AI15] in the worst case.

Context-sensitive Grammars When parsing beyond context-free grammars, the general PSPACE complexity [GJ79] led developers to search for alternatives. There exists a class of mildly context-sensitive grammars [Jos85], which has polynomial complexity $\mathcal{O}(n^k)$ (where n is an input size and k is some constant) by definition. This class includes tree adjoining grammars [JS97], linear context-free rewriting systems [VSWJ87], or multiple context-free grammars [SMFK91].

In the domain of programming languages, context-sensitive features (such as indentation) do not impose any asymptotic performance overhead when suitably implemented, *e.g.*, in Python [Pyt] a scanner contains stack of indentation levels to emit `indent` and `dedent` tokens. The grammar itself is then defined in a context-free form.

Parser Combinators The outstanding computation power and composability of parser combinators are obtained at the cost of performance. A parser combinator uses the full power of a Turing-equivalent formalism to recognize even simple languages that could be recognized by finite state machines or pushdown automata.

In Scala [Ode07], parser combinator libraries [MPO08] are optimized using macros [Bur13] and staging [RO10]. Both of the approaches lead to significant run-time speedup [BJ14, JCS$^+$14] as much of the overhead of parser combinators is removed thanks to pre-computations in a static phase. Other approaches battle performance problems using more efficient structures, macros *etc.* (see *Parboiled 2*,[3] *attoparsec*[4] or *FParsec*[5]).

Scannerless Grammars The expressive power of the scannerless parsing formalism comes at a price of lesser efficiency as well. Tokens are recognized using a more powerful yet more time and memory intensive parsing algorithm. The problem was addressed for scannerless GLR utilizing an adapted version of the Right-Nulled Generalized LR (RNGLR) parsing algorithm [EKV09].

2.1.5 Parsing Frameworks

Not all the technologies discussed in the previous section can be combined in a single parsing framework. To provide an overview how these techniques are used in real-world frameworks, we briefly describe existing frameworks and technologies implemented by these frameworks.

Yacc Family Yet Another Compiler Compiler (Yacc) [Joh75] is an LR parser generator used as the default generator on most Unix systems. Yacc is part of a standard scanner-parser pipeline, lexical analysis being performed by Lex [LS75]. The output of Yacc is a table-driven parser. There are many variants of Yacc, such as Bison [Lev09],

[3] http://www.webcitation.org/6k6195CiS
[4] http://www.webcitation.org/6k61DC2EA
[5] http://www.webcitation.org/6k61HcnHU

JavaCC,[6] SmaCC [BR], or Coco/R [Ter05]. The modern variants of Yacc utilize recursive descent/ascent rather than the original table-driven approach.

ANTLR ANother Tool for Language Recognition (ANTLR) [PQ95, Par07] is an expressive parser generator for processing structured text or binary files. ANTLR employs a standard scanner-parser pipeline. The generated parser has the top-down nature and utilizes a variant of LL recursive-descent parsing LL(*) [PF11], syntactic and semantic predicates [PQ94], and actions. ANTLR is used in many projects, *e.g.*, Groovy, Hibernate, Jython, and the Xtext [EV06] environment.[7]

TXL TXL [CHP88, Cor06] is a programming language specifically designed for manipulating and experimenting with programming language notations. The paradigm of TXL consists of taking a baseline grammar and specifying syntactic modifications to the grammar representing new language features or extensions.

TXL is based on a top-down parser resembling the generalized LL parser of ANTLR with backtracking similar to the Prolog's Definite Clause Grammars (DCG) [PW86]. TXL adopts technologies such as robust parsing [BG82], island grammars [Moo01], and agile parsing [DCMS03].

Syntax Definition Formalism The Syntax Definition Formalism (SDF) [HHKR89, Vis97a] is a formalism for definition of syntax which is an alternative to BNF. Contrary to BNF, SDF provides a wider scope, covers both lexical and abstract syntax and offers modular definitions. SDF is supported by Scannerless Generalized LR Parsing [Vis97b]. SDF utilizes indentation based divide-and-conquer approach to error recovery [dJNNKV09]. SDF is used in environments such as ASF+SDF [Kli93, BDH$^+$01], RascalIMPL [KvdSV09], Spoofax [KV10], or Stratego/XT [VeaB98, Vis02].

Happy Happy [GM95, Hap10] is a functional parser generator for Haskell similar to Yacc. It takes a BNF specification and produces a Haskell module containing a parser for the grammar.

Parsec Parsec [LM01] is a library for writing parsers in Haskell based on monadic higher-order parser combinators [HM96]. Contrary to a functional parser generator (*e.g.*, Happy) that offers a fixed set of combinators, Parsec allows for definitions of new combinators that fit the application domain. There exist various implementations for other languages such as Scala [MPO08], Erlang, Java, or OCaml.

Rats! Rats![8] is a parser generator supporting extensible syntax. Rats! utilizes a modular version of PEGs [For04] with semantic actions. Rats! is used in the *xtc* framework.[9]

PetitParser PetitParser [RDGN10, KLR$^+$13] is a PEG-based, parser combinator library for Smalltalk. PetitParser is used in the Moose analysis environment [ND04, Gî0].

[6] `http://www.webcitation.org/6k60x69Ga`

[7] A framework for development of programming and domain-specific languages.

[8] `http://www.webcitation.org/6k61Ps1QO`

[9] `http://www.webcitation.org/6k61WrLRC`

OMeta OMeta [WP07] is a specialized object-oriented programming language for pattern matching. OMeta is based on PEGs [For04], which have been extended to handle arbitrary kinds of data.

2.2 Existing Limitations

Given the state of the art, we see the following problems: (i) current parsing frameworks do not support parsing formalisms to express context-sensitive restrictions of programming languages; (ii) support for layoutsensitivity is not part of any of existing parsing frameworks (however research in this area is emerging); (iii) generalized parsing provides a set of results and requires an additional post-processing phase; (iv) without a baseline grammar island grammars do not provide the composability required by agile modeling; and (v) scannerless parsing and parser combinators impose noticeable overhead on parser performance.

As a suitable base framework for agile modeling, we decided to use PetitParser. It is based on the following technologies that, from the perspective of a parser for agile modeling, offer the following advantages: (i) parsing expression grammars have a good expressive power, are unambiguous and are closed under union; (ii) scannerless parsing provides a unified parsing formalism for grammar definitions and improves composability; (iii) packrat parsing ensures a linear asymptotic performance of parsing expression grammars; and (iv) parser combinators allow PEGs to be extended with specialized functionality.

On the other hand we have to deal with (i) missing support for context-sensitive definitions; (ii) limited composability of island grammars; and (iii) poor performance of scannerless parsing and parser combinators.

The issues cannot be avoided by choosing a different framework. None of the frameworks from subsection 2.1.5 supports context-sensitive grammar definitions. There are no high-performance and flexible alternatives to scannerless parsing or parser combinators. From the point of agile modeling, there is no better alternative to island grammars. A noise skipping parser is not deterministic and does not allow a grammar engineer to control the skipped input. Agile parsing and skeleton grammars require a baseline grammar, which is not available in the context of agile modeling. We discuss the issues of PetitParser from the perspective of agile modeling in the remainder of this section.

Context-Sensitivity in Programming Languages

Although many programming languages are designed with parsing in mind and they fit into the context-free class, not all of them are context-free. And other sources (*e.g.*, domain specific languages) interesting for agile modeling might not be context-free as well.

For example, Haskell, Python, or F# use indentation to specify block boundaries, an extra stack of opened elements is required to recognize an XML-like language without a schema, Ruby has no formal grammar specification whatsoever (only a c-like file[10]), similarly to Markdown, which is specified by a set of examples.[11]

Parsing expression grammars cannot express these languages on their own and existing *ad hoc* solutions do not fit the needs of agile modeling, which aims to remove the

[10] http://www.webcitation.org/6k62SaBg8

[11] http://www.webcitation.org/6k61x4RSO

implementation burden from an expert. Agile modeling requires formalized context-sensitive definitions that allow for an efficient parsing.

Island Grammars

An island grammar precisely defines only a subset of a language syntax (islands), while the rest of the syntax (water) is defined imprecisely. Usually water is defined as the negation of islands. Albeit simple, such a definition of water is naïve and impedes composition of islands. When developing an island grammar, sooner or later a language engineer has to create water tailored to each individual island.

Such an approach increases the burden on the language engineer. This again contradicts the goal of agile modeling. Agile modeling requires composable island grammars that do not require human intervention for different types of composition.

Performance Issues

The unified parsing algorithm of scannerless parsers and the universality and flexibility of parser combinators introduce a noticeable performance overhead. Speed-wise a parser based on these technologies cannot compete with a scanner-parser pipeline generated by well-performing parser generators or an optimized hand-written parser.

This is again against the goal of agile modeling, which aims to save the expert's time, not to waste it. Agile modeling requires well-performing and flexible parsers.

2.3 Our Solution

In this thesis, we address the issues with context-sensitivity, island grammars and performance problems. First we describe parsing expression grammars, parser combinators and PetitParser in chapter 3. Based on these technologies, we propose the following extensions:

Parsing Contexts extend parsing expression grammars with extra stacks to allow for context-sensitive definitions, *e.g.*, indentation (see chapter 4).

Bounded seas are a context-sensitive extension inspired by island grammars, which provides composable, robust, reusable and easy way to extract information from source code (see chapter 5).

Dynamic Parsing Strategies choose the minimal and most efficient parsing strategy required to parse a given grammar fragment. This greatly improves parser performance while imposing no limitations on the underlying grammar (see chapter 6).

In chapter 7 we present a case-study in which we prototype a Ruby parser for a class hierarchy analysis and we extend it for a call graph analysis. We implement the parser with context-sensitive rules in several iterations constantly improving precision and recall. We show that with our extensions a language engineer can extract useful data from early stages of parser development. The precision and recall improve as more grammar rules are specified. Thanks to dynamic parsing strategies, the performance of the parser is improved by factor of nine, which significantly shortens the feedback loop.

3
Parsing Expression Grammars and PetitParser

For reasons summarized in section 2.2 we decided to use PetitParser, which is based on parsing expression grammars, as a suitable framework for agile modeling. In this section we provide a brief summary of parsing expression grammars and PetitParser.

3.1 Parsing Expression Grammars

Parsing expression grammars (PEGs) developed by Ford [For04] are an alternative, recognition-based formal foundation for language syntax. PEGs are stylistically similar to CFGs with regular expression-like features, similarly to the Extended Backus-Naur Form (EBNF) notation [Wir77, ISO96]. The key difference is that in place of the unordered choice operator `|` used to indicate alternative expansions for a non-terminal in EBNF, PEGs use the prioritized choice operator `/`. This operator lists alternative patterns to be tested in order, unconditionally using the first successful match. The EBNF rules

```
A → a b | a
A → a   |  a b
```

are equivalent in CFGs, but the PEG rules

```
A ← a b / a
A ← a   /  a b
```

are different. The second alternative in the latter PEG rule will never succeed because the first choice is always taken if the input string to be recognized begins with `'a'`.

A PEG may be viewed as a formal description of a top-down parser. PEGs have more syntactic expressiveness than the LL(k) language class typically associated with top-down parsers, however, and can express all deterministic LR(k) languages and many others, including some context-sensitive languages. All PEGs can be parsed in linear time using a packrat parser [For02b].

Operator	**Description**
`''`	Literal string
`[]`	Character class
•	Any Character
`e?`	Optional
`e∗`	Zero or more
`e+`	One or more
`&e`	And-predicate
`!e`	Not-predicate
e_1 e_2	Sequence
e_1/e_2	Prioritized Choice

Table 3.1: PEG operators

The set of operators used to express PEG definitions is in Table 3.1. An example of a definition describing identifiers looks as in Listing 3.1.

```
id          ← idStart idCont* spacing
idStart     ← [a-zA-Z]
idCont      ← idStart / [0-9]
spacing     ← space*
space       ← ' ' / '\t' / '\n' / '\r\n' / '\r'
```

Listing 3.1: Example of a PEG definition

Thy syntax resembles the EBNF notation the differences being (i) already discussed prioritized choice `/`; (ii) syntactic predicates `!` and `&`; and (iii) non-terminal assignment `←` instead of `→`. The semantics is also similar, yet with important differences: (i) PEGs use the PEG formalism (instead of regular expressions) to describe the lexical syntax; (ii) choice is prioritized (ordered); and (iii) repetitions are greedy. The unified syntax frees lexical elements from the restrictions of regular languages (*e.g.*, a language engineer can express Pascal-like nested comments as tokens).[1] Moreover prioritized choices, greedy repetitions and syntactic predicates allow one to express disambiguation meta-rules on the grammar definition level (*e.g.*, syntax of lambda abstractions, `let` expressions and conditionals in Haskell [Has] that are ambiguous in CFGs and have to be disambiguated by the longest-match meta-rule).

3.1.1 PEG Analysis

Parser implementors often would like to analyze the behavior of a particular grammar over arbitrary input strings. While many interesting properties of PEGs are undecidable in general, conservative analysis proves useful and adequate for many practical

[1] *e.g.*, `(* this is (* a nested comment *) which continues here *)`

purposes, especially for grammar optimizations. We briefly discuss possible optimizations in this section and we describe them in detail later in chapter 6.

The first set from traditional parsing theory [GJ08a, pp. 235-361] can be computed even for PEs [Red09]. For example, any character in the `[a − zA − Z]` character class is in the first set of `id` (see Listing 3.1). We provide the formal definition of the first set for PEGs used in this work in Definition A.5. The first set can be used to optimize superfluous invocations, for example, to fail `id` directly if the peek character of the input is not a letter avoiding a superfluous invocation of the underlying sequence of `idStart` , `idCont∗` and `spacing` .

Furthermore, PEs can be interpreted abstractly to decide if an expression can accept an empty string, can succeed on some string or can fail on some string. For example, `idCont` can succeed on some input, *e.g.*, `'a'` . It can also fail on some input, *e.g.*, `'*'` and it can never accept an empty string ϵ . On the other hand, `idCont∗` can succeed on some input, *e.g.*, `'a'` and cannot fail on any input because it can accept ϵ . Because an abstract simulation does not depend on the input string, and there is a finite number of expressions in a grammar, we can compute an abstract simulation over any grammar [For04]. We provide the formal definition of an abstract simulation in Definition A.6. The abstract simulation can be used to optimize superfluous memoization, for example, to omit memoization before parsing `idCont∗` because, based on the abstract simulation, `idCont∗` cannot fail and the created memento is therefore never used.

3.1.2 Parser Combinators

Primarily, combinators in PetitParser are used to implement the PEG operators. The advantage of parsing expressions being implemented as parser combinators is in the composability of such a solution. All the parsing expressions or possible extensions form a composite pattern [Gam97]. Each expression can be treated uniformly and can accept an arbitrary sub-expression.

Therefore, combinators are used by clients to implement custom extensions of PetitParser. For example there exists a custom longest-match choice combinator, which, contrary to the ordered choice, unconditionally evaluates both alternatives and returns the result of the alternative that consumes most characters from the input. Last but not least, combinators are useful to define special expressions such as a start of a line or an end of a word, which are not part of a standard PEG formalism.

Parser combinators in general can have arbitrary behavior, however, throughout this thesis, we slightly restrict the behavior of parser combinators. For example, parser combinators used in this work delegate only to a single parsing expression and therefore they are unable to implement parsing expressions such as sequences or choices. We impose restrictions on parser combinators to simplify definitions and formalism. In practice, the restrictions can be avoided (as in PetitParser, which we discuss in the following section 3.2).

Because of naming conflicts we use k for parser combinator functions. We formalize parser combinators as follows:

Definition 3.1 (Parser Combinator)**.** A parser combinator k is a function that accepts a pair (e, x) as input and returns an output pair (o, y), where e is a parsing expression e, x is input $x \in \Sigma^*$, o indicates the result of the combinator, and $y \in \Sigma^*$ is the remainder of input. The distinguished symbol f indicates failure. □

For example, we define the negation operator of PEGs as a parser combinator as in Algorithm 3.1. When used with an expression e, the optional combinator evaluates e. Based on the result o it returns o or, if o is failure, ϵ . .

```
function NEGATION($e, x$)
    $(o, y) = parseOn(e, x)$                    ▷ evaluate $e$ for input $x$
    if $o = f$ then
        $\uparrow (\epsilon, x)$
    else
        $\uparrow (f, x)$
```

Algorithm 3.1: Negation parser combinator. If o is failure it returns ϵ, it returns f otherwise.

Let us extend the standard definition of PEGs (see Definition A.1) with parser combinators and we formalize their semantics (see Definition A.2).

Definition 3.2. (Parsing Expression Grammar with Parser Combinators) A parsing expression grammar with parser combinators is a 5-tuple $G = (N, \Sigma, R, e_s, K)$ where N is a finite set of nonterminals, Σ is a finite set of terminal symbols, R is a finite set of rules, K is a finite set of combinators, e_s is a starting expression.

Each rule $r \in R$ is a pair (A, e) which we write $A \leftarrow e$, $A \in N$ and e is a parsing expression. Parsing expressions (PEs) are defined inductively, if e_1 and e_2 are parsing expressions, then so are the following:

- ϵ , an empty string
- $'\mathtt{t}^{+}'$, any literal, $t \in \Sigma$
- $[\mathtt{t}^{+}]$, any character class, $t \in \Sigma$
- `A` , any nonterminal, $A \in N$
- `e?` , an optional expression
- $\mathtt{e}_1\mathtt{e}_2$, a sequence
- $\mathtt{e}_1/\mathtt{e}_2$, a prioritized choice
- `e*` , a zero-or-more repetitions
- `!e` , a not-predicate
- `&e` , an and-predicate
- `k(e)` , a parser combinator, $k \in K$

□

Definition 3.3 (Parser Combinator Semantics)**.** Parser combinator expression `k(e)` is a partially evaluated combinator function k. We extend the semantics of PEGs as defined in Definition A.2 with the semantics of a parser combinator expression as follows:

***Parser Combinator** (success):*

$$\frac{k(e,x) = (o,y)}{(k(e),x) \Rightarrow (o,y)}$$

***Parser Combinator** (failure):*

$$\frac{k(e,x) = (f,x)}{(k(e),x) \Rightarrow (f,x)}$$

□

We already discussed that parser combinators used in this work wrap only a single expression. Another restriction imposed on parser combinators is that parser combinators have to behave as first set terminals. For this reason `k(e)` cannot implement a repetition operator, because there is `ϵ` in the first set of a repetition operator (see Definition A.5).

Definition 3.4 (First Set of Parser Combinators)**.** We extend the definition of first (see Definition A.5) with parser combinators. They behave as first set terminals:

Combinator

$$\frac{}{\mathit{FIRST}(k(e)) = k(e)}$$

□

For example, consider a PEG rule that requires an identifier on the beginning of a line: `^ id` (where `^` stands for start of a line). The first set of the rule is a set with the single element: start of a line `^`.

The definition of an abstract simulation is permissive and allows for abstract result of a parser combinator. The abstract simulation of parser combinators is defined as follows:

Definition 3.5 (Abstract Simulation of Parser Combinators)**.** We extend the standard definition of an abstract simulation (see Definition A.6) with parser combinators. The abstract result of a parser combinator can be anything:

8. (a) $k(e) \rightharpoonup 0$
   (b) $k(e) \rightharpoonup 1$
   (c) $k(e) \rightharpoonup f$

□

For example, the abstract result of an abstract simulation of the `^` rule is `{0 1 f}`, meaning that a start of a line combinator can succeed while consuming no input, can succeed while consuming some input and can fail. It is not true that a start of a line combinator can succeed while consuming some input; this is a drawback of such a definition and the price for the flexibility of combinators.

3.2 PetitParser

PetitParser [RDGN10, KLR$^+$13] is a parser combinator framework [HM96] that utilizes packrat parsing [For02b], scannerless parsing [Vis97b] and parsing expression grammars (PEGs) [For04]. PetitParser is implemented in Pharo,[2] Smalltalk/X,[3] Java[4] and Dart.[5]

Operator	Description
`''`	Literal string
`[]`	Character class
`[] negate`	Complement of a character class
`•`	Any Character
`#letter`	Characters [a-zA-Z]
`#digit`	Characters [0-9]
`#space`	Characters [\t\n␣]
`^`	Start of a line
`e?`	Optional
`e*`	Zero or more
`e+`	One or more
`&e`	And-predicate
`!e`	Not-predicate
e_1 e_2	Sequence
e_1/e_2	Prioritized Choice
`e trim`	Trim spacing
`e token`	Trim spacing and build a token
`e map : action`	Semantic Action
`e memoize`	Packrat parser

Table 3.2: PetitParser operators

PetitParser uses an internal DSL similar to a standard PEG syntax as briefly described in Table 3.2. As an example, consider a snippet

```
id          ← #letter (#letter / #digit)*
```

that creates a nonterminal `id`, which consists of a letter followed by an arbitrary number of letters and digits (similarly to the `id` defined in pure PEG syntax in Listing 3.1).

Trimming and Tokenization PetitParser is scannerless [Vis97b], but dedicated `TokenParser` and `TrimParser` are at hand to deal with spacing (see Listing 3.1). These parsers trim the whitespaces (or even comments if specified) from input before and after a parse attempt. `TokenParser` returns a `Token` instance holding the

[2] http://smalltalkhub.com/#!/~Moose/PetitParser

[3] http://www.webcitation.org/6k62sGRlg

[4] http://www.webcitation.org/6k62uAxHz

[5] http://www.webcitation.org/6k62vRsWA

parsed string in the `inputValue` instance variable and its start and end positions (see Listing C.8). A token can be created using the `token` keyword:

```
id          ← #letter (#letter / #digit)*
idToken     ← id token
```

Parser Invocation When a root parser is asked to perform a parse attempt on input by calling `parse: input`, three things happen: (i) a `context` object representing an input stream is created from `input`; (ii) `parseOn: context` is called on the root parser; and (iii) the result of this call is returned. During an invocation parser combinators delegate their work to the underlying combinators. As an example, consider the action on the `idToken`:

```
id          ← #letter (#letter / #digit)*
idToken     ← id token
lcIdToken   ← idToken map: [:t | t inputValue asLowercase ]
```

The `map:` keyword creates an `Action` parser, the implementation of which can be found in Listing C.1. The `Action` parser invokes the underlying `idToken` and converts the result of `inputValue` to lowercase by invoking the action block. In case `idToken` fails and returns an instance of `Failure`, `Action` returns this failure immediately without invoking the action block.

Backtracking and Memoization PetitParser utilizes backtracking. Thanks to its backtracking capabilities, a top-down combinator-based parser is not limited to LL(k) grammars [AU72] but instead it can handle unlimited lookahead.

In PetitParser, before every possible backtracking point, the current context is remembered in a `Memento` instance. In case a decision turns out to be a wrong one, the context is restored from the memento. The same memento is used when memoizing the result of a parse attempt (to allow for packrat parsing [For02b]). A dedicated `Memoizing` parser combinator creates a memento, performs the parse attempt and stores a *memento-result* pair into a buffer. Later, if the memoizing parser is invoked again and the *memento-result* pair is found in the buffer, the result is returned directly.

To see how PetitParser backtracks, consider the `Sequence` parser implementation in Listing C.6. A sequence must iterate over all its children, collect their results and return a collection of the results. In case of failure, the context is restored to the original state and an instance of `Failure` is returned. Note that a parser returning failure from `parseOn:` is responsible for restoring the context to its initial state, *i.e.*, as it was before the `parseOn:` invocation.

Parser Combinators The strength of PetitParser is its flexibility and extensibility. PetitParser uses PEGs with parser combinators as defined in Definition 3.2. Any object that understands `parseOn: context` and follows its contract can be used as a parser combinator. All the standard operators of PEGs are implemented as a subclass of an abstract class `Parser` that defines the combinator interface. Users can implement their own combinators, *e.g.*, a start of a line combinator, either by subclassing `Parser` or by using a dedicated `Wrapping` parser (see Listing C.9), which serves as an adapter [GHVJ93] from a block closure interface to a parser combinator interface.

Restrictions imposed on parser combinators in this work are overcome in the case of PetitParser. PetitParser allows a combinator to specify the output of a PEG analysis (see subsection 3.1.1). For example, a repetition combinator of PetitParser modifies the default behavior of a first set analysis and adds ϵ into the output of the analysis.

4
Context Sensitivity in Parsing Expression Grammars

The domain of context-free languages has been extensively explored and there exist numerous techniques for parsing (all or a subset of) context-free languages. These techniques have been implemented in numerous parser generator frameworks, are well-understood and widely used in language compilers and interpreters. Unfortunately, not all programming languages or other sources interesting for agile modeling are context-free.

For example, Haskell, Python, or F# use indentation to specify block boundaries, C and C++ differentiate between `typedef` and identifiers,[1] an extra stack of opened elements is required to recognize an XML-like language without a schema, *heredoc* string literals of Ruby are context-sensitive,[2] in Markdown the beginning of a line associates the remaining content with arbitrarily interleaved lists and quoted blocks,[3] HTTP headers contain `lenght` field that specifies the length of the request body,[4] YAML allows one to specify block indentation in a block header,[5] and even an `if-then-else` statement produces a LR shift-reduce conflict.[6]

Without a formalized and well-performing approach to handle context-sensitivity, context-sensitive languages are excluded from agile modeling. Furthermore, as we argue in the next chapter, it is the context-sensitivity that allows for composable and flexible semi-parsers. Last but not least, it is a context-sensitive feature — layout — that is a good proxy to a document structure [HGH08], which can be leveraged by a parser for agile modeling.

Because of poor understandability, difficult parseability and insufficient seman-

[1] http://www.webcitation.org/6h0WvYD2B

[2] http://ruby-doc.org/core-2.3.0/doc/syntax/literals_rdoc.html#label-Here+Documents

[3] http://www.webcitation.org/6k61x4RSO

[4] http://www.webcitation.org/6k62Jw6Qi

[5] http://www.webcitation.org/6h0Xnc1m9

[6] http://www.webcitation.org/6k62MaMAT

tic suitability of context-sensitive grammars, implementors of programming languages use standard context-free parsing techniques and adopt various ways to express context-sensitive features, including *ad hoc* approaches like hand-written parsers, preprocessing and specialized lexers, post-processing of ambiguous parser output, or more formal approaches like attributes [Knu68], affixes [Kos91], and grammar adaptations [Chr09, RVB$^+$12].

In our work we propose an extension of the PEG formalism to express context-sensitive languages — *parsing contexts* [KLN14b]. Parsing contexts are a straightforward extension of top-down parsers and do not require any pre- or post-processing of an input stream. Definition rules remain in a simple context-free form $A \rightarrow \alpha$. Parsing contexts use stacks of values to steer the parser decisions with context-sensitive restrictions and are designed to allow for an efficient implementation.

4.1 Motivating Example

To illustrate the problem with context-sensitivity, consider Ruby strings. Ruby provides a lot of different string literals some of which require context-sensitive parsing. An example is the *here document*,[7] as we briefly discussed in Listing 1.1. It allows a programmer to write multi-line blocks of text by delimiting them with an identifier of her choosing as seen in Listing 4.1.

```
string = <<foobar
  This is a string
foobar

anotherString = <<end
  This is also a string
end
```

Listing 4.1: Two Ruby strings delimited by an arbitrary identifier.

A PEG definition of `heredoc` that a beginner could attempt to write looks as in Listing 4.2. The `openHeredoc` rule reads the open sequence. The `content` rule reads any character (`•`) as long as there is `closeHeredoc`. The `closeHeredoc` rule closes the string by consuming the final identifier.

This obviously does not work because any word consisting of letters is an identifier, including `'This'` — the first word in the string. The `ID` rule in `closeHeredoc` is context-sensitive and depends on the result of `ID` from `openHeredoc`.

The problem of context-free grammars is that they are not capable of expressing an arbitrary number of long-range relations.[8] If heredoc of Ruby had a pre-defined set of openings and closings[9], *e.g.*, `longstring`, `langtextli` and `dlouhytextik`, the `heredoc` rule might look like this:

[7] `http://ruby-doc.org/core-2.3.0/doc/syntax/literals_rdoc.html#label-Here+Documents`

[8] For example the $a^n b^n a^n$, $n > 0$ language, as can be proven using the pumping lemma (see `http://www.webcitation.org/6k7i6ZAdX`).

[9] which would be equivalent to n limited by a constant in $a^n b^m a^n$

```
ID            ← #letter+

heredoc       ← openHeredoc
                 content
                closeHeredoc
openHeredoc   ← '<<'ID
content       ← (!closeHeredoc •)*
closeHeredoc  ← ID
```

Listing 4.2: Naive definition of `heredoc`. It does not work, because `heredoc` is context sensitive and this definition does not take this into account.

```
heredoc       ← '<<'longtext     content1 longtext     /
                '<<'langtextli   content2 langtextli   /
                '<<'dlouhytextik content3 dlouhytextik
```

Listing 4.3: Grammar for `heredoc` with finite number of openings and closings

Yet, Ruby's heredoc allows for an arbitrary identifier `ID` to begin and end the heredoc element. In order to express such a context-sensitive restriction, the result of `ID` in `openHeredoc` has to be stored and compared in `closeHeredoc`. We now present a mechanism to do so.

4.2 Parsing Contexts

Parsing contexts extend PEGs (see Definition 3.2) with the notion of stacks. Stacks can hold arbitrary values and are manipulated using dedicated push $\underline{\nabla}$ and pop $\underline{\triangle}$ operators. Stacks can keep track of an arbitrary number of nested context-sensitive restrictions (*e.g.*, indentation, *etc.*) Stacks can be accessed by parser combinators, but, for the purposes of this work, we restrict parser combinators from modifying these stacks.[10] Combinators that access stacks are used to express the context-sensitive restrictions and to steer parser decisions.

Parsing contexts are an extension of PEGs, they do not modify the standard semantics of PEGs and they use the same form of rules: $A \leftarrow \alpha$. This preserves the advantages of context-free definitions: understandability, parsability and semantic suitability. Parsing contexts extend the formalism of PEGs with two new operators: push $\underline{\nabla}$ and pop $\underline{\triangle}$. The standard algorithms for the first set (see Definition A.5) and the abstract simulation (see Definition A.6) can be extended to support for these. The fact that the parsing context is manipulated only via rather '*low-level*' push and pop operators opens a space for analyses that allow for context-sensitive performance optimizations (as we show in chapter 6).

[10] This restriction can be avoided in the case of PetitParser by using properties as discussed in section 3.2.

4.2.1 Context-Sensitive Extension

With parsing contexts, we have a toolbox to define context-sensitive grammars using the PEG formalism. Let us now revisit the problem with the Ruby's *here documents*.

When parsing the open rule, we store the parsed value using the push `∇` operation to a stack. When parsing `closeHeredoc`, we compare the parsed value with the value on the top of the stack. To perform the comparison, we specify a parser combinator `cmp` that compares the top of a given stack with the return value of a given expression and returns failure if they are not equal (see Algorithm 4.2 for precise definition). Last but not least we clear the top of the stack with pop `△` if comparison succeeds.

The context-sensitive PEG definition of Ruby's heredoc is in Listing 4.4. The `openID` rule opens heredoc and saves the result of `ID` (*e.g.*, `'foobar'`) into the stack identified by the `here` identifier.[11] The `closeID` rule compares the result of `ID` with the top of the `here` stack and if they match, the top is popped from the `here` stack, `closeId` fails otherwise.

```
ID           ← #letter+

heredoc      ← openHeredoc
                content
               closeHeredoc
openHeredoc  ← '<<'openID
content      ← (! closeHeredoc •)*
closeHeredoc ← closeID

openID       ← ∇_here(ID)
closeID      ← cmp_here(ID) △_here
```

Listing 4.4: Context-sensitive `heredoc` definition.

Note that the `cmp` combinator can be reused for other purposes such as enforcing the correct close tag in XML-like languages. On the other hand, `cmp` itself is not sufficient to recognize Python-like `indent` and `dedent` tokens or other forms of indentation. For these other combinators are needed.

4.2.2 Indentation Stack

There are many variants of indentation (see Appendix D for more details). All of them are based on Landin's offside rule [Lan66], inspired by an offside in soccer. Once the offside line is set, the code (just like a player) is not allowed to cross the offside line.[12] As with soccer, the trick is to know where the line is and not get ahead of it.

Parsing contexts support indentation rules by offering an *indentation stack*. The indentation stack serves as a store for offside lines, which are naturally manipulated via push `∇` and pop `△` operators.

[11] Because we use a special stack (identified by `here`) for the context-sensitive restrictions of of the here document, another context-sensitive restriction can be expressed independently using a different stack.

[12] We assume that only spaces are used. If tabulators are used, they are replaced by a predefined number of spaces.

Offside Rule

The offside line is set in different cases, depending on a language. Once an offside line is set, code might appear in three positions: i. *aligned*; directly on the offside line if the column of code is the same as the column of the offside line; ii. *in onside*; to the right of the offside line if the column of code is strictly greater than the column of the offside line; and iii. *in offside*; to the left of the offside line if the column of code is strictly smaller than the column of the offside line.

In Python, for example, it is the first statement of a block that sets the offside line. The remaining statements of the block must be aligned, *i.e.*, must be directly on the offside line. Any other placement is invalid (see Listing 4.5).

```
while (count < 9):
   print 'The count is:', count   # sets the offside line
   count = count + 1              # aligned
     print 'loop ends'            # invalid alignment
```

Listing 4.5: Python block example.

Since blocks of code can be arbitrarily nested, offside lines can be nested as well. If an inner block sets the new offside line, the offside line of the outer block is remembered. If code appears in an offside position of the inner block, the offside line of the outer block is restored.

In Python, for example, blocks can be nested as depicted in Listing 4.6. If a statement appears in an offside position of an inner block, the block is terminated and the previous offside line is restored.

```
while (count < 9):
   print 'The count is:', count   # sets the offside line
   if (count % 2) == 0:
      print 'The count is event'  # new offside line is set
      count = count + 1
   else:                          # offside line restored
      print 'The count is odd'    # new offside line is set
      count = count + 1
```

Listing 4.6: Nested blocks in Python.

Offside Rule with Parsing Contexts

Givi [Giv13] implemented the indentation-sensitive rules by adding indentation-specific support to PetitParser. Building on this work, we embed the support into the more general concept of parsing contexts.

The offside rule operates with a concept of columns that is unknown to PEGs. Therefore, we provide a dedicated combinator `col` returning an integer representing the current column in the input string (the combinator is defined in Algorithm 4.1).

When provided with compare (`cmp` as in Algorithm 4.2), greater than (`gt` as in Algorithm 4.3) and smaller than (`st` as in Algorithm 4.4) combinators, a language engineer can set the offside line and verify the alignment as shown in Listing 4.7.

Note that rules in Listing 4.7 use a separate indentation stack IS and they can be used independently on other context-sensitive rules.

```
setOL      ← ∇_IS(col)      // set the offside line
removeOL   ← Δ_IS           // remove the offside line

aligns     ← cmp_IS(col)    // aligns to the offside line?
offside    ← st_IS(col)     // in offside?
onside     ← gt_IS(col)     // in onside?
```

Listing 4.7: PEG definitions of set, remove and align to offside line.

4.3 Parsing Contexts in Parsing Expression Grammars

In this section we provide a formal definition of Context-Sensitive PEGs (CS-PEGs) and specify parser combinators utilized to parse context-sensitive languages. Last but not least we provide an analysis to determine if an expression changes context or not.

Definition 4.1 (Context-Sensitive PEGs (CS-PEGs))**.** A context-sensitive parsing expression grammar is a 6-tuple $G = (N, \Sigma, R, e_s, K, C_s)$ where N is a finite set of nonterminals, Σ is a finite set of terminal symbols, R is a finite set of rules, e_s is a starting expression, K is a finite set of combinators, and C_s is an initial context — a finite set of an identifier to stack mappings. Each context entry $c \in C$ is a pair (id, S). id identifies a particular stack S, we write this as S_{id}.

Each rule $r \in R$ is a pair (A, e) which we write $A \leftarrow e$, $A \in N$ and e is a parsing expression. Parsing expressions (PEs) are defined inductively, if e_1 and e_2 are parsing expressions, then so is:

- ϵ , an empty string
- $'\mathtt{t}^{+}'$, any literal, $t \in \Sigma$
- $[\mathtt{t}^{+}]$, any character class, $t \in \Sigma$
- A , any nonterminal, $A \in N$
- e? , an optional expression
- $e_1 e_2$, a sequence
- e_1 / e_2 , a prioritized choice
- e$*$, a zero-or-more repetitions
- !e , a not-predicate
- &e , an and-predicate
- k , a parser combinator, $k \in K$

- $\underline{\triangledown}_{\texttt{id}}\texttt{e}$, a push to the stack S, $(id, S) \in C$
- $\underline{\triangle}_{\texttt{id}}$, a pop from the stack S, $(id, S) \in C$

□

Definition 4.2 (CS-PEG Semantics)**.** To formalize the semantics of a context-sensitive grammar $G = (N, \Sigma, R, e_s, K, C_s)$, we define a relation $\Rightarrow$ from triples of the form (e, x, C) to the output triples (o, y, C'), where e is a parsing expression, $x \in \Sigma^*$ is an input string to be recognized, o indicates the result of a recognition attempt, $y \in \Sigma^*$ is a remainder of input, and C, C' are context mappings.

The standard PEG operators as defined in Definition A.2 extend straightforwardly:

Empty:

$$\frac{x \in T^*}{(\epsilon, x, C) \Rightarrow (\epsilon, x, C)}$$

Terminal *(success):*

$$\frac{a \in T, x \in T^*}{(a, ax, C) \Rightarrow (a, x, C)}$$

Terminal *(failure):*

$$\frac{a \neq b, \quad (a, b, C) \Rightarrow (f, a, C)}{(a, bx, C) \Rightarrow (f, bx, C)}$$

Nonterminal:

$$\frac{A \leftarrow e \in R \quad (e, x, C) \Rightarrow (o, y, C')}{(A, x, C) \Rightarrow (o, y, C')}$$

Sequence *(success case):*

$$\frac{(e_1, x, C) \Rightarrow (o_1, y_1, C_1) \quad (e_2, y_1, C_1) \Rightarrow (o_2, y_2, C_2)}{(e_1 e_2, x, C) \Rightarrow (o_1 o_2, y_2, C_2)}$$

Sequence *(failure 1):*

$$\frac{(e_1, x, C) \Rightarrow (f, x, C)}{(e_1 e_2, x, C) \Rightarrow (f, x, C)}$$

Sequence *(failure 2):*

$$\frac{(e_1, x, C) \Rightarrow (o, y, C_1) \quad (e_2, y, C_1) \Rightarrow (f, y, C_1)}{(e_1 e_2, x, C) \Rightarrow (f, x, C)}$$

Choice *(option 1):*

$$\frac{(e_1, x, C) \Rightarrow (o, y, C')}{(e_1/e_2, x, C) \Rightarrow (o, y, C')}$$

Choice *(option 2):*

$$\frac{(e_1, x, C) \Rightarrow (f, x, C) \quad (e_2, x, C) \Rightarrow (o, y, C')}{(e_1/e_2, x, C) \Rightarrow (o, y, C')}$$

Repetitions *(repetition):*

$$\frac{(e, x, C) \Rightarrow (o_1, y_1, C_1) \quad (e*, y_1, C_1) \Rightarrow (o_2, y_2, C_2)}{(e*, x, C) \Rightarrow (o_1 o_2, y_2, C_2)}$$

Repetitions *(termination):*

$$\frac{(e, x, C) \Rightarrow (f, x, C)}{(e*, x, C) \Rightarrow (\epsilon, x, C)}$$

Not predicate *(success):*

$$\frac{(e, x, C) \Rightarrow (o, y, C')}{(!e, x, C) \Rightarrow (f, x, C)}$$

Not predicate *(failure):*

$$\frac{(e, x, C) \Rightarrow (f, x, C)}{(!e, x, C) \Rightarrow (\epsilon, x, C)}$$

□

Definition 4.3 (Context Manipulation Semantics)**.** Only the push and pop operations can modify a context: If S is a stack of elements $o_n : ... : o_2 : o_1 : []$, $[]$ denotes an empty stack, o_1 is the bottom element, o_n is the top element and $(o{:}S)$ denotes a stack S with o on top, $\{..{}_C, S_{id}\}$ denotes a parsing context $C = \{(i_1, s_1), (i_2, s_2), ..., (i_n, s_n), (id, S)\}$ where $..{}_C$ is a shorthand for $(i_1, s_1), (i_2, s_2), ..., (i_n, s_n)$, the semantics of stack manipulation parsing expressions is defined as follows:

Push $\underline{\triangledown}$ *(success):*

$$\frac{(e, x, C) \Rightarrow (o, y, \{..{}_{C'}, S_{id}\})}{(\underline{\triangledown}_{id} e, x, C)) \Rightarrow (o, y, \{..{}_{C'}, (o{:}S)_{id}\})}$$

Push $\underline{\triangledown}$ *(failure):*

$$\frac{(e, x, C) \Rightarrow (f, x, C)}{(\underline{\triangledown}_{id} e, x, C)) \Rightarrow (f, x, C)}$$

Pop $\underline{\vartriangle}$ *success:*

$$\frac{}{(\underline{\vartriangle}_{id}, x, \{..{}_C, (o{:}S)_{id}\})) \Rightarrow (o, x, \{..{}_C, S_{id}\})}$$

Pop $\underline{\vartriangle}$ *(empty case):*

$$\frac{}{(\underline{\vartriangle}_{id}, x, \{..{}_C, []_{id}\})) \Rightarrow (f, x, \{..{}_C, []_S\})}$$

□

Definition 4.4 (Context-Sensitive Parser Combinator)**.** A context-sensitive parser combinator is a function k that accepts a quadruple (id, e, x, C) where id is a stack identifier, e is a parsing expression, x is input $x \in \Sigma^*$, C is a context, and returns an output triple (o, y, C'), where o indicates a result of the combinator, $y \in \Sigma^*$ is remainder of input and C' is a new context. The distinguished symbol f indicates failure. □

Definition 4.5 (Context-Sensitive Parser Combinator Semantics)**.** A context-sensitive parser combinator expression $k_{id}(e)$ is a partially evaluated combinator function k where id refers to a stack identifier and e to an underlying parsing expression. The semantics is defined as follows:

Parser Combinator *(success):*

$$\frac{k(e, id, x, C) = (o, y, C')}{(k_{id}(e), x, C) \Rightarrow (o, y, C')}$$

Parser Combinator *(failure):*

$$\frac{k(e, id, x, C) = (f, x, C)}{(k_{id}(e), x, C) \Rightarrow (f, x, C)}$$

□

We impose one more restriction on parser combinators. Parser combinators in this work do not modify a parsing context, they only read it. Again, this restriction is not essential,[13] it simplifies the formalism used in this work.

4.3.1 Parser Combinators

To implement context sensitive features, we utilize several context-sensitive operations. First of all is a column `col` operator (see Algorithm 4.1), which is essential to bring the notion of columns into PEGs. Because the `col` parser combinator does not depend on any underlying expression e, it will be used with `e = ϵ`: `col ( ϵ )`, for simplicity we omit `ϵ` and write only `col`.

The function `getColumnOf(x)` in `col` returns the current column of input x. Its implementation is up to the implementor of an input stream. For example, an input stream can increment the column with each increment of a stream position and set the column to zero after each newline character.

```
function COL($e, id, x, C$)                          ▷ Column parser combinator
    $col = getColumnOf(x)$
    $\uparrow (col, x, C)$
```

Algorithm 4.1: Column parser combinator. Returns the column of x in input.

To compare column returned by the `col` combinator with the column of the offside line, three different combinators are needed: equals `=`, greater than `>`, and smaller than `<` (see Algorithm 4.2, Algorithm 4.3 and Algorithm 4.4). They all simply compare the result o with the top of a stack S and return either o or failure f.

```
function CMP($S, o, y$)                              ▷ Compare parser combinator
    $(o, y, \{..._{C'}, S_{id}\}) = parseOn(e, x, C)$    ▷ evaluate $e$ for input $x$ in context $C$
    if $(S_{id} = (p : S'_{id}))$ && $o = p$ then $\uparrow (o, y, C')$    ▷ Success, $S$ contains $o$
    else $\uparrow (f, x)C$                          ▷ Failure otherwise
```

Algorithm 4.2: Compare parser combinator. Checks if the top of a stack S_{id} in context contains the same value as the result of e.

4.3.2 CS-PEG analysis

In this section we extend the definitions of the first set (see Definition A.5) and the abstract simulation (see Definition A.6) to support CS-PEGs. Furthermore, we introduce a new context manipulation analysis, which can be used for parser optimizations as demonstrated later in chapter 6.

[13] In PetitParser combinators are expected to modify the context. The implementors of combinators can nevertheless override this default behavior.

```
function GT($e, id, x, C$)
    $(o, y, \{..{}_{C'}, S_{id}\}) = parseOn(e, x, C)$
    if $(S_{id} = (p : S'_{id}))$ && $o > p$ then $\uparrow (o, y, C')$
    else $\uparrow (f, x, C)$
```

Algorithm 4.3: Greater-than parser combinator. Checks if the result of e has a greater value than the top of a stack S_{id}.

```
function ST($e, id, x, C$)
    $(o, y, \{..{}_{C'}, S_{id}\}) = parseOn(e, x, C)$
    if $(S_{id} = (p : S'_{id}))$ && $o < p$ then $\uparrow (o, y, C')$
    else $\uparrow (f, x, C)$
```

Algorithm 4.4: Smaller-than parser combinator. Checks if the result of e has a smaller value than the top of a stack S_{id}.

First of push and pop operators

Because $\underline{\triangledown}$ pushes the result of an underlying combinator onto a stack, the first set of $\underline{\triangledown}$ is the first set of the underlying combinator. The $\underline{\vartriangle}$ simply pops from a stack and, from the point of the first set analysis, behaves as ϵ. For example, the first set of $\underline{\triangledown}$`(col)` is { `col` } and the first set of $\underline{\vartriangle}$ is { ϵ }. The formal definition is provided in Definition 4.6.

Definition 4.6 (Context-Sensitive First)**.** We extend the definition of first (see Definition A.5) with push $\underline{\triangledown}$ and pop $\underline{\vartriangle}$ operators.

Push

$$\frac{}{\mathit{FIRST}(\underline{\triangledown}(e)) = \mathit{FIRST}(e)}$$

Pop

$$\frac{}{\mathit{FIRST}(\underline{\vartriangle}) = \{\epsilon\}}$$

□

Context-Sensitive Abstract Simulation

The $\underline{\triangledown}$ operator does not modify the result of an underlying combinator. Therefore the abstract result of an abstract simulation of $\underline{\triangledown}$ is the abstract result of the underlying combinator. The $\underline{\vartriangle}$ operation can fail if a stack to be popped is empty, therefore there are two possible abstract results of $\underline{\vartriangle}$; either $\underline{\vartriangle}$ succeeds while consuming no input (*i.e.*, the abstract result is `0`) or it can fail (*i.e.*, the abstract result is `1`). The formal definition of the abstract simulation of $\underline{\triangledown}$ and $\underline{\vartriangle}$ is in Definition 4.7.

Definition 4.7 (Context-Sensitive Abstract Simulation)**.** To support the abstract simulation in CS-PEGs, we extend the relation $\rightharpoonup$ from Definition A.6 as follows:

9. (a) $\underline{\nabla}(e) \rightharpoonup 0$ if $e \rightharpoonup 0$.
   (b) $\underline{\nabla}(e) \rightharpoonup 1$ if $e \rightharpoonup 1$.
   (c) $\underline{\nabla}(e) \rightharpoonup f$ if $e \rightharpoonup f$.
10. (a) $\underline{\triangle} \rightharpoonup 0$.
    (b) $\underline{\triangle} \rightharpoonup f$.

□

Push-Pop Analysis

The *push–pop* analysis shows how a particular expression changes a stack in a context. Based on the *push–pop* analysis, there are four possible outputs:

1. If an expression e does not modify the stack, the result is `0`. This is the case of the standard parsing expressions or, for example, the `aligns` expression from Listing 4.7, which reads a stack, but does not modify it.
2. If an expression e pushes to a stack, the result is $\underline{\nabla}$. This happens, for example, in the case of `setOL` from Listing 4.7.
3. If an expression e pops from a stack, the result is $\underline{\triangle}$. This happens, for example, in the case of `removeOL` from Listing 4.7.
4. If an expression modifies a stack in some other way, *e.g.*, `removeOL*`, which pops all the elements from a stack and we don't know how many elements are popped, the result of the *push–pop* analysis is `1`.

The formal definition of the *push–pop* analysis is in Definition 4.8.

Definition 4.8 (*Push–pop* analysis)**.** We define a push–pop relation $\hookrightarrow$ consisting of triples (e, S, o), where e is an expression, S is a stack in a parsing context and $o \in \{0, \underline{\nabla}, \underline{\triangle}, 1\}$. $e \hookrightarrow_S 0$ means that e does not modify the stack S. $e \hookrightarrow_S \underline{\nabla}$ means that e pushes an element to the stack S. $e \hookrightarrow_S \underline{\triangle}$ means that e pops an element from the stack S. $e \hookrightarrow_S 1$ means that e modifies the stack S in some other way than push or pop. We define the push-pop relation $\hookrightarrow$ as follows:

1. $\epsilon \hookrightarrow_S 0$.
2. $t \hookrightarrow_S 0, t \in T$.
3. (a) $\underline{\nabla}(e) \hookrightarrow_S \underline{\nabla}$ if $e \hookrightarrow_S 0$
   (b) $\underline{\nabla}(e) \hookrightarrow_S 1$ otherwise
4. $\underline{\triangle} \hookrightarrow_S \underline{\triangle}$
5. $A \hookrightarrow_S o$ if $e \hookrightarrow_S o$ and $A \leftarrow e$ is a rule of the grammar G.
6. (a) $e_1 e_2 \hookrightarrow_S 0$ if $e_1 \hookrightarrow_S 0$ and $e_2 \hookrightarrow_S 0$.
   (b) $e_1 e_2 \hookrightarrow_S 0$ if $e_1 \hookrightarrow_S \underline{\nabla}$ and $e_2 \hookrightarrow_S \underline{\triangle}$.
   (c) $e_1 e_2 \hookrightarrow_S \underline{\nabla}$ if $e_1 \hookrightarrow_S \underline{\nabla}$ and $e_2 \hookrightarrow_S 0$.
   (d) $e_1 e_2 \hookrightarrow_S \underline{\nabla}$ if $e_1 \hookrightarrow_S 0$ and $e_2 \hookrightarrow_S \underline{\nabla}$.

   (e) $e_1 e_2 \hookrightarrow_S \underline{\triangle}$ if $e_1 \hookrightarrow_S \underline{\triangle}$ and $e_2 \hookrightarrow_S 0$.
   (f) $e_1 e_2 \hookrightarrow_S \underline{\triangle}$ if $e_1 \hookrightarrow_S 0$ and $e_2 \hookrightarrow_S \underline{\triangle}$.
   (g) $e_1 e_2 \hookrightarrow_S 1$ otherwise

7. (a) $e_1/e_2 \hookrightarrow_S 0$ if $e_1 \hookrightarrow_S 0$ and $e_2 \hookrightarrow_S 0$
   (b) $e_1/e_2 \hookrightarrow_S \underline{\triangledown}$ if $e_1 \hookrightarrow_S \underline{\triangledown}$ and $e_2 \hookrightarrow_S \underline{\triangledown}$
   (c) $e_1/e_2 \hookrightarrow_S \underline{\triangle}$ if $e_1 \hookrightarrow_S \underline{\triangle}$ and $e_2 \hookrightarrow_S \underline{\triangle}$
   (d) $e_1/e_2 \hookrightarrow_S 1$ otherwise

8. (a) $e* \hookrightarrow_S 0$ if $e \hookrightarrow_S 0$
   (b) $e* \hookrightarrow_S 1$ otherwise

9. $!e \hookrightarrow 0$

10. $k(e) \hookrightarrow o$ if $e \hookrightarrow_S o$

□

As a practical example, consider the `heredoc` rule from Listing 4.4:

```
heredoc     ←  openHeredoc
                  content
               closeHeredoc
```

The `openHeredoc` and `closeHeredoc` rules are push $\underline{\triangledown}$ and pop $\underline{\triangle}$ respectively. Based on the rule 8(a), the *push–pop* analysis outcome of `content` is 0, because the *push–pop* analysis outcome of `!closeHeredoc •` is 0 (based on the rule 9). The whole `heredoc` sequence does not change the context, because the initial push in the rule `openHeredoc` is reverted by the last pop in `closeHeredoc`. Accordingly, the result of abstract simulation is 0 because of the rule 6(b).

4.4 Implementation

The advantage of parsing contexts is their straightforward implementation. In PetitParser it suffices to modify `context` not to represent only an input stream, but an input stream and a set of mappings of identifiers to stacks. The `Context` class contains two instance variables, an input stream `stream` and a set of mappings `stacks`.

```
Object subclass: #Context
    instanceVariables: ’stacks stream’
```

For convenience, the new `Context` has the same interface as the old `Context` and the new one is interchangeable with the old one. Therefore the existing parsers of PetitParser do not need to be modified.

The memento of a parsing context is a position in the stream and a deep copy of all the stacks (see Listing 4.8). The copy is needed in `remember` as well as in `restore:` to preserve immutability of a memento. If a stack S in context is restored from a memento without the copy and S is modified, the memento is modified as well. If the memento is accessed in the future for another restore operation, we will need the unmodified version of S.

```
Context>>remember
    ↑ Memento new
        position: stream position;
        stacks: stacks deepCopy;
        yourself

Context>>restore: memento
   stream position: memento position.
   stacks ← memento stacks deepCopy.
```

Listing 4.8: Memoization of `Context`.

The push and pop operators are implemented as in Listing 4.9 and Listing 4.10. The `Push` parser contains a stack identifier `stackID` and a reference to a `delegate`, from which a value to be pushed on the stack is obtained. The `Pop` parser just pops from a stack identified by `stackID`.

```
Parser subclass: #Push
  instanceVariables: 'stackID delegate'

Push>>parseOn: context
  | result |
  result ← delegate parseOn: context.
  result ifSuccess: [
    context stackNamed: stackID push: result
  ].
  ↑ result
```

Listing 4.9: `Push` parser implementation.

```
Parser subclass: #Pop
  instanceVariables: 'stackID'

Pop>>parseOn: context
  | result |
  (context stackNamed: stackID) ifEmpty: [
    ↑ Failure new: 'Cannot pop an empty stack'
  ]
  ↑ (context stackNamed: stackID) pop.
```

Listing 4.10: `Pop` parser implementation.

4.4.1 Performance

In this section we briefly report on the performance of parsing contexts in PetitParser. Suppose s is the number of elements in all the context stacks, c is the number of possible stack states of stacks, n is the length of input, m is the number of keys into the memoization table of a packrat parser, and o is the number of parsing expressions in a grammar. A context-free parser has $s = 0$ because there are no context-sensitive restrictions. This means a memento is created in constant time. Furthermore $m = n$ because a key to the memoization table is a position in an input stream. Last but not least o is a constant because grammar definitions are finite. With a packrat parser, each expression is invoked at most once per memoization key. Therefore the complexity of a context-free parser is $\mathcal{O}(m * o)$ = $\mathcal{O}(n)$.

The time to remember and restore the context of a context-sensitive parser is s. In typical scenarios the number of elements in stacks is limited by a constant. The time complexity is also linear, because there is a constant number of possible stack states. Therefore there are $n * c$ possible keys to a memoization table (for each position in input at most c states of stacks). The time complexity of such a grammar is then $\mathcal{O}(s * n * c * o)$, which is $\mathcal{O}(n)$.

For example, in case of a layout-sensitive grammar the level of nesting and therefore the number of offside lines in stacks is limited by p, the maximum nesting in code. Therefore time to remember a stack is p, $s = p$ (supposing there is no other stack in a context). The stack can contain from zero to p offside lines and because an offside line is typically strictly further to the right than the previous one, there are p different states of an indentation stack. If there is no other stack in a parsing context, c equals p. Therefore there is a limited number of keys to a memoization table, for each position p different stacks, which is $m = n * p$ keys. At each position there is a limited amount of memoizations, in the worst case o, the number of parsing expressions in a grammar. The overall complexity is then $\mathcal{O}(n * p * p * o)$ where only n is not a constant, therefore the complexity is $\mathcal{O}(n)$.

In unusual scenarios s can be an arbitrary number. Furthermore the linear complexity of packrat parsing cannot be guaranteed. If a stack S contains at most s elements and each of these elements is, for example, an integer, there are very high $(int_{max})^s$ possible states of the stack S, where int_{max} is the maximum value of an integer. Moreover, in theory, s can be asymptotically larger than n. This means, if stacks can contain only integers, the complexity is $\mathcal{O}(n * (int_{max})^s * s * o)$, where only o is a constant.

Nevertheless, agile modeling does not focus on the worst-case scenario and we are not aware of any practical grammar with such bad time complexity. Based on our measurements all the practical grammars[14] presented in this work have linear asymptotic complexity.

As an example of an unusual grammar, consider a grammar that stores every character into the context and performs one backtrack per character. The parse time is $\mathcal{O}(n * (s + s))$, one remember (s) and one restore (s) per each character, *i.e.*, $\mathcal{O}(n^2)$, because $s = n$.

To measure overhead of mementos in PetitParser, we micro-benchmark two context-sensitive grammars and their context-free counterparts. The grammar definitions can be viewed and measurements can be reproduced from the Smalltalk image available online.[15]

Consume-All is a benchmark of a parser that consumes the whole input character by character. Before a character is consumed the `and` predicate is invoked: `(& • )`. The `and` predicate utilizes backtracking and a memento is created for

[14] Except the *consume-all* grammar from the following text, which we use only to demonstrate the overhead of memoization.

[15] `http://scg.unibe.ch/research/parsingForAgileModeling`

each character (see Listing C.2). If we don't use the `and` predicate, there is no memoization and we would not measure any memoization overhead. The input consists of random words. *Context-Free Consume-All* is the context-free version of Consume-All that simply consumes any character in input: $((\& \bullet)\ \bullet)*$. *Context-Sensitive Consume-All* is the context-sensitive version of Consume-All that stores every character into the parsing context: $((\& \bullet)\ \underline{\nabla}_{s} \bullet)*$.

Expressions is a benchmark of a parser that recognizes an arithmetic expressions with operators `+`, `-` and parentheses `(` `)`, for example `'2*(3+4)'`. The underlying grammar is in a deterministic form. Input consists of random arithmetic expressions. *Context-Free Expressions* is the context-free version of *Expressions* that simply recognizes arithmetic expressions. *Indentation-Sensitive Expressions* is a context-sensitive version of *Expressions* that utilizes indentation instead of opening and closing brackets `(` `)`. For example, `'2*(3+4)'` is represented as follows:

```
2*
  3+4
```

The Context-Sensitive Consume-All benchmark in Figure 4.1 shows that the overhead of mementos kicks-in almost immediately resulting in very high parse-time even for small inputs. The expressions benchmark in Figure 4.2 shows linear parse-time for both versions, though the indentation sensitive version has approximately five times worse performance.

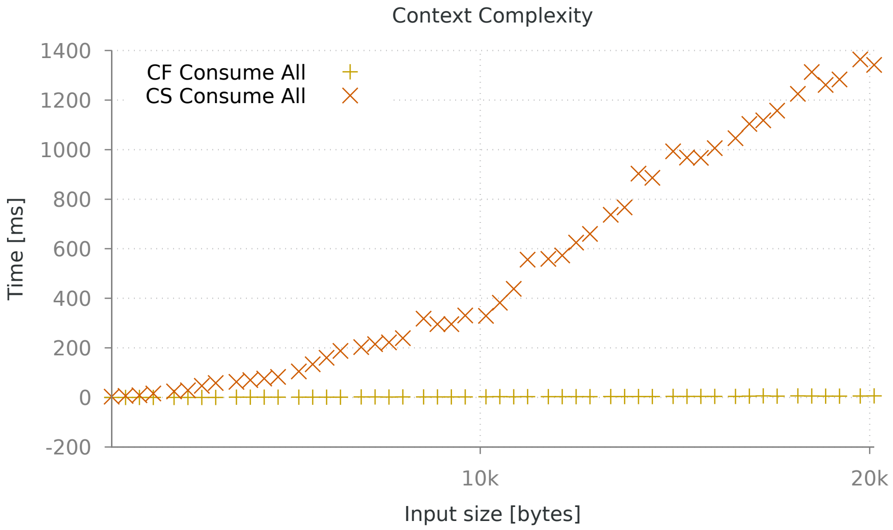


Figure 4.1: Dependency of parse time on the input size for context-free and context-sensitive variants of *Consume-all*.

Unfortunately, overhead is higher the more complicated the underlying grammar is (as can be seen in the Python Case Study in subsection 4.5.1 where the indentation overhead increases parse time by a factor of ten!). This is undesirable behavior and the overhead of mementos must be reduced. How to do this, we show in chapter 6.

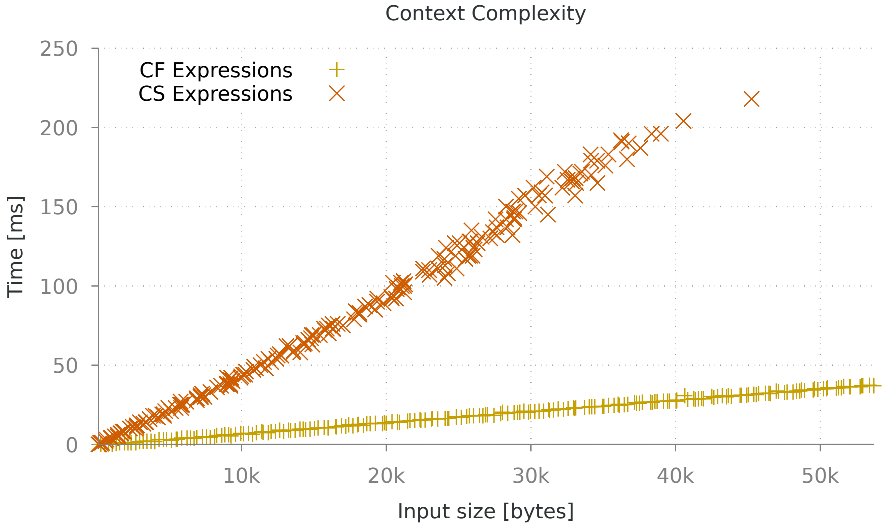


Figure 4.2: Dependency of parse time on the input size for for context-free and context-sensitive variants of *Expressions*.

The cause for the measured overhead is backtracking. Instead of an integer, which is sufficient to restore the state of a context-free parser, the deep copy of stacks is needed to restore the state of a context-sensitive parser. The more complex a grammar is, the more backtracking happens and more time is spent in memoizations. In chapter 6 we reduce the memoization overhead by smarter memoization: we avoid the full copy of stacks if we can guarantee that the stacks will not be modified and thus do not need to be restored.

4.5 Case Studies

To evaluate parsing contexts in real world scenarios, we present two case studies. The Python case study focuses on indentation. The Markdown case study challenges the flexibility of parsing contexts.

4.5.1 Python

The official Python grammar[16] is defined in a context-free form. It uses two indentation-sensitive tokens `indent` and `dedent` [17] produced by a specialized scanner that uses a stack of indentation levels to emit `indent` and `dedent` tokens.

In Python, the offside line is set by the first non-whitespace character on a line. The `indent` token is recognized when the first non-whitespace character of the current line is in onside. The `dedent` token is recognized when the first non-whitespace character of the current line is in offside. After `dedent` code must be aligned to the previous offside line. Neither `indent` nor `dedent` are recognized in case lines

[16] http://www.webcitation.org/6k633YacT

[17] http://www.webcitation.org/6k637RJ7V

are joined explicitly (using `\`) or implicitly (in parentheses `()`, square brackets `[]` and curly braces `{}`):

```
if 1900 < year < 2100 and 1 <= month <= 12 \  # join
   and 1 <= day <= 31 and 0 <= hour < 24 \     # join
   and 0 <= minute < 60 and 0 <= second < 60:
        return 1                               # indent

month_names = [
  'Januari', 'Februari', 'Maart',       # These are the
  'April',   'Mei',      'Juni',        # Dutch names
  'Juli',    'Augustus', 'September',   # for the months
  'Oktober', 'November', 'December']    # of the year
```

Indentation is also ignored in short- and long-strings:

```
'''
 I am a very long Python string
      spreading over multiple lines!
'''
```

The scannerless nature of PEGs prevents a language engineer from recognizing indents and dedents in a lexical phase and instead indents and dedents have to be recognized during parsing. With offside line operations as defined in Listing 4.7, `INDENT` and `DEDENT` can be defined as in Listing 4.11. The `align` rule consumes whitespace from the beginning of a line up-to the offside line. `INDENT` is recognized when there is an extra whitespace after the leading whitespace (consumed by `align`). `DEDENT` is recognized if after consuming whitespace and popping from the indentation stack, the current character is aligned (single dedent) or in offside (if there are two consecutive dedents, the first one is in offside and the second one is aligned).

```
// consume whitespace until aligned
align    ← ^(!aligns #space)*

INDENT  ← align #space+ setOL
DEDENT  ← #space* removeOL (offside / aligns)
```

Listing 4.11: Python's `indent` and `dedent` tokens.

The only rule of the Python grammar utilizing `indent` and `dedent` is `suite`:

```
suite → #newline INDENT stmt+ DEDENT
```

It is used in `if`, `for`, `try` and with statements and in class and function definitions.

Validation

To validate our approach, we implemented a semi-parser that extracts all the elements of Python that utilize indentation.[18] We extracted the structure of these programs as nested lists. For example, the following structure

[18] *i.e.*, `if_stmt`, `for_stmt`, `try_stmt`, `with_stmt`, `classdef` and `funcdef`

```
(<class>Shape(
    <def>draw (if())
    <class>Position ()
))
```

refers to a class `Shape`, with a method `draw` and an inner class `Position`, where `draw` contains an `if` statement. We used the Jython parser[19] to extract the reference structure. We compared the structure of our parser to the structure produced by the Jython parser.

We ran our validation on 917 files from three popular github projects: Django [20], Reddit [21] and Tornado [22]. We did not detect any differences between the two compared outputs.

Performance

Now we briefly report on performance of Python grammar implemented with parsing contexts, with focus on the overhead caused by indentation. We compare three parsers implemented by the author of this thesis: *Python Preprocessor*, *Context-Free Python* (CF Python) and *Python*. The Python preprocessor recognizes only `indent` and `dedent` in Python code skipping over the rest of input.[23] The Context-Free Python parser requires the Python preprocessor to insert `indent` and `dedent` into the file. The Python parser detects the `indent` and `dedent` tokens on its own utilizing parsing contexts and indentation stack.

We measure the time required to parse 229 randomly selected files, from three popular github projects: Django, Reddit and Tornado. The dependency between input size and parse time is shown in Figure 4.3. The Python parser is approximately ten times slower than its context-free counterpart, average time per character of Python parser being 15μs and average time per character of CF Python is 1.5μs. The performance of the Python preprocessor is comparable to the performance of the CF Python parser. The Python preprocessor and the CF Python parser in a pipeline are still five times faster than the Python parser. The extra time needed by the Python parser is spent in memoization of the indentation stack.

The measurements suggest linear asymptotic parse time in respect to the input size. However, the constant overhead is high. The optimization of the Python parser is discussed in chapter 6.

4.5.2 Markdown

The original syntax of Markdown by Gruber[24] is not formally specified, however there is an ongoing effort to formalize it, known as CommonMark.[25] In this section we refer to the CommonMark 0.19 specification.[26]

[19] http://www.webcitation.org/6k63EtVgf

[20] http://www.webcitation.org/6k63SAh1M

[21] http://www.webcitation.org/6k63U3rAH

[22] http://www.webcitation.org/6k63V40FG

[23] To validate the PythonPreprocessor we compared its output to that of Jython's tokenizer

[24] http://www.webcitation.org/6k61x4RSO

[25] http://www.webcitation.org/6k63cVKbG

[26] http://spec.commonmark.org/0.19/

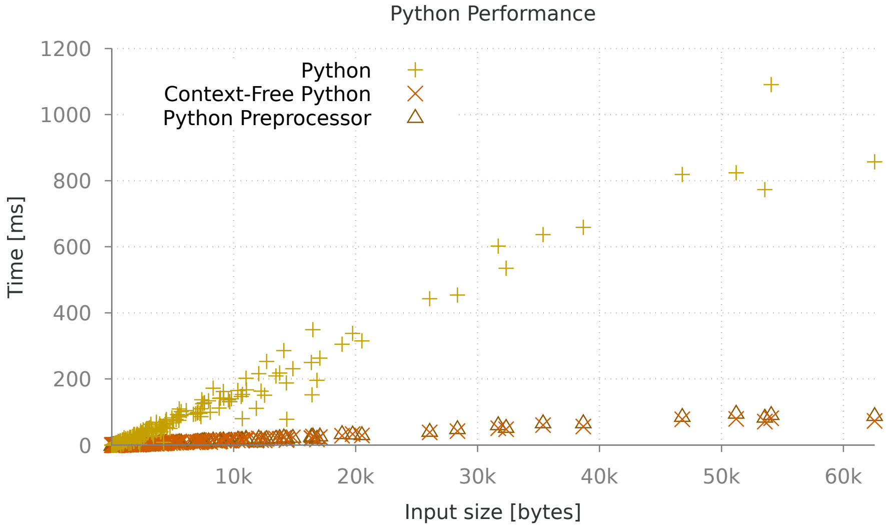


Figure 4.3: Dependency of parse time on the input size for Python, CF Python and Python Preprocessor.

CommonMark does not use a variation of Landin's offside rule, and contrary to many other languages that utilize indentation to delimit one kind of structure (*i.e.*, blocks), CommonMark uses a beginning of a line to delimit two kinds of structures: quoted blocks and lists. Furthermore, indentation of CommonMark does not operate with the concept of columns. It relies on the prefix of a content to assign the content to the corresponding structure element.

Quoted blocks and lists of CommonMark can be arbitrarily nested within each other. Any valid CommonMark can be a content of a list item or a quoted block. For this reason CommonMark uses not only whitespace `␣`, but also `>` to denote structure delimiters (see Listing 4.12).

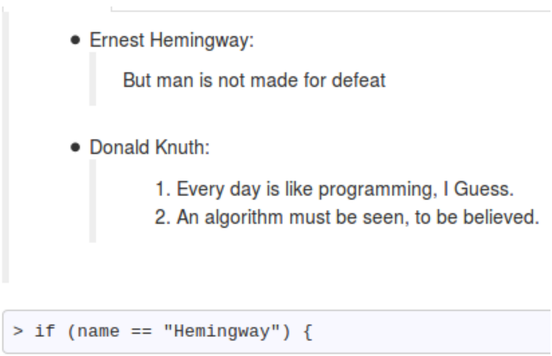
Each structure has its own prefix and the structure remains valid as long as its prefix appears at the beginning of a line. A quoted block has to start with `>`. The first line of a list item has to start with the item delimiter (*e.g.*, `'—'`) and the following lines have to start with enough spaces `␣` to align with the first line. If quoted blocks and list items are nested, the prefix is concatenation of the nested elements.

An example is in Listing 4.12. Line 9 nicely demonstrates the prefix of a content. Even though `> if` is aligned to the innermost quotes, the outer quotes are missing. For this reason `> if (...)` is not part of the outermost quoted block and the outermost block is ended. Four spaces are considered to be a code block indicator and therefore the text is rendered as code (see Figure 4.4).

CommonMark is a great parsing challenge as it is not formally defined. CommonMark is specified in the form of rules, each of which provides an expected output for the given input. The CommonMark parsing strategy is separated into two phases: (i) a block structure pass that recognizes block structures such as quoted blocks, lists, nested lists code blocks, *etc.* utilizing the context-sensitive rules; and (ii) an inline pass that recognizes text decorations such as emphasis, bold, underline *etc.*

```
> - Ernest Hemingway:
>   > But man is not made
>   > for defeat
> - Donald Knuth:
>   >  1. Every day is like programming,
>   >     I Guess.
>   >  1. An algorithm must be seen,
>   >     to be believed.
␣␣␣␣> if (name == "Hemingway") //line prefix differs
```

Listing 4.12: Indentation rules of CommonMark.



- Ernest Hemingway:

  > But man is not made for defeat

- Donald Knuth:

  > 1. Every day is like programming, I Guess.
  > 2. An algorithm must be seen, to be believed.

```
> if (name == "Hemingway") {
```

Figure 4.4: Rendering of the quoted list from Listing 4.12.

Implementation We focus on the indentation-sensitive part of CommonMark. In our parsing approach, a content is consumed line by line, the initial prefix being an empty string `$\epsilon$`. Whenever a quoted block or a list item starts, its prefix is pushed onto the stack S. The content after a line prefix belongs to the the block or list item as long as the prefix corresponds to the prefix on the stack S. The prefix is checked by matching all the elements on the stack starting from the beginning of a line. For this we implemented a dedicated `prefix` combinator (see Algorithm 4.5).

```
function PREFIX($e, id, y_1y_2...y_ny', C$)
    $list$ = stackAsList($S$)
    for $\forall x_i \in list$ do
        if $y_i \neq x_i$ then $\uparrow (f, y_1y_2...y_ny', C)$
    $\uparrow (y_1y_2...y_n, y', C')$
```

Algorithm 4.5: Prefix parser combinator. Compares the current input with the elements on the stack. Succeeds if input starts with all of the elements, fails otherwise.

With the prefix combinator, a simplified grammar of CommonMark looks as in Listing 4.13. The start rule is `content`. The `content` rule is a sequence of lines with the given prefix. If a new structure starts (*i.e.*, `quoteBlock` or `listItem`), the prefix of the structure (*i.e.*, `>` or `␣`) is pushed to S. Such an element is valid as long as there is a content with the new prefix. The `listItem` starts with `'- '`, yet spaces `'  '` are pushed to the stack S so the remaining lines are aligned with the first line of the item.

```
content     ← (prefix contentLine)*
contentLine ← quoteBlock /
              list /
              paragraph /
              ...

quoteBlock  ← ∇S('> ') contentLine   // a quote block starts
                 content
              △S                     // a quote block ends
list        ← listItem+
                                     // a list item starts
listItem    ← '- ' ∇S(ε map: ['  ']) contentLine
                 content
              △S                     // a list item ends
```

Listing 4.13: Simplified version of CommonMark that can parse arbitrarily nested lists and quoted blocks.

Results We implement a CS-PEG grammar for the context-sensitive phase of CommonMark. Its slightly simplified definition with descriptions can be found in Appendix G and the full grammar can be found in the prepared image available online.[27]

We validated our implementation on the complete set of 226 examples[28] of the context-sensitive phase with all the cases passing.

4.6 Related Work

In this section we briefly report on other technologies for context-sensitive parsing:

Context-Sensitive Grammars

There are multiple ways to express context-sensitive constraints in grammars. One of them is to use context-sensitive rules, *i.e.*, $\alpha A \beta \rightarrow \alpha \gamma \beta$ form, where $A \in N, \alpha, \beta \in (N \cup \Sigma)^*$ and $\gamma \in (N \cup \Sigma)^+$. Such rules, even though equivalent to the linear bounded Turing machines, are not really intuitive, impose serious challenges on a parsing technology (complex and with exponential times in general) and do not support semantic actions easily.

Consider the language $a^n b^n c^n$ whose grammar is in Listing 4.14. If a grammar engineer tries to extend it to $a^n b^n c^n d^n$ she finds out that it is not a straightforward task. Furthermore, if she tries to construct an abstract syntax tree out of this grammar, she will have a hard time to figure out where to attach semantic actions to build the tree.

[27] http://scg.unibe.ch/research/parsingForAgileModeling

[28] http://spec.commonmark.org/0.19/

```
S       → aSBc | abc
cB      → Bc
bB      → bb
```

Listing 4.14: Context-Sensitive BNF Grammar for a language $a^n b^n c^n$.

Attribute and Affix Grammars

Attribute [Knu68, Knu90] and affix [Kos91] grammars extend context-free grammars with inherited and synthesized attributes and evaluation rules (in the case of attribute grammars) or predicates (in the case of affix grammars). Attributes and evaluation or predicate rules might be used to verify context-sensitive constraints, for example by counting the number of `'a'`, `'b'` and `'c'` characters and checking for their equality as demonstrated in Listing 4.15.

```
S       → A(n) B(n) C(n)   { check(A.n == B.n),
                             check(A.n == C.n) }
A1      → a                { A1.n = 1 }
          | aA2            { A1.n = A2.n+1 }
B1    → b                  { B1.n = 1 }
          | bB2            { B1.n = B2.n+1 }
C1    → c                  { C1.n = 1 }
          | cC2            { C1.n = C2.n+1 }
```

Listing 4.15: Attribute grammar for $a^n b^n c^n$.

Parsing contexts are close to attribute grammars. The attributes can be perceived as values in a context and parser combinators accessing the context values as evaluation rules. Nevertheless, parsing contexts give more power to parser combinators than attribute grammars give to evaluation rules. Parser combinators can access input or even consume input. This opens a space for definition of indentation rules (accessing the current column in input) as we did in Python subsection 4.5.1 or consuming line prefixes (by consuming all input as defined in elements of prefix stack) as we did in Markdown subsection 4.5.2.

Two-Level Grammars We already mentioned that context-free grammars can express long-range relations but they are not capable of expressing an infinite number of them (see Listing 4.3). Unless we provide a grammar with an infinite number of rules. The idea of two-level grammars is to provide such an infinite grammar. The trick is to derive an infinite grammar from a finite definition.

An example of two-level grammars is VW grammars [Wij69], which were used in the definition of the programming language Algol 68. A VW grammar consists of a finite set of meta-rules, which are used to derive possibly infinitely many production rules and are implemented, for example, by the *yo-yo* parser.[29]

Tree-Adjoining Grammars (TAGs) Tree adjoining grammars [Jos85, JS97] fall into the class of mildly context-sensitive languages. They are related to context-

[29] http://www.webcitation.org/6k63lw3OE

free grammars, but the elementary unit of rewriting is a tree rather than a symbol. Whereas context-free grammars have rules for rewriting symbols, tree-adjoining grammars rewrite nodes of trees. TAGs are more powerful than CFGs but less powerful than CSGs. TAGs are typically applied in linguistics.

Boolean Grammars Boolean grammars [HS91, Okh04] obtain more than context-free power by extending context-free grammars with boolean operators *negation* and *intersection*. Heilbrunnen and Schmitz [HS91] provided an $\mathcal{O}(n^3)$ recognizer for Boolean grammars based on an adapted Earley parser. Okhotin [Okh05] shows how to use them to enforce context-sensitive restrictions in a simple C-like programming language, including checks for use of undefined identifiers, multiple definitions, and calling a function with a wrong number of parameters.

Adaptable Grammars Adaptable grammars [Chr09, RVB$^+$12] explicitly provide mechanisms within the parsing formalism to allow their own production rules to be manipulated. This can be used to express context-sensitive restriction, including indentation. The parser combinator nature of PetitParser allows for adaptability. However, the fact that a grammar can be adapted at any point complicates grammar analyses and consequent optimizations.

4.7 Conclusion

In this chapter we presented parsing contexts — a simple extension of PEGs that allows a language engineer to define context-sensitive restrictions such as indentation or context-sensitive behavior of MarkDown. Parsing contexts do not modify the semantics of existing PEGs and are easy to implement. Parsing contexts do not ensure linear asymptotic performance. However, according to our experience, parsing contexts perform in linear asymptotic time for languages typical in software engineering. Unfortunately, a parser utilizing parsing contexts suffers from high constant overhead.

The implementation, tests, validations, benchmarks and measurements this chapter can be re-run. The prepared image is available online.[30]

[30] `http://scg.unibe.ch/research/parsingForAgileModeling`

5 Semi-Parsing with Bounded Seas

Island grammars [Moo01] offer a way to parse input without complete knowledge of the target grammar. They are especially useful for extracting selected information from source files, reverse engineering and similar applications. The approach assumes that only a subset of the language syntax is known or of interest (the islands), while the rest of the syntax is undefined (the water). During parsing, any unrecognized input (water) is skipped until an island is found.

A common misconception is that water should consume everything until some island is detected. Rules for such water are easy to define, but they cause composability problems. Consider a parser where local variables are defined as islands within a method body. Now suppose a method declaring no local variables is followed by one that does. In this case the water might consume the end of the first method as well as the start of the second method until a variable declaration is found. The method variables from the second method will then be improperly assigned to the first one.

In practice, language engineers define many small islands to guide the parsing process. However it is difficult to define such islands in a robust way so that they function correctly in multiple contexts. As a consequence they are neither reusable nor composable.

To prevent our variable declaring island from skipping to another method, we have to make its water stop at most at the end of a method. In general, we have to analyze and update water of each particular island, depending on its context. Yet island-specific water is fragile, hard to define and it is not reusable. It is fragile, because it requires re-evaluation by a language engineer after any change in a grammar. It is hard to define, because it requires the engineer's time for detailed analysis of a grammar. It is not reusable, because island-specific water depends on rules following the island, thus it is tailored to the context in which the island is used — it is not general.

We propose a new technique for island parsing: *bounded seas* [KLN14a]. Bounded seas are composable, reusable, robust and easy to use. The key idea of bounded seas is that specialized water is defined for each particular island (depending on the context of the island) so that an island can be embedded into any rule. To achieve such composability, water is not allowed to consume any input that would be consumed by the following rule. To prevent fragility and to improve reusability, we compute rules for water automatically, without user interaction.

The chapter is organized as follows: Section 5.1 motivates this work by presenting the limitations of island grammars with an example. Section 5.2 presents our solution to overcome these limitations by introducing bounded seas. Section 5.3 introduces a sea operator for PEGs, which creates a bounded sea from an arbitrary PEG expression. Section 5.4 presents our implementation of bounded seas in PetitParser. Section 5.5 analyzes how well bounded seas perform compare to other island parsers. Section 5.6 surveys other semi-parsing techniques and highlights similarities and differences between them and bounded seas. Finally, section 5.7 concludes this chapter.

5.1 Motivating Example

Let us consider the source code in Listing 5.1 written in Ruby. We don't have a grammar specification for the code, because the parser was written using *ad hoc* techniques, and though we do have access to its implementation, we want to use a different programming language. Let us suppose that our task is to extract class and method names. Classes may be contained within other classes and we need to keep track of which class each method belongs to.

```
class Shape
    @position

    def getPosition
        return position
    end

    @uid = UIDGenerator.newUID
end
```

Listing 5.1: Source code of the `Shape` class in Ruby.

5.1.1 Why not use Regular Expressions?

To extract a flat list of method names, we could use regular expressions. We need, however, to keep track of the nesting of classes and methods within classes. Regular expressions are only capable of keeping track of finite state, so are formally too weak to analyze our input. To deal with nested structures, we need at least a context-free parser.

Modern implementations of regular expression frameworks can parse more than regular languages (*e.g.*, using recursive patterns as in Ruby,[1] Perl[2] or PCRE[3]). Such powerful frameworks can handle our rather simple task. However regular expressions are not meant to specify complex grammars.

[1] http://ruby-doc.org/core-2.0.0/Regexp.html

[2] http://www.webcitation.org/6k63y2Dqf

[3] http://www.webcitation.org/6k64HgWs5

5.1.2 A Naïve Island Grammar

To write a parser, we need a grammar. Because a grammar can easily consist of a hundred rules (*e.g.*, ≈ 80 for Python, ≈ 160 for Ruby, ≈ 180 for Java) and since we are only interested in specific parts of the grammar, we define an island grammar with fewer than ten rules as in Listing 5.2. We initially assume that each class body contains just one method. Since we are interested in extracting method names, we define the `method` rule as an island inside of the `methodInWater` rule which surrounds it with water. The `methodInWater` rule is defined imprecisely: water skips everything until the string `'method'` is found. We also define the `methodBody` rule, which skips everything until `'end'` is found.

```
start         ← class
class         ← 'class' id classBody 'end'
classBody     ← methodInWater

methodInWater ← (!'def' •)* method (!'end' •)*

method        ← 'def' id methodBody
methodBody    ← (!'end' •)* 'end'

id            ← #letter+
```

Listing 5.2: Our first island grammar.

Composability Problems.

The `methodInWater` rule in the grammar in Listing 5.2 uses a naïve definition of water. It will work as long as we do not complicate the grammar.

Suppose that in order to allow for multiple classes in a single file we modify the start allowing repetitions:

```
start ← class∗
```

Parsing the input in Listing 5.3 should fail because `Shape` does not contain a method. The result, however, is only one class — `Shape` (instead of `Shape` and `Circle`) — with one method `getDiameter`, which is wrong. We see that our water is too greedy here, trying to find `method` at any cost and ignoring `'end'` and the `Circle` definition.

Things do not get better when we allow multiple repetitions of `methodInWater` within `classBody` (`classBody ← methodInWater∗`). The parser stays confused, and the result stays incorrect. A language engineer has to use predicates to prevent the incorrect decisions of PEGs.

5.1.3 An Advanced Island Grammar

To make the `methodInWater` rule composable we must make it possible for it to be embedded into optional (`?`) or repetition (`+` , `∗`) rules. We consequently define

```
class Shape
    @uid = UIDGenerator.newUID;
end

class Circle
    @diameter

    def getDiameter
        return diameter
    end
end
```

Listing 5.3: Source code of `Shape` and `Circle` classes.

the grammar as in Listing 5.4. This new definition can properly parse multiple classes in a file with an arbitrary number of methods in a class. We achieve composability by forbidding water in `methodInWater` to go beyond the `'end'` keyword and by forbidding water to consume any method definition.

```
start          ← class*
class          ← 'class' id classBody 'end'
classBody      ← methodInWater+ / (!'end' •)*

methodInWater  ← (!'def' !'end' •)*
                   method
                 (!'def' !'end' •)*
method         ← 'def' id methodBody
methodBody     ← blockBody 'end'

blockInWater   ← (!'if' !'while' !'for' !... !'end' •)*
                   block
                 (!'if' !'while' !'for' !... !'end' •)*
block          ← 'if' blockBody 'end' /
                 'while' blockBody 'end' /
                 'for' blockBody 'end' /
                 ...
blockBody      ← blockInWater+ / (!'end' •)*

id             ← #letter+
```

Listing 5.4: Complete and final island grammar.

One can see that the syntactic predicates in `methodInWater` are more complicated. They are inferred from the rest of the grammar by analyzing which tokens can appear after the `method` island.

This might be particularly tiresome in the case of `blockInWater`. Once a language engineer realizes that almost any control flow structure in Ruby ends with `'end'`, she has to define many other block structures (see `block`) to properly pair `'end'` and update corresponding water rules (see `blockInWater`). Such `blockInWater` cannot be easily reused, *e.g.*, in classes with curly brackets `{` `}`, because the water predicates are missing the `!'{'` and `!'}'` rules.

Since everything is working as we expect, it is time to to allow for nested classes, *i.e.*, we extend the rule `classBody` to:

```
classBody ← (methodInWater / classInWater)*
```

We have to revise the predicates of `methodInWater` and add the `!'class'` predicate. We have to find the proper predicates for the `classInWater` rule as well. This is becoming tiresome even in our toy example.

Ease of Use, Robustness, and Reusability Problems.

The limitations of defining `methodInWater` and `classInWater` by hand illustrate the general problems of semi-parsing [DCMS03, Zay14] with island grammars:

1. Water rules are hard to define correctly because they require the entire grammar to be analysed.
2. The definition of water is fragile because predicates need to be re-evaluated after any change in a grammar.
3. Finally, the water rules are tailored just for a specific grammar and cannot be reused in another grammar with different rules.

5.2 Bounded Seas

We have shown that water must be tailored both to the island within the water and to the surroundings of the water (*e.g.*, `methodInWater` in Listing 5.4). In our work we define a *bounded sea* to be *an island surrounded by context-aware water*.

To automate the definition of bounded seas we introduce a new operator for building tolerant grammars: the *sea operator*. We use the notation `∼island∼` to create a sea from `island`, which can be a terminal or non-terminal. Instead of having to produce a complex definition of a sea, a language engineer can use the sea operator which will do the hard work. Listing 5.5 shows how the grammar of Listing 5.4 can be defined using the sea operator.

A rule defined with the sea operator (*e.g.*, `∼method∼`) maintains the composability property of the advanced grammar since by applying the sea operator we search for the island in a restricted scope. Moreover, such a rule is reusable, robust, and simple to define. Bounded seas are based on two ideas:

1. *Water never consumes any input from the right context of the bounded sea*, *i.e.*, any input that can appear after the bounded sea. This is very different from the water of “traditional” island grammars, where water might not consume a part of a valid input (cf. section 5.1.2). The water of bounded seas is unambiguous, thus improving composability.
2. *Everything is fully automated.* The sea is created using the sea operator, *i.e.*, `∼island∼`. Once a sea is placed in a grammar, the grammar is analyzed and appropriate water is created without user interaction. This way a sea can be

```
start        ← class*
class        ← 'class' id classBody 'end'
classBody    ← ~method~+ / ~ε~

method       ← 'def' id methodBody
methodBody   ← blockBody 'end'

block        ← 'if' blockBody 'end' /
               'while' blockBody 'end' /
               'for' blockBody 'end' /
                ...
blockBody    ← ~block~+ / ~ε~

id           ← #letter+
```

Listing 5.5: Island Grammar from Listing 5.4 rewritten with the sea operator.

placed in any grammar. In case the grammar is changed, the water is recomputed automatically. Automatic water computation eases grammar definition, and ensures robustness and reusability of rules.

Since bounded seas are designed to be integrated into a PetitParser, they have to handle the flexibility of parser combinators.

5.2.1 The Sea Boundary

Ideally water should never consume any input that can appear after a bounded sea, *i.e.*, it should never consume input from its right context. We will call the right context the *boundary* of a sea.

The right context of a sea consists of inputs accepted by parsing expressions that appear after the island of the sea. In the case of `A ← ~'a'~ (B/C)`, the right context of `~'a'~` is any input accepted either by `B` or by `C`.

Being aware of the boundary, a tolerant parser can search for methods in a class without the risk that other classes will interfere. Bounded seas would correctly parse the input in Listing 5.3 because water of a method sea would not be allowed to consume `'end'`, which is a boundary of `~method~`.

The island-specific water has to stop in two cases: first, when an island is reached; second, when a boundary is reached. If a boundary is reached before an island is found, the sea fails. The fact that a sea can fail implies that a sea can be embedded into optional or repetition expressions without ambiguous results. For example, we can define the superclass specification as an optional island:

```
~classDef~ ~superclassSpec~? classBody 'end'
```

If `superclassSpec` is not present for the particular class, it will simply fail upon reaching `classBody` instead of searching for `superclassSpec` further and further. The same holds for repetitions.

```
classBody ← ~method~*
```

This rule will consume methods only until it reaches `'end'` in the input string, since `'end'` is in the boundary of `~method~`. This means methods in another class cannot be inadvertently consumed.

Definition 5.1 (Bounded Sea)**.** We first define bounded seas generally, and subsequently provide a PEG-specific definition. A *bounded sea* consists of a sequence of three parsing phases:

1. **Before-Water:** Consume input until an island or the right context appears. Fail the whole sea if the right context appears. Continue if island is reached.
2. **Island:** Consume an island.
3. **After-Water:** Consume input until the right context is reached.

□

5.2.2 The Context Sensitivity of Bounded Seas

In order to preserve the unambiguity of water in bounded seas, they need to be context-sensitive. A bounded sea recognizes different substrings of input depending on what surrounds the sea. There are two cases where context-sensitivity emerges:

1. A bounded sea recognizes different input depending on the right context — what immediately follows the sea.
2. A bounded sea recognizes different input depending on the left context — what immediately precedes the sea.

Let us demonstrate the context sensitivity of bounded seas using the rules from Listing 5.6 and two inputs: `'..a..b..'` and `'..a..c..'`. On its own, `A` recognizes any input with `'a'` and `B` recognizes any input with `'b'` (see rows 1-4 in Table 5.1), because they are not bounded by anything.

```
A  ← ~'a'~
B  ← ~'b'~

R1 ← A          R2 ← B
R3 ← A 'b'      R4 ← A 'c'
R5 ← A  B
```

Listing 5.6: Rules for demonstrating context-sensitive behavior.

However, when the two islands are not alone, their boundary can differ, depending on the context. The right context of `A` is `'b'` in `R3`, and the right context of `A` is `'c'` in `R4`. Therefore `A` consumes different substrings of input depending whether it is called from `R3` or `R4` (see rows 5-8 in Table 5.1).

	Rule	Input	Result	
1	`R1 ← A`	`'..a..b..'`	`A` recognizes '..a..b..'	
2	`R1 ← A`	`'..a..c..'`	`A` recognizes '..a..c..'	
3	`R2 ← B`	`'..a..b..'`	`B` recognizes '..a..b..'	
4	`R2 ← B`	`'..a..c..'`	`B` fails	
5	`R3 ← A'b'`	`'..a..b..'`	`A` recognizes '..a..'	`'b'` recognizes 'b'
6	`R3 ← A'b'`	`'..a..c..'`	`A` recognizes '..a..b..'	`'b'` fails
7	`R4 ← A'c'`	`'..a..b..'`	`A` recognizes '..a..b..'	`'c'` fails
8	`R4 ← A'c'`	`'..a..c..'`	`A` recognizes '..a..'	`'c'` recognizes 'c'
9	`R5 ← AB`	`'..a..b..'`	`A` recognizes '..a..'	`B` recognizes 'b..'
10	`R5 ← AB`	`'..a..c..'`	`A` recognizes '..a..c..'	`B` fails

Table 5.1: The seas `A` and `B` recognize different inputs depending on the context.

Overlapping seas

A more complex case of context-sensitivity, which we call the *overlapping sea problem*, arises when one sea is immediately followed by another. Consider, for example, the rule `R5`, where the sea `A` has as its right context `B`, which is also a sea. Note that the before-water of `B` should consume anything up to its island `'b'` or its own right context, *including the island of its preceding sea* `A`. Now, the before-water of `A` should consume anything up to either its island `'a'` or its right context `B`. But the very search for the right context will now consume the island we are looking for, since before-water of `B` will consume `'a'`! We must therefore take special care to avoid a "shipwreck" in the case of overlapping seas by disabling the before-water of the second sea. Therefore `B` recognizes `'..a..b..'` when called from `R2` and `'b..'` when called from `R5` (see rows 3 and 9 in Table 5.1). For the detailed example of the `∼a∼∼b∼` sequence, see section B.3.

5.3 Bounded Seas in Parsing Expression Grammars

Starting from the context-sensitive definition of PEGs (see Definition 4.1), we now show how to add the sea operator to PEGs while avoiding the overlapping sea problem. To define the sea operator, we first need the following two abstractions:

The water operator *consumes uninteresting input.* Water (`≈`) is a new PEG prefix operator that takes as its argument an expression that specifies when the water ends. We discuss this in detail in subsection 5.3.1.

The *NEXT* function *approximates the boundary of a sea.* Intuitively, *NEXT(e)* returns the set of expressions[4] that can appear directly after a particular expression e. The details of the *NEXT* function are given in subsection 5.3.2.

[4] The *NEXT* function is modeled after FOLLOW sets from parsing theory, except that instead of returning a set of tokens, it returns a set of expressions.

Definition 5.2 (Sea Operator)**.** Given the definitions of $\approx$ and *NEXT*, we define the sea operator as follows: $\sim$e$\sim$ is a sequence expression

$\approx$(e / $next_1$ / $next_2$ / ... / $next_n$)
e
$\approx$($next_1$ / $next_2$ / ... / $next_n$)

where *NEXT(e)* $= \{next_1, next_2, \ldots, next_n\}$.

□

That is, the before-water consumes everything up to the island or the boundary, and the after-water consumes everything up to the boundary.

5.3.1 The *Water* Operator

The purpose of a water expression is to consume uninteresting input. Water consumes input until it encounters the expression specified in its argument (*i.e.*, the *boundary*). We must, however, take care to avoid the overlapping sea problem.

If two seas overlap (one sea is followed by another), the right boundary of the first sea starts with the second sea. Yet it should only start with the island of the second sea as illustrated in subsection 5.2.2. In order to do so, the second sea has to simply disable its before-water.

We detect overlapping seas as follows: if sea s_2 is invoked from the water of another sea s_1, it means that the water of s_1 is testing for its boundary s_2 and thus s_2 has to disable its before-water. To distinguish between nested seas (*e.g.*, $\sim$'x'($\sim$'y'$\sim$)'z'$\sim$) and overlapping seas (*e.g.*, $\sim$'x'$\sim$ $\sim$'y'$\sim$), we test the position where this sea was invoked. In the case of nested seas the positions differ, and in the case of overlapping seas they are the same.

Definition 5.3. (PEGs with Bounded Seas (BS-PEGs)) A parsing expression grammar with bounded seas — an extension of CS-PEGs (see Definition 4.1) — is a 6-tuple $G = (N, \Sigma, R, e_s, K, C_s)$ where N is a finite set of nonterminals, Σ is a finite set of terminal symbols, R is a finite set of rules, e_s is a starting expression, K is a finite set of combinators, C_s is an initial context — a finite set of an identifier to a stack mappings. Each context entry $c \in C$ is a pair (id, S). id identifies a particular stack S, we write this as S_{id}, i identifies an invocation stack.

Each rule $r \in R$ is a pair (A, e) which we write $A \leftarrow e$, $A \in N$ and e is a parsing expression. Parsing expressions (PEs) are defined inductively, if e_1 and e_2 are parsing expressions, then so is:

- ϵ , an empty string
- 't$^+$' , any literal, $t \in \Sigma$
- [t$^+$] , any character class, $t \in \Sigma$
- A , any nonterminal, $A \in N$
- e? , an optional expression
- e_1e_2 , a sequence

- $\texttt{e}_1/\texttt{e}_2$, a prioritized choice
- $\texttt{e}*$, a zero-or-more repetitions
- $!\texttt{e}$, a not-predicate
- $\&\texttt{e}$, an and-predicate
- $\texttt{k}$, a parser combinator, $k \in K$
- $\underline{\nabla}_{\texttt{id}}\texttt{e}$, a push to the stack S, $(id, S) \in C$
- $\underline{\triangle}_{\texttt{id}}$, a pop from the stack S, $(id, S) \in C$
- $\approx$, a water operator

□

Definition 5.4 (Semantics of BS-PEGs)**.** In order to detect overlapping seas and to compute the *NEXT* set, we extend the semantics of context-sensitive PEGs (see Definition 4.2) with a stack of invoked expressions and their positions S_I. For standard PEG operators there is no change except that an explicit invocation stack S_I is maintained. The relation $\Rightarrow$ has same inputs and outputs: from a triple of the form (e, x, C) to the output triples (o, y, C'), where e is a parsing expression, $x \in \Sigma^*$ is an input string to be recognized, o indicates the result of a recognition attempt, $y \in \Sigma^*$ is a remainder of input, C, C' are context mappings.

As in Definition 4.3, $\{.._C, S_{id}\}$ denotes a parsing context $C = \{(i_1, s_1), (i_2, s_2), ..., (i_n, s_n), (id, S)\}$ where $.._C$ is a shorthand for (i_1, s_1), $(i_2, s_2), ..., (i_n, s_n)$. Each context entry $c \in C$ is a pair (id, S). id identifies a particular stack S, we write this as S_{id},

$S_I \in C$ is an invocation stack of *expression–position* tuples $(e_n, p_n) : ... : (e_2, p_2) : (e_1, p_1) : []$, $[]$ denotes an empty stack, (e_1, p_1) the bottom element, (e_n, p_n) the top element, $((e, p) : S)$ denotes a stack with tuple (e, p) on the top and stack S below. S_I is initialized with the pair $(e_s, 0)$.

A function $pos : \Sigma^* \times \Sigma^* \rightarrow \mathbb{N}$ returns the position of y in the $input = xyz$ and, as a shorthand, we write p_x to refer to $pos(x, input)$. The distinguished symbol f indicates failure. We define $\Rightarrow$ inductively as follows (without any semantic changes for standard PEG operators):

Empty:

$$\frac{x \in \Sigma^*}{(\epsilon, x, \{.._C, S_I\}) \Rightarrow (\epsilon, x, \{.._C, S_I\})}$$

Terminal *(success):*

$$\frac{a \in \Sigma, x \in \Sigma^*}{(a, ax, \{.._C, S_I\}) \Rightarrow (a, x, \{.._C, S_I\})}$$

Terminal *(failure):*

$$\frac{a, b \in \Sigma, x \in \Sigma^*}{(a, bx, \{.._C, S_I\}) \Rightarrow (f, bx, \{.._C, S_I\})}$$

Nonterminal:

$$\frac{\begin{array}{c} S'_I = ((A, p_x) : S)_I \qquad A \leftarrow e \in R \\ (e, x, \{.._C, S'_I\}) \Rightarrow (o, y, \{.._{C'}, S'_I\}) \end{array}}{(A, x, \{.._C, S_I\}) \Rightarrow (o, y, \{.._{C'}, S_I\})}$$

Sequence *(success case):*

$$\frac{\begin{array}{c} S'_I = ((e_1e_2, p_x) : S)_I \\ (e_1, x, \{.._C, S'_I\}) \Rightarrow (o_1, y_1, \{.._{C_1}, S'_I\}) \\ (e_2, y_1, \{.._{C_1}, S'_I\}) \Rightarrow (o_2, y_2, \{.._{C_2}, S'_I\}) \end{array}}{(e_1e_2, x, \{.._C, S_I\}) \Rightarrow (o_1o_2, y_2, \{.._{C_2}, S_I\})}$$

Sequence *(failure 1):*

$$\frac{\begin{array}{c} S'_I = ((e_1e_2, p_x) : S)_I \\ (e_1, x, \{.._C, S'_I\}) \Rightarrow (f, x, \{.._C, S'_I\}) \end{array}}{(e_1e_2, x, \{.._C, S_I\}) \Rightarrow (f, x, \{.._C, S_I\})}$$

Sequence *(failure 2):*

$$\frac{\begin{array}{c} S'_I = ((e_1e_2, p_x) : S)_I \\ (e_1, x, \{.., S'_I\}) \Rightarrow (o, y_1, \{.._{C_1}, S'_I\}) \\ (e_2, y_1, \{.._{C_1}, S'_I\}) \Rightarrow (f, y_1, \{.._{C_1}, S'_I\}) \end{array}}{(e_1e_2, x, \{.., S_I\}) \Rightarrow (f, x, \{.._C, S_I\})}$$

Choice *(option 1):*

$$\frac{\begin{array}{c} S'_I = ((e_1/e_2, p_x) : S)_I \\ (e_1, x, \{.._C, S'_I\}) \Rightarrow (o, y, \{.._{C'}, S'_I\}) \end{array}}{(e_1/e_2, x, \{.._C, S_I\}) \Rightarrow (o, y, \{.._{C'}, S_I\})}$$

Choice *(option 2):*

$$\frac{\begin{array}{c} S'_I = ((e_1/e_2, p_x) : S)_I \\ (e_1, x, \{.._C, S'_I\}) \Rightarrow (f, x, \{.._C, S'_I\}) \\ (e_2, x, \{.._C, S'_I\}) \Rightarrow (o, y, \{.._{C'}, S'_I\}) \end{array}}{(e_1/e_2, x, \{.._C, S_I\}) \Rightarrow (o, y, \{.._{C'}, S_I\})}$$

Repetitions *(repetition):*

$$\frac{\begin{array}{c} S'_I = ((e*, p_x) : S)_I \\ (e, x, \{.._C, S'_I\}) \Rightarrow (o_1, y_1, \{.._{C_1}, S'_I\}) \\ (e*, y_1, \{.._{C_1}, S_I\}) \Rightarrow (o_2, y_2, \{.._{C_2}, S_I\}) \end{array}}{(e*, x, \{.._C, S_I\}) \Rightarrow (o_1o_2, y_2, \{.._{C_2}, S_I\})}$$

Repetitions *(termination):*

$$\frac{\begin{array}{c} S'_I = ((e*, p_x) : S)_I \\ (e, x, \{.._C, S'_I\}) \Rightarrow (f, x, \{.._C, S'_I\}) \end{array}}{(e*, x, \{.._C, S_I\}) \Rightarrow (\epsilon, x, \{.._C, S_I\})}$$

Not predicate *(success):*

$$\frac{\begin{array}{c} S'_I = ((!e, p_x) : S)_I \\ (e, x, \{.._C, S'_I\}) \Rightarrow (o, y, \{.._{C'}, S'_I\}) \end{array}}{(!e, x, \{.._C, S_I\}) \Rightarrow (f, x, \{.._C, S_I\})}$$

Not predicate *(failure):*

$$\frac{\begin{array}{c} S'_I = ((!e, p_x) : S)_I \\ (e, x, \{.._C, S'_I\}) \Rightarrow (f, x, \{.._C, S'_I\}) \end{array}}{(!e, x, \{.._C, S_I\}) \Rightarrow (\epsilon, x, \{.._C, S_I\})}$$

Parser combinators remain also unaffected. The parser combinator semantics is defined as follows:

Parser Combinator *(success):*

$$\frac{\begin{array}{c} S'_I = ((k_S(e), p_x) : S)_I \\ (e, x, \{.._C, S'_I, S_S\}) \Rightarrow (o_1, y_1, \{.._{C'}, S'_I, S'_S\}) \\ k(S'_S, o_1, y_1) = (o_2, y_2) \end{array}}{(k_S(e), x, \{.._C, S_I, S_S\}) \Rightarrow (o_2, y_2, \{.._{C'}, S_I, S'_S\})}$$

Parser Combinator *(failure 1):*

$$\frac{\begin{array}{c} S'_I = ((k_S(e), p_x) : S)_I \\ (e, x, \{.._C, S'_I, S_S\}) \Rightarrow (o_1, y, \{.._{C'}, S'_I, S'_S\}) \\ k(S'_S, o_1, y) = (f, y') \end{array}}{(k_S(e), x, \{.._C, S_I, S_S\}) \Rightarrow (f, x, \{.._C, S_I, S_S\})}$$

Parser Combinator *(failure 2):*

$$\frac{\begin{array}{c} S'_I = ((k_S(e), p_x) : S)_I \\ (e, x, \{.._C, S'_I, S_S\}) \Rightarrow (f, x, \{.._C, S'_I, S'_S\}) \end{array}}{(k_S(e), x, \{.._C, S_I, S_S\}) \Rightarrow (f, x, \{.._C, S_I, S_S\})}$$

The context manipulation operators operates as usually as well:

Push $\underline{\triangledown}$ *(success):*

$$\frac{\begin{array}{c} S'_I = ((\underline{\triangledown}_S e, p_x) : S)_I \\ (e, x, \{.._C, S'_I, S_S\}) \Rightarrow (o, y, \{.._{C'}, S'_I, S'_S\}) \end{array}}{(\underline{\triangledown}_S e, x, \{.._C, S_I, S_S\}) \Rightarrow (o, y, \{.._{C'}, S_I, (o : S')_S\})}$$

Push $\underline{\triangledown}$ *(failure):*

$$\frac{\begin{array}{c} S'_I = ((\underline{\triangledown}_S e, p_x) : S)_I \\ (e, x, \{.._C, S'_I, S_S\}) \Rightarrow (f, x, \{.._C, S'_I, S_S\}) \end{array}}{(\underline{\triangledown}_S e, x, \{.._C, S_I, S_S\}) \Rightarrow (f, x, \{.._C, S_I, S_S\})}$$

Pop $\underline{\triangle}$ *(success:)*

$$\frac{S_S = (e : S')_S}{(\underline{\triangle}_S, x, \{.._C, S_I, S_S\}) \Rightarrow (o, x, \{.._C, S_I, S'_S\})}$$

Pop $\underline{\triangle}$ *(empty case):*

$$\frac{S_S = []_S}{(\underline{\triangle}_S, x, \{.._C, S_I, S_S\}) \Rightarrow (f, x, \{.._C, S_I, S'_S\})}$$

□

A detailed example demonstrating how is S_I manipulated can be found in section B.3.

Definition 5.5 (Seas Overlap)**.** We define a function $\mathcal{SO}$ as follows:

$$\mathcal{SO}(S_I) = \begin{cases} \texttt{true} & S_I = ((\approx e, p) : S'_I) \wedge (\approx e', p) \in S'_I \\ \texttt{false} & \text{otherwise} \end{cases}$$

In other words, two seas overlap if there are two waters invoked at the same position, one of which is on the top of invocation stack S_I.

□

Definition 5.6 (Water Operator)**.** With the BS-PEGs we can define a prefix *water operator* $\approx$. It searches for a boundary and consumes input until it reaches a boundary. If the water starts a boundary of another sea, it stops immediately.

Water `≈` *(overlapping):*

$$\frac{\mathcal{SO}(S'_I)}{(\approx e, x, \{.._C, S_I\}) \Rightarrow (\epsilon, x, \{.._C, S_I\})}$$

Water `≈` *(boundary):*

$$\frac{\begin{array}{c} S'_I = ((\approx e, p_x) : S)_I \\ (e, y, \{.._C, S'_I\}) \Rightarrow (o, y', \{.._{C'}, S'_I\}) \\ (e, x''y, \{.._C, S'_I\}) \Rightarrow (f, x''y, \{.._C, S'_I\}) \\ \forall x'x'' : x = x'x'' \end{array}}{(\approx e, xy, \{.._C, S_I\}) \Rightarrow (x, y, \{.._C, S_I\})}$$

The boundary case of water consumes the shortest possible prefix x from input xy such that e succeeds on y.

□

Nested seas In the case of *directly nested seas* (*e.g.*, `∼∼island∼∼`) we obtain the same behavior as with `∼island∼`. Two seas overlap in the case a sea is directly invoked from another sea without consuming any input. Applying the rule *overlapping* from Definition 5.6, water of the inner sea is eliminated and the boundary is the same for the both seas. Therefore `∼∼island∼∼` is equivalent to `∼island∼`.

5.3.2 The *NEXT* function

Any input that can appear after a sea forms the boundary of the sea. The *NEXT* function returns a set of expressions that can appear directly after a particular expression. The *NEXT* set is inspired by the *FOLLOW* set from the standard parsing theory [GJ08b], which can be modified for PEGs [Red09]. Contrary to the follow set, which returns only terminal expressions, the next set returns terminal or composite expressions.

Consider the grammar in the example from Listing 5.7. The `code` rule is defined in such a way that it accepts an arbitrary number of class and structure islands in the beginning (classes and structures can be in any order) and there is a main method at the end. Intuitively, another class island, a structure island or a main method can appear after a class island.

The *NEXT* set approximates the boundary. Its expressions recognize prefixes of the boundary and not necessarily the whole boundary. The reason for using *NEXT* is the limited backtracking ability of PEGs. PEGs are not capable of taking globally correct decisions because they are not able to revert choices that have already been taken.[5]

For practical reasons, elements of *NEXT* cannot accept an empty string. For example, an optional expression is not a suitable approximation of a boundary because it succeeds for any input. Consider a simple expression:

```
∼island∼ 'a'? 'b'
```

The `'a'?` can appear after the `∼island∼` but `'b'` as well if `'a'` fails. Therefore *NEXT* has to return `'a'? 'b'`, not just `'a'?`. We use the abstract simulation (see Definition A.6) in order to recognize an expression that accepts an empty string as defined in Definition A.8.

[5] See for example: `http://www.webcitation.org/6YrGmNAi7`

```
code         ← (∼class∼ / ∼struct∼)* mainMethod
class        ← 'class'  ID classBody
stuct        ← 'struct' ID structBody
mainMethod   ← 'public' 'def' 'main' block

classBody    ← ...
structBody   ← ...
block        ← ...
ID           ← ...
```

Listing 5.7: Definition of code that consists of classes and structures followed by main method.

Static vs Dynamic *NEXT*

Similarly to *FOLLOW*, *NEXT* can be computed statically [GJ08a, Red09] or dynamically [SD96]. The dynamic way is more precise, but imposes run-time overhead because it requires as a parameter the up-to-date invocation stack S_I, which is available only at runtime. The static one is less precise, but does not impose any run-time overhead because it requires as parameters grammar G and expression e, which are available at any time. The better precision of the dynamic *NEXT* set is caused by the fact that dynamic *NEXT* knows from which parent an expression e is invoked. Static *NEXT* has to consider all the possible parents and take the union of the results of each of them. We present a computation for both versions, dynamic and static:

Definition 5.7 (Dynamic *NEXT*)**.** Let S be an invocation stack of *(expression, position)* pairs representing positions and invoked parsing expressions, where $S = ((e,p) : S')$ represents an S' with a (e,p) on top, `$$` is a special symbol signaling end of input, and $E_1 \times E_2$ is a product of two sets of parsing expressions E_1 and E_2, such that $E_1 \times E_2 = \{e_i e_j \mid e_i \in E_1, e_j \in E_2\}$, we define the dynamic *NEXT* set $\mathcal{N}_D(S)$ as a set of expressions such that:

Nonterminal

$$\frac{S = (e,p) : (A \leftarrow e, p') : S'}{\mathcal{N}_D(S) = \mathcal{N}_D((A,p') : S')}$$

Sequence *(case 1):*

$$\frac{S = (e_1,p) : (e_1e_2,p') : S' \qquad e_2 \not\rightharpoonup 0}{\mathcal{N}_D(S) = \{e_2\}}$$

Sequence *(case 2):*

$$\frac{S = (e_1,p) : (e_1e_2,p') : S' \qquad e_2 \rightharpoonup 0}{\mathcal{N}_D(S) = (\{e_2\} \times \mathcal{N}_D((e_1e_2,p) : S')}$$

Sequence *(case 3):*

$$\frac{S = (e_2,p) : (e_1e_2,p') : S'}{\mathcal{N}_D(S) = \mathcal{N}_D((e_1e_2,p) : S')}$$

Choice *(option 1):*

$$\frac{S = (e_1,p) : (e_1/e_2,p') : S'}{\mathcal{N}_D(S) = \mathcal{N}_D((e_1/e_2,p) : S')}$$

Choice *(option 2):*

$$\frac{S = (e_2,p) : (e_1/e_2,p') : S'}{\mathcal{N}_D(S) = \mathcal{N}_D((e_1/e_2,p') : S')}$$

Repetition *(case 1):*

$$\frac{S = (e,p) : (e*,p') : S' \qquad e \not\rightharpoonup 0}{\mathcal{N}_D(S) = \{e\} \cup \mathcal{N}_D((e*,p') : S')}$$

Repetition *(case 2):*

$$\frac{S = (e,p) : (e*,p') : S' \qquad e_2 \rightharpoonup 0}{\mathcal{N}_D(S) = \{e\} \times \mathcal{N}_D((e*,p') : S')}$$

Not Predicate

$$\frac{S = (e,p) : (!e,p') : S'}{\mathcal{N}_D(S) = \{\$\$\}}$$

Combinator

$$\frac{S = (e,p) : (k(e),p') : S'}{\mathcal{N}_D(S) = \mathcal{N}_D((k(e),p') : S')}$$

Push

$$\frac{S = (e,p) : (\underline{\triangledown}(e),p') : S'}{\mathcal{N}_D(S) = \mathcal{N}_D((\underline{\triangledown}(e),p') : S')}$$

Pop

$$\frac{S = (e,p) : (\underline{\vartriangle}(e),p') : S'}{\mathcal{N}_D(S) = \mathcal{N}_D((\underline{\vartriangle}(e),p') : S')}$$

Empty Stack *(termination):*

$$\frac{S = \{\}}{\mathcal{N}_D(S) = \{\$\$\}}$$

□

Definition 5.8 (Static *NEXT*). If $G = (N, \Sigma, R, e_s, K, C_s)$ is a BS-PEG, PE_G is a set of all parsing expressions in the grammar G, S is a subset of PE_G, $S \subseteq PE_G$, where $S = \{e : S'\}$ represents an S with an element e and $S' = S \setminus \{e\}$, `$$` is a special symbol signaling end of input, $\mathcal{P}$ is a parent function (see Definition A.4), and $E_1 \times E_2$ is a product of two sets of parsing expressions E_1 and E_2, such that $E_1 \times E_2 = \{e_i e_j \mid e_i \in E_1, e_j \in E_2\}$, we define the static *NEXT* set $\mathcal{N}_S(e, PE_G)$ of an expression e in a grammar G as a set of expressions such that:

Not a parent

$$\frac{S = \{e' : S'\} \qquad \neg\mathcal{P}(e,e')}{\mathcal{N}_S(e,S) = \mathcal{N}_S(e,S')}$$

Nonterminal

$$\frac{S = \{e' : S'\} \qquad e' \leftarrow e \qquad e \in \Sigma}{\mathcal{N}_S(e,S) = \mathcal{N}_S(e', PE_G) \cup \mathcal{N}_S(e,S')}$$

Sequence *(case 1):*

$$\frac{S = \{e' : S'\} \qquad e' = e_1 e_2 \qquad e_2 \not\rightharpoonup 0}{\mathcal{N}_S(e_1,S) = \{e_2\} \cup \mathcal{N}_S(e_1,S')}$$

Sequence *(case 2):*

$$\frac{S = \{e' : S'\} \qquad e' = e_1 e_2 \qquad e_2 \rightharpoonup 0}{\mathcal{N}_S(e_1,S) = (\{e_2\} \times \mathcal{N}_S(e', PE_G)) \cup \mathcal{N}_S(e_1,S')}$$

Sequence *(case 3):*

$$\frac{S = \{e' : S'\} \qquad e' = e_1 e_2}{\mathcal{N}_S(e_2, S) = \mathcal{N}_S(e', PE_G) \cup \mathcal{N}_S(e_2, S')}$$

Choice *(option 1):*

$$\frac{S = \{e' : S'\} \qquad e' = e_1/e_2}{\mathcal{N}_S(e, S) = \mathcal{N}_S(e', PE_G) \cup \mathcal{N}_S(e_1, S')}$$

Choice *(option 2):*

$$\frac{S = \{e' : S'\} \qquad e' = e_1/e_2}{\mathcal{N}_S(e, S) = \mathcal{N}_S(e', PE_G) \cup \mathcal{N}_S(e_2, S')}$$

Repetition *(case 1):*

$$\frac{S = \{e' : S'\} \qquad e' = e* \qquad e \not\rightharpoonup 0}{\mathcal{N}_S(e, S) = \{e\} \cup \mathcal{N}_S(e', PE_G) \cup \mathcal{N}_S(e, S')}$$

Repetition *(case 2):*

$$\frac{S = \{e' : S'\} \qquad e' = e* \qquad e_2 \rightharpoonup 0}{\mathcal{N}_S(e, S) = (\{e\} \times \mathcal{N}_S(e', PE_G)) \cup \mathcal{N}_S(e', PE_G) \cup \mathcal{N}_S(e, S')}$$

Not Predicate

$$\frac{S = \{e' : S'\} \qquad e' = !e}{\mathcal{N}_S(e, S) = \{\$\$\} \cup \mathcal{N}_S(e, S')}$$

Combinator

$$\frac{S = \{e' : S'\} \qquad e' = k(e)}{\mathcal{N}_S(e, S) = \mathcal{N}_S(e', PE_G) \cup \mathcal{N}_S(e, S')}$$

Push

$$\frac{S = \{e' : S'\} \qquad e' = \underline{\triangledown}(e)}{\mathcal{N}_S(e, S) = \mathcal{N}_S(e', PE_G) \cup \mathcal{N}_S(e, S')}$$

Pop

$$\frac{S = \{e' : S'\} \qquad e' = \underline{\vartriangle}(e)}{\mathcal{N}_S(e, S) = \mathcal{N}_S(e', PE_G) \cup \mathcal{N}_S(e, S')}$$

Start Expression

$$\frac{e = e_s}{\mathcal{N}_S(e, S) = \{\$\$\} \cup \mathcal{N}_S(e, S')}$$

Empty Set *(termination):*

$$\frac{S = \{\}}{\mathcal{N}_S(e, S) = \{\}}$$

□

Examples of dynamic and static computation can be found in section B.1 and section B.2

FOLLOW vs. *NEXT*

The *NEXT* function introduces extra complexity into bounded seas, even though it resembles the *FOLLOW* function from LL parsing theory [GJ08a, pp. 235-361]. The key difference between *FOLLOW* and *NEXT* is that the former returns only terminals, while the latter returns parsing expressions.

Why is it not sufficient to use the well-known *FOLLOW* sets instead of the more complicated *NEXT* function? The reason is that the right context (boundary) of a sea is in general an $LL(k), k \geq 1$ language, and a simple *FOLLOW* set is not usually sufficient to recognize the boundary.

As an example, consider the grammar from Listing 5.7. The boundary of `class` is *NEXT*(`class`) = { `∼class∼` , `∼struct∼` , `mainMethod` }. Suppose that instead we take as the boundary of `class` its *FOLLOW* set, *i.e.*, *FOLLOW*(`class`) = { `'class'` , `'struct'` , `'public'` }. If there are other elements in the input that start with `'public'` (*e.g.*, `'public int i = 0;'`), they will be indistinguishable from the `mainMethod` and the water of bounded seas would finish in an invalid position.

Bounded seas are supposed to work only with a skeleton of an original grammar with as little information as possible. Therefore, information about other input that can interfere with a boundary (*e.g.*, `'public int i = 0;'`) is not usually available. If bounded seas are are provided with a baseline grammar this would not be problem as the techniques described by Klusener and Lämmel [KL03] can then be applied.

5.3.3 BS-PEG analysis

In this section we extend the definitions of a first set, an abstract simulation, and a *push–pop* analysis to support BS-PEGs.

Definition 5.9 (First set with Bounded Seas)**.** To support the first set analysis of BS-PEGs, we extend the definition of first (see Definition 4.6) with a first rule for a water operator.

Water Operator

$$\frac{}{\textit{FIRST}(\approx(e)) = \{x \mid \forall x \in \Sigma\} \cup \{\epsilon\}}$$

Meaning that water can start with anything or can accept an empty string, no matter what is the underlying expression e. □

Definition 5.10 (Abstract Simulation with Bounded Seas)**.** To support the abstract simulation in BS-PEGs, we extend the relation $\rightharpoonup$ from Definition 4.7 as follows:

11. (a) $\approx(e) \rightharpoonup 1$

    (b) $\approx(e) \rightharpoonup 0$

Meaning water never fails and may accept an empty string.

□

Definition 5.11 (*Push–pop* analysis with Bounded Seas)**.** Even though the BS-PEGs manipulate the invocation stack, the rules always keep the stack in its original state, *i.e.*, in the state as upon invocation. Therefore, the *push–pop* analysis does not change and we extend the relation $\hookrightarrow$ from Definition 4.8 with water operator $\approx$ as follows:

11. $\approx(e) \hookrightarrow o$ if $e \hookrightarrow_S o$

Meaning water (though context-sensitive) does not modify context and always keeps it in the state as upon invocation.

□

5.4 Implementation

We implement the sea operator in a single combinator that represents the water–island–water sequence. The `BoundedSea` parser is defined as in Listing C.11. Even though a bounded sea consists of a sequence of three parsers, it has only one instance variable `island` (see Listing C.10), before-water and after-water being created automatically depending on the grammar. The `parseOn:` method of `BoundedSea` is in Listing C.11. The three phases of `parseOn:` correspond to the phases in Definition 5.1. The check for overlapping seas is in `parseBeforeWater:` (see Listing C.12). The remaining semantics of water is a simple search for a boundary represented by a `goUpTo:` method.

The *NEXT* function

We experimented with both implementations of *NEXT*: dynamic *NEXT* (see Definition 5.7) and static *NEXT* (see Definition 5.8). To compute the dynamic *NEXT* set we use the method invocation stack (do not confuse with the invocation stack in `Context`), which is a first class entity in Pharo and which is accessed using the `thisContext` pseudo-variable and is managed by the Pharo runtime. The dynamic *NEXT* implementation follows straightforwardly from the recursive Definition 5.7 (see Listing C.13). We use `acceptsEpsilon` to determine if a parser can succeed without consuming input (as defined in Definition A.8).

To compute static *NEXT*, we hook into the `parse:` method of a root parser (see section 3.2). From this root, we access all the parsing expressions PE and compute static *NEXT*. The static *NEXT* implementation follows straightforwardly from the recursive Definition 5.8 (see Listing C.14).

Our current implementation utilizes the static *NEXT* set. The static *NEXT* provides better performance, because the overhead can be moved from parse time to compile time. Nevertheless, the language engineers should not reuse the same instance of a sea from multiple nonterminals to obtain better precision as discussed in subsection 5.3.2.

Overlapping Seas

The implementation of bounded seas in PetitParser requires an invocation stack in the parsing context (represented by the `Context` class) to detect overlapping seas. Contrary to the method invocation stack, the invocation stack in the parsing context contains invocation positions. The context invocation stack is managed by the following methods of `Context`: `invoked:`, `return:`, and `fail:` that push to and pop from the invocation stack the positions of bounded seas (see Listing C.15). Overlapping seas can be detected trivially (see Listing C.16) by comparing the two top positions.

Memoization

For recursive structures such as nested blocks, bounded seas tend to invoke the same sea on the same position over and over again. This leads to exponential parse times. We use the idea of packrat parsing [For02a] to prevent this. We implement a memoized version of seas, which remembers a result after being invoked at a given position and returns the cached result in a subsequent invocation for that position, the same way as the memoizing parser (see section 3.2) works.

5.4.1 Performance

In this section we briefly report on the performance of bounded seas. We use static *NEXT* and the memoized version of seas. We focus on the time complexity of the three different placements of a sea: standalone seas, repetition of a sea and a nested sea. In order to show the overhead of our implementation of bounded seas, we compare with equivalent island grammars (*i.e.*, using an "advanced" type of an island grammar as in Listing 5.4). We perform measurements on the following parsers and inputs:

Stand-alone sea $\sim'\texttt{a}'\sim$ searches for the island `'a'` in input. Input consists of randomly generated string of dots `.` (representing water) and a single character `'a'` at a random position.

Repetition of a sea $\sim'\texttt{a}'+\sim$ searches for sequences of islands `'a'` in input. Input consists of a randomly generated string of dots `.` (representing water) and island characters `'a'`, *e.g.*, `'..a.....a....a...aa..'`.

Nested sea $\texttt{block} \leftarrow \sim'\{'\texttt{block} + /\sim \epsilon \sim '\}'\sim+$ searches for sequences of nested blocks in input. Input consists of blocks starting with `'{'` and ending with `'}'`. A block contains a possibly empty sequence of other blocks, for example: `'{...{}.{.{..}.}...}'`.

Figure 5.1 shows that, in the case of all the tested grammars, the time complexity of seas is linear compared to the input size. For all the sea grammars used in this work we measure the linear asymptotic time. Nevertheless the parse time is high, approximately ten times higher than its island variant. This is rather high price for the comfort that bounded seas offer. We show how to completely eliminate the overhead of bounded seas in subsection 6.4.6.

5.5 Java Parser Case Study

The goal of this case study is to demonstrate the suitability of bounded seas for extracting data from Java sources without any baseline grammar provided. First we focus on a simpler task without considering nested classes. Because bounded seas target extensibility we subsequently investigate the effort required to extend the parser with nested classes. In the last step we focus on extracting return types of methods.

We compare four kinds of Java parsers and we measure how well can they extract classes and their methods from a Java source code:

Java Parser is an open-source Java parser using PetitParser [RDGN10] provided by the Moose analysis platform community [NDG05]. We used version 167.[6]

Naïve Islands is an island parser with water defined simply as the negation of the island we are searching for. The sea rules in this parser can be reused, because they do not consider their surroundings and they are grammar-independent. The sea rules are defined in a simple form: consume input until an island is found, then consume an island.

[6] http://smalltalkhub.com/#!/~Moose/PetitJava/

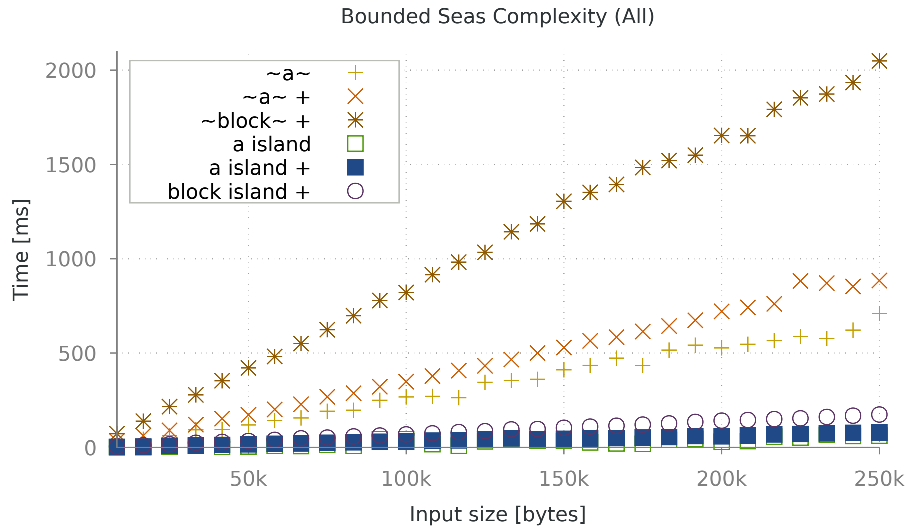


Figure 5.1: The performance comparison of memoized bounded seas for a stand-alone sea $\sim'\mathtt{a}'\sim$, a repetition of a sea $\sim'\mathtt{a}'\sim +$ and for a nested sea and their island variants on randomly generated inputs.

Advanced Islands is a more complex version of the naïve island parser. The water is more complicated to prevent the most frequent failures of island parsers. The sea rules in this parser are hard-wired to the grammar and cannot be reused. The sea rules are customized for a particular islands.

Sea Parser is a parser written using bounded seas. The sea rules were defined using the sea operator.

The Java Parser parses Java 6 code. All the semi-parsers (naïve, advanced and sea) are identical, with approximately 20 rules per each and are written by the author of this thesis. They differ only in the way the seas and islands are constructed. PetitJava itself contains over 200 rules. The semi-parsers were designed to extract classes and the methods that belong to them.

It is very likely that the advanced island parser can be modified to achieve better precision, but at the cost of considerable engineering work. We demonstrate that naïve water rules do not work and that the advanced version of water is needed. We further show that with bounded seas we can obtain high precision and performance without needing to define an advanced island parser, outperforming even the PetitJava parser because semi-parsers provide more robustness. Finally, we show that extending an island parser is a highly demanding task, unless bounded seas are used.

Test Data For our case study selected 1129 files defining a class (N) from the JDK 6 library. These files contain 1666 classes and subclasses and 20828 methods M in total. We extract the reference data using the VerveineJ[7] parser and store it as a Moose model [ND04].

[7] `http://www.webcitation.org/6k64XjAki`

Each parser returns an AST containing `JavaDefinition` nodes with a list of `JavaMethod` nodes out of which we extract a set of m fully-qualified method names,[8] some of which are true positives m_{tp}. If a parser fails, the set of extracted methods is empty. We measure precision $P = |m_{tp}|/|m|$) and recall $R = |m_{tp}|/|M|$.

5.5.1 Without Nested Classes

First of all, we evaluate our parsers on extracting method names without considering the nested classes and their methods. We can easily skip the nested classes by defining properly paired blocks starting with `'{'` and ending with `'}'` and ignoring everything inside.

Results As we see in Figure 5.2, the Java Parser provides perfect precision, but recall is bad because of many failures.[9] On the other hand, all the island parsers (island, advanced and bounded) are very robust, they almost never fail, but neither precision nor recall are perfect, even though they are relatively high. Amongst the imprecise parsers, the Bounded parser provides the best precision and recall, even outperforming the Java Parser in terms of F_1-measure.[10] The detailed numbers are provided in Table 5.2.

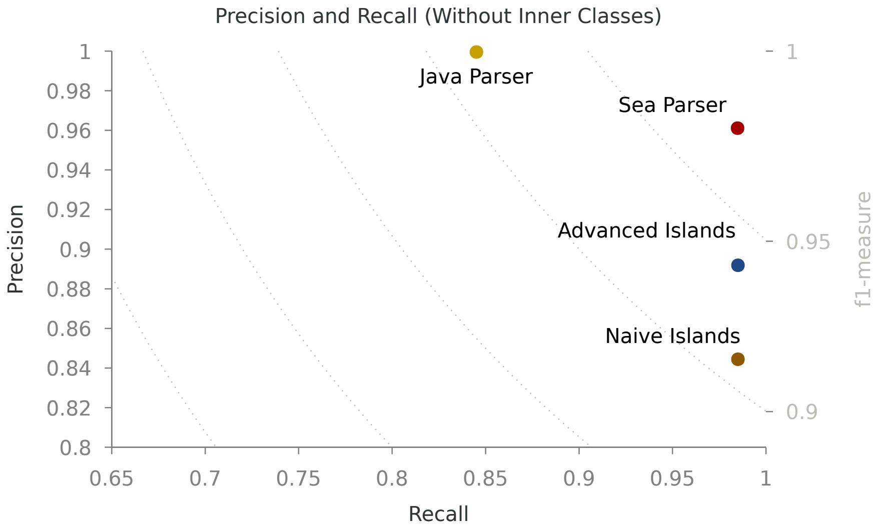


Figure 5.2: Precision and recall of different versions of Java parser in extracting method names, the nested classes are excluded.

5.5.2 With Nested Classes

In this step, we extend our island parsers to include nested classes and their methods. We do this by making a single change, where we extend the `classBody` rule from

[8] `http://www.webcitation.org/6k64hqyyu`

[9] The Java Parser failures are due to bugs in the grammar specification.

[10] `http://www.webcitation.org/6k64iiBRB`

Parser	Precision	Recall
Java Parser	1.00	0.84
Islands	0.84	0.98
Advanced islands	0.89	0.98
Sea Parser	0.96	0.98

Table 5.2: Precision and recall of the four tested parsers without considering nested classes.

this:[11]

```
classBody ← '{' method island * '}'
```

to this:

```
classBody ← '{' (method / class) island * '}'
```

Results As we see in Figure 5.3 the Java Parser performs as in the previous case. Yet the imprecise parsers (Island, Advanced) start to struggle. Their recall has dropped. On the other hand, the Sea Parser maintains high precision and recall and still outperforms the Java Parser.

We also measured the *Refined* parser, which made use of refined rules for water to take into account the grammar changes.[12] This improved recall and parsing time. We would, however, need to invest even more effort to reach the quality of the Sea Parser. The detailed numbers are provided in Table 5.3.

Parser	Precision	Recall
Java Parser	1.00	0.82
Islands	0.83	0.77
Advanced Islands	0.87	0.70
Refined Islands	0.88	0.86
Sea Parser	0.94	0.98

Table 5.3: Precision and recall of the four tested parsers with considering nested classes.

5.5.3 With Return Types

Some might object that the results of the naïve parser are reasonably good considering the low effort to implement such a parser. This is because the naïve parser is very good in searching for the *identifier–brackets* pattern, since methods are defined as follows:

```
methodDef   ← (modifiers island) (returnType island) id args
```

Such a definition of a naïve parser works when detecting method names, but it does not work well with return types. In a typical scenario `methodDef` accepts (i) an empty

[11] `island` creates either an island, an advanced island or a bounded sea depending on the parser we use.

[12] We investigated the reasons for failures and added an extra boundary to `classBody` .

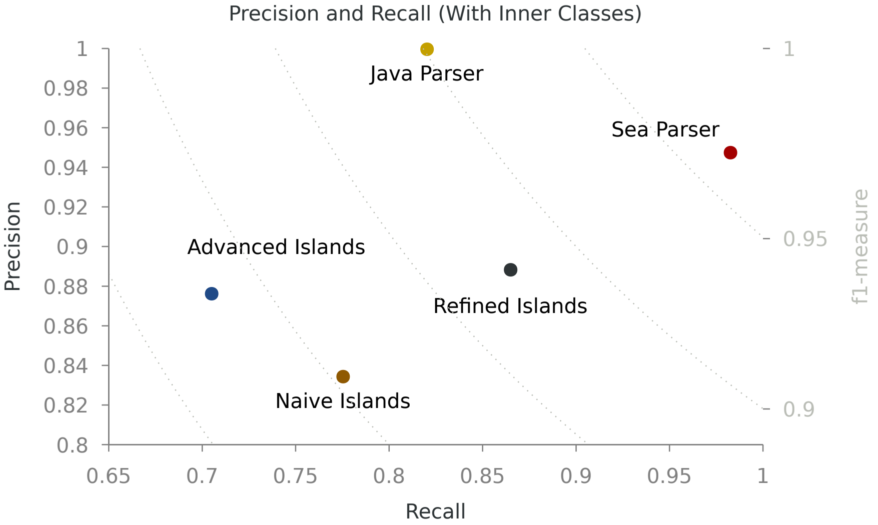


Figure 5.3: Precision and recall of different versions of Java parser in extracting method names, the nested classes are included.

string as modifiers; (ii) any identifier as a return type; and (iii) skips almost anything until another identifier followed by arguments is found. Other parsers do not suffer from this problem because their boundaries prevent them from skipping over relevant parts of input.

Results The result with nested classes can be seen in Figure 5.4. The naïve parser cannot extract the return type, while all the other parsers work reasonably well without any modifications. The decreased precision (see Table 5.4) is caused by array types that are not properly handled either by the Java parser or by the island or sea parsers.

Parser	**Precision**	**Recall**
Java Parser	0.94	0.77
Islands	0.05	0.05
Advanced Islands	0.79	0.64
Refined Islands	0.80	0.79
Sea Parser	0.89	0.93

Table 5.4: Precision and recall of the four tested parsers with considering nested classes and return types.

5.5.4 Performance

To compare the performance of the bounded seas with other parsers we present a graph measuring the time to extract method names with return types and nested classes based on the file size. We visualize the dependency between the file size and time to parse for Java, naïve, advanced′ and sea parsers in Figure 5.5. Time-wise, the bounded seas

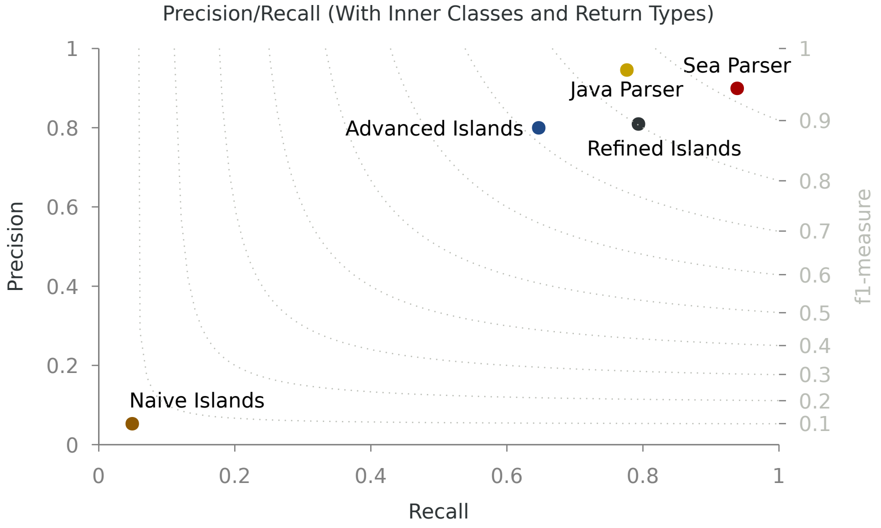


Figure 5.4: Precision and recall of different versions of Java parser in extracting method names and return types. The nested classes are included.

perform approximately three times slower than the their island variant and with linear asymptotic complexity. We discuss performance optimizations in subsection 6.4.6.

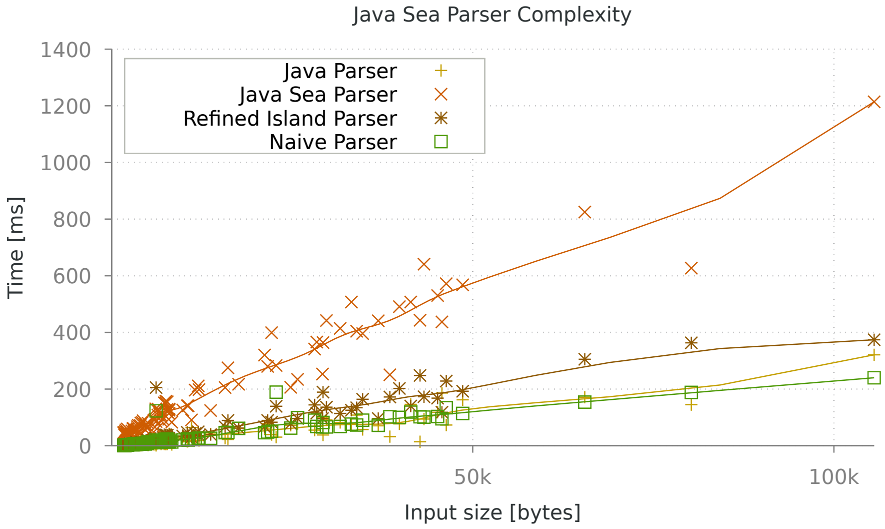


Figure 5.5: Dependency between parse time and input size for different versions of Java parsers.

5.6 Related Work

In this section we discuss other semi-parsing approaches.

Agile Parsing

Agile parsing [DCMS03] is a paradigm for source analysis and reverse engineering tools. In agile parsing the effective grammar used by a particular tool is a combination of two parts: the standard base grammar for the input language, and a set of explicit grammar overrides that modify the parse to support the task at hand. There are several agile parsing idioms: i. *rule abstraction* (grammar rules can be parametrized); ii. *grammar specialization* (grammar rules can be specialized based on the semantic needs); iii. *grammar categorization* (to deal with context-free ambiguities); iv. *union of grammars* (to unify multiple grammars); v. *markup* (to match and mark chunks of interest); vi. *semi-parsing* (to define islands and lakes); and vii. *data structure grammars* (separate grammars that hold auxiliary data structures).

The semi-parsing idiom [DCMS03] uses the *not* predicate to prevent water from consuming islands. This approach is the same as that taken by bounded seas. Contrary to the semi-parsing idiom, bounded seas are able to infer the predicates on their own. The agile parsing idioms are based on a transformation of a well-defined baseline grammar, whereas bounded seas do not expect such a well-defined grammar and must infer the predicates only from the available *skeleton*.

Island Grammars

Island grammars were proposed by Moonen [Moo01] as a method of semi-parsing to deal with irregularities in the artifacts that are typical for the reverse engineering domain. Island grammars make use of a special syntactic rule called *water* that can accept any input. Water is annotated with a special keyword `avoid` that will ensure that water will be accepted only if there is no other rule that can be applied.

Contrary to Moonen, we propose boundaries (based on the *NEXT* function) that limit the scope in which water can be applied. Because each island has a different boundary, our solution does not use the single water rule; instead our water is tailored to each particular island. Therefore bounded seas offer much better composability than island grammars.

Non-Greedy Rules

Non-greedy operators are well-known from regular expressions introduced in Perl. [13] `??`, `*?`, and `+?` are non-greedy versions of `?`, `*` and `+`, which match as little of a string as possible while preserving the overall match. The backtracking algorithm admits a simple implementation of non-greedy operators: try the shorter match before the longer one. For example, in a standard backtracking implementation, `e?` first tries using `e` and then tries not using it; `e??` uses the other order.[14]

Non-greedy operators are also available in ANTLR as parser operators. A non-greedy parser matches the shortest sequence of tokens that preserves a successful parse for a valid input sentence. Contrary to regular expressions, a non-greedy parser never makes a decision that will ultimately cause valid input to fail later on during the parse.

[13] `http://www.webcitation.org/6k63y2Dqf`

[14] `http://www.webcitation.org/6ivDnxZIf`

The central idea is to match the shortest sequence of tokens that preserves a successful parse for a valid input sentence.[15]

Bounded seas are distinct from non-greedy rules in two ways. First, bounded seas do not require globally correct decisions, since they are not available in traditional PEGs. Though PEGs can backtrack while choosing between alternatives, once the choice is made it cannot be changed, thus making a globally correct decision impossible. In order to realize non-greedy repetitions, PEGs utilize predicates, which have to be specified by a language engineer (as illustrated in section 5.1). Bounded seas remove the burden of predicates from a language engineer by computing the *NEXT* set automatically.

Second, bounded seas target transparent composability. A language engineer can treat a bounded sea like any other PEG rule. For example, we can define a sea with an island as `∼island∼` and use the `sea` from any parsing expression. With such a definition we can change the `body` rule in the following grammar

```
start  ←  ('begin' body 'end')*
body   ←  sea
sea    ←  ∼island∼
```

to `body ← sea*`, `body ← sea?` or `body ← sea?sea?` without any further modifications.

When using the lazy predicates of context-free grammars, the definitions of the rules are more complicated and differ for each of the `body` variants.

If we define `sea` using non-greedy repetition `*?`, The EBNF equivalent of `body ← sea` is defined as:

```
start  →  ('begin' body 'end')*
body   →  sea
sea    →  ●*? 'body' ●*?
```

The EBNF equivalent of `body ← sea?` is defined as:

```
start  →  ('begin' body)*
body   →  sea
sea    →  ●*? ('body'|'end')
```

The EBNF equivalent of `body ← sea*` is defined as:

```
start  →  ('begin' body 'end')*
body   →  sea
sea    →  ('body'| ●)*?
```

And the EBNF equivalent of a sequence of two optional seas `body ← sea?sea?` as:

```
start  →  ('begin' body)*
body   →  sea1
sea1   →  ●*?('body1' sea2   | 'body2' 'end'   | 'end')
sea2   →  ●*?('body2' 'end'  | 'end')
```

The snippets illustrate that bounded seas offer much easier and more transparent composability compared the non-greedy operators of CFGs.

[15] http://www.webcitation.org/6ivEIY4AO

Noise Skipping Parser

GLR* is a noise-skipping parsing algorithm for context-free grammars able to parse any input sentence by ignoring unrecognizable parts of the sentence [LT93]. The parser nondeterministically skips some words in a sentence and returns the parse with fewest skipped words. The parser is a modification of the Generalized LR (Tomita) parsing algorithm [Tom85].

The GLR* application domain is parsing of spontaneous speech. Contrary to bounded seas, GLR* itself decides what is noise (water in our case) and where it is. In the case of bounded seas the placement of noise (water) is explicitly stated. While noise skipping parsers are tailored to GLR and speech recognition, bounded seas target agile modeling, can be used with PEGs, are deterministic and allow a grammar engineer to control when to skip input.

Fuzzy Parsing

The term fuzzy parser was coined for Sniff [Bis92], a commercial C++ IDE that uses a hand-made top-down parser. Sniff can process incomplete programs or programs with errors by focusing on symbol declarations (classes, members, functions, variables) and ignoring function bodies. In linguistics or natural language processing [Asv95], the notion of fuzzy parsing corresponds to an algorithm that recognizes fuzzy languages.

The semi-formal definition of a fuzzy parser was introduced by Koppler [Kop97]. Fuzzy parsers recognize only parts of a language by means of an unstructured set of rules. Compared with whole-language parsers, a fuzzy parser remains idle until its scanner encounters an anchor in the input or reaches the end of the input. Thereafter the parser behaves like a normal parser. In the fuzzy parsing framework, islands can occur in any order, always start with a terminal and everything between them is ignored; in the bounded seas paradigm, islands are constrained by the context-free structure of the grammar, which makes bounded seas more suitable when extracting structured data and thus more suitable for agile modeling.

Skeleton Grammars

Skeleton grammars [KL03] address the issue of false positives and false negatives when performing tolerant parsing by inferring a tolerant (skeleton) grammar from a precise baseline grammar.

Our approach tackles the same problem as skeleton grammars: improving the precision of island grammars. They both maintain the composability property and both can be automated. Skeleton grammars use the standard first and follow sets known from standard parsing theory [GJ08a, pp. 235-361] for synchronization with the baseline grammar.

Bounded seas, however, do not require a precise baseline grammar and they have to find the point of synchronization based only on the the main grammar itself. Therefore the main grammar has to contain all the relevant information (*e.g.*, when extracting classes and methods with bounded seas block definitions are essential to place methods properly). Because the main grammar of bounded seas is typically far from complete, bounded seas use the *NEXT* set (instead of first and follow) to reach the required precision. If bounded seas are provided with a baseline grammar, boundaries can be precisely computed from the baseline. The fact that bounded seas do not require the baseline grammar makes them suitable for agile modeling, which focuses on incremental development of a grammar from scratch.

Bridge Parsing

Bridge parsing is a novel, lightweight recovery algorithm that complements existing recovery techniques [NNEH09]. Bridge parsing extends an island grammar with the notion of bridges and reefs. Islands denote tokens that open or close scopes. Reefs are attributed tokens that add information (*e.g.*, indentation) to nearby islands. Islands and reefs are created in a tokenizing phase. Bridges connect matching opening and closing islands in a bridge-building phase. The corresponding islands are searched with the help of reefs (*e.g.*, indentation can be used to find matching brackets). If some islands are not connected (*e.g.*, if the opening or closing scope island is missing), the bridge repair phase tries to repair them with the help of information from reefs.

The focus of bounded seas is on data extraction rather than on error recovery and bounded seas are missing advanced error-recovery techniques available in the bridge parsing. Bounded seas are meant to be used on valid inputs without errors. If an erroneous chunk appears, bounded seas skip such a chunk until a valid chunk is found. To our best knowledge, techniques used in bridge parsing are complementary to bounded seas and might help improve precision of bounded seas on erroneous inputs.

Permissive Grammars The idea of a permissive grammar [KdJNNV09, JKSV12] is to extend an existing baseline grammar so that it accepts inputs with minor errors (*e.g.*, missing brackets, semicolons, *etc.*). A permissive grammar is also a normal grammar and can be tweaked by a language engineer. Using a specialized version of the GLR algorithm, both syntactically correct and incorrect programs can be efficiently parsed using these grammars [KdJNNV09].

Contrary to bounded seas, which target the area of rapid data extraction, permissive grammars are supposed to help IDE developers with interactive parsing and error recovery as the user is writing a program. Similarly to bounded seas, permissive grammars extend the concept of island grammars and use water for error recovery. Even though bounded seas can be used to skip over noise in input, a bounded sea handles missing or misspelled input simply by ignoring the whole erroneous chunk until a valid chunk is found. Permissive grammars try to find the best way to fix an erroneous chunk (and not only skip over it). Bounded seas fit better the domain of agile modeling, as they do not target error recovery, can skip large chunks of code and do not require a baseline grammar.

5.7 Conclusion

In this chapter we introduced bounded seas — a composable, reusable, robust and easy to use semi-parsing technique focused on reverse engineering. We show that with bounded seas it is possible to create a robust, precise and easy to extend parser for extracting data from a source code without any baseline grammar provided. In the concrete example of Java parsers in Pharo, bounded seas outperform any other parser available in terms of precision and recall. Because of context-sensitivity of bounded seas, they suffer from a worse performance than their island counterparts.

The implementation, tests, validations, benchmarks and measurements from this chapter can be re-run. The prepared image is available online.[16]

[16] http://scg.unibe.ch/research/parsingForAgileModeling

6 Adaptable Parsing Strategies

In the previous chapters we focused on flexibility of a framework for agile modeling. We use scannerless parsing to reduce compositional obstacles and deal with lexical ambiguities and we utilize a PEG-based parser combinator framework to allow for modularity and extensibility. We extended PEGs with (i) context-sensitive operators to allow for indentation and other context-sensitive grammar definitions; and (ii) bounded seas to allow for tolerant and composable grammar definitions.

Yet, these technologies comes at a price of lesser efficiency. With the scannerless nature of PEGs their complex formalism is used to parse tokens that could be recognized by finite state machines. Unlimited lookahead of PEGs comes at a cost of constant backtracking even when parsing deterministic languages. With the context-sensitive extension complex context-aware mementos are used even in cases when a context is not modified and for the same reason bounded seas suffer from a worse performance compared to their island variants. Last but not least, there is a powerful Turing-equivalent formalism behind every parser combinator while only a fraction of its capabilities is utilized most of the time.

Scanning parsers offer excellent performance thanks to the scanner-parser pipeline, which utilizes two different parsing strategies. First a lexical strategy recognizes regular elements — tokens — and later these tokens are used to recognize a context-free structure. Such an approach is however static, as the two phases are strictly separated and the set of applicable grammars is limited.

In our approach we extend the idea of different parsing strategies. We combine multiple parsing strategies in a single parsing framework that switches between these strategies dynamically, back and forth, during a single parsing phase. The framework selects the most appropriate parsing strategy for each nonterminal and switches to the selected strategy whenever the nonterminal is being parsed.

We present a *parser compiler*, a source-to-source translator that applies different parsing strategies for different parts of a grammar. Based on a compile-time analysis of a grammar, a parser compiler switches between a parser and a recognizer to avoid superfluous object allocations, applies simpler and more efficient parsing strategies to recognize tokens, optimizes choices to minimize backtracking, applies context-sensitive memoization only when necessary, and reduces the composition overhead of parser combinators while preserving their flexibility.

The result of *“compilation”* is an optimized top-down parser that provides performance comparable to a hand-written code. In particular, based on our benchmarks covering several different grammars for PetitParser,[1] a parser compiler offers a performance improvement by a factor ranging from two to twenty-five, depending on a grammar. Based on our Smalltalk case study, our approach is 10% slower compared to a highly-optimized, hand-written parser.

The chapter is organized as follows: We explain parsing overhead of parser combinators on a concrete example of a grammar defined in PetitParser in section 6.1. In section 6.2 we introduce a parser compiler, which eliminates the overhead of PetitParser. In section 6.3 we describe in detail how do we identify and apply different parsing strategies. In section 6.4 we analyze performance of a parser compiler in different configurations. In section 6.5 we briefly discuss related work and finally, section 6.6 concludes this chapter.

6.1 Motivating Example

A grammar fragment written in DSL used by PetitParser (see Table 3.2) describing a simple programming language is shown in Listing 6.1.

```
letterOrDigit←  #letter / #digit
identifier   ←  (#letter letterOrDigit*)
idToken      ←  identifier token
classToken   ←  ('class' &#space) token

class        ←  classToken  idToken  body
         map: [:classToken :idToken :body |
                ClassNode new
                    name: idToken value;
                    body: body
              ]
body         ←  indent
                    (class / method)*
                dedent
program      ←  class+
```

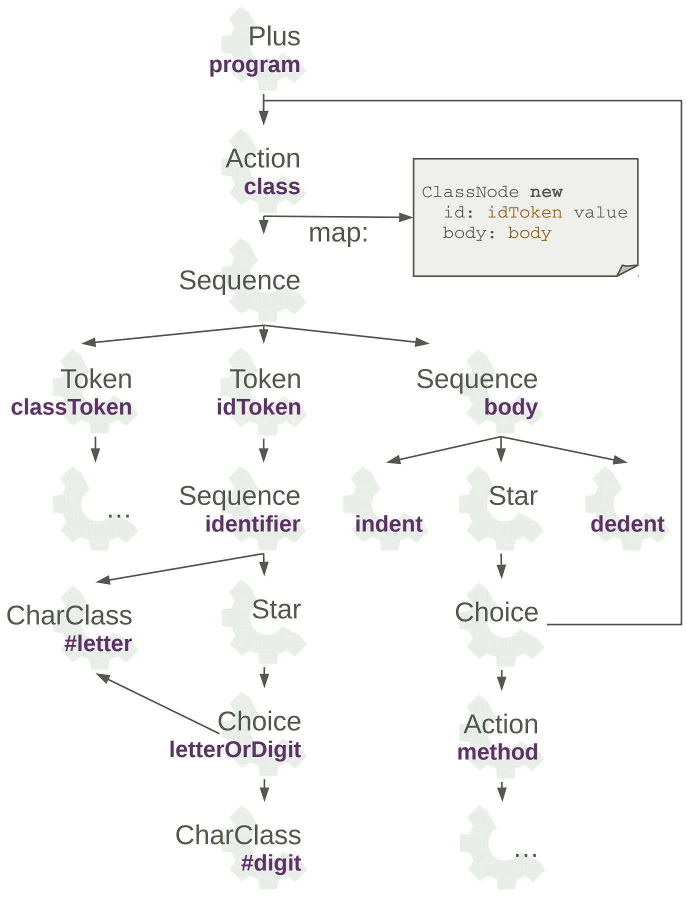
Listing 6.1: Simple grammar in PetitParser DSL.

A program in this grammar consists of a non-empty sequence of classes. A class starts with a `classToken`, followed by an `idToken` and a `body`. The `classToken` rule is a `'class'` keyword followed by a space. Identifiers start with a letter followed by any number of letters or digits. Class keyword and identifiers are transformed into instances of `Token`, which keep information about start and end positions and the string value of a token. There is a semantic action associated to a `class` rule that creates an instance of `ClassNode` filled with an identifier value and a class body.

[1] arithmetic expressions, Java, Smalltalk and Python

A class body is indentation-sensitive, *i.e.*, `indent` and `dedent` determine the scope of a class (instead of commonly used brackets *e.g.*, `{` and `}`). The `indent` and `dedent` rules determine whether a line is indented, *i.e.*, on a column strictly greater than the previous line or dedented, *i.e.*, on a column strictly smaller than the previous line. The class body contains a sequence of classes and methods.

Combinators Composition Figure 6.1 shows a composition of parser combinators that are created after evaluating the code in Listing 6.1. In the case of PetitParser, this composition is created *"ahead-of-time"* before a parse attempt takes place and can be reused for multiple parse attempts.



```
ClassNode new
  id: idToken value
  body: body
```

Figure 6.1: The structure of parser combinators created after evaluating the code of Listing 6.1.

The root `program` rule is translated into a `Plus` parser referencing the `class` combinator. The `class` rule is an `Action` parser — a parser that evaluates a block — specified by a `map:` parameter, the arguments being collected from the result of an underlying `Sequence` parser. The `idToken` and `classToken` rules are `Token` parsers, which consume whitespace and create token instances. The `'class'` rule is a `Literal` parser, which is a leaf combinator and does not refer to any other combinators. The `classToken` rule contains a sequence of `Literal` and `AndPredicate` combinators (omitted in the Figure 6.1). The `identifier` rule is a sequence of `CharClass` and `Star` parsers. The `Choice` parser from the `letterOrDigit` rule shares the `CharClass` parser with its `identifier` grand-parent. The `body` rule is a sequence of `indent`, `Star` and `dedent`, where `indent` and `dedent` are Python-like `indent` and `dedent` as we defined earlier in Listing 4.11. The `class` combinator is shared by `program` and `body`. The structure of `method` has been omitted.

Though straightforward and easy to understand, such a composition of parser combinators suffers from several performance issues as discussed in the following sections.

6.1.1 Composition Overhead

Composition overhead is caused by calling complex objects (parsers) even for simple operations.

For example, consider the grammar shown in Listing 6.1 and `letterOrDigit*` in `identifier`. This can be implemented as a simple loop, but in reality many methods are called. For each character, the following parsers are invoked: A `Star` parser (see Listing C.7 in Appendix C), a `Choice` parser (see Listing C.4 in Appendix C) and two `CharClass` parsers (see Listing C.3 in Appendix C). Each of these parsers contains at least five lines of code, averaging twenty lines of code per recognized character.

The same situation can be observed when parsing `&#space`. `AndPredicate` (see Listing C.2) and `CharClass` (see Listing C.3) are invoked during parsing. The `and` predicate creates a memento and the `char` operator moves in the stream just to be moved back by the `and` predicate in the next step. Ideally, the result can be determined with a single comparison of a stream peek.

6.1.2 Superfluous Intermediate Objects

Superfluous intermediate objects are allocated when an intermediate object is created but not used. A terminal combinator typically returns a literal (*e.g.*, character) and non-terminal combinator returns a composition of its children (*e.g.*, in collection). Together, they build a concrete syntax tree (CST),[2] which is later transformed into a more appropriate representation. Yet in some cases, this CST is never used or its usage is needlessly complicated.

For example, consider input `'Petit'` parsed by `idToken`. The return value of `idToken` is a `Token` object containing `'Petit'` as a value and `1` and `7` as start and end positions. Yet before a `Token` is created, a `Star` parser (see Listing C.7) and a `Sequence` parser (see Listing C.6) create intermediate collections resulting in `(P,(e,t,i,t))` nested arrays that are later flattened into `'Petit'` in `TokenParser` (see Listing C.8) again.

[2] http://www.webcitation.org/6k64scfyi

Another example is the action block in the `class`. The `classToken` creates an instance of `Token`. Yet the token is never used and its instantiation is interesting only for a garbage collector. Moreover, the `Sequence` parser wraps the results into a collection and the `Action` parser unwraps the elements from the collection in the very next step in order to pass them to the action block as arguments (see Listing C.1).

Furthermore some mementos in sequences are never used because an underlying parser never fails (it is nullable as explained in Definition A.7). For example, `Sequence` in `idToken` creates a memento that is never used because the second part of the `identifier` sequence, *i.e.*, `letterOrDigit*`, is nullable.

6.1.3 Backtracking Overhead

Backtracking overhead arises when a parser enters a choice alternative that is predestined (based on the next k tokens) to fail. Before failure, intermediate structures, mementos and `Failure` instances are created and time is wasted.

Consider input `'123'` and the `idToken` rule, which starts only with a letter. Before failure, the following parsers are invoked: `TokenParser` (see Listing C.8), `Sequence` (see Listing C.6), `CharClass` (see Listing C.3). Furthermore, as `TokenParser` tries to consume a whitespace (*e.g.*, using `#space*`), another `Star` (see Listing C.7) and `CharClass` (see Listing C.3) are invoked. During the process, two mementos are created. These mementos are used to restore a context even though nothing has changed, because parsing has failed on the very first character. Last but not least, a `Failure` instance is created.

As a different example, consider `letterOrDigit` or the more complex choice variant `class/method` that can be (for clarity reasons) expanded to:

```
(('class' token) idToken body) /
(('def' token) idToken body)
```

The choice always invokes the first alternative and underlying combinators before the second alternative. In some cases, *e.g.*, for input `'def bark ...'`, based on the first character of input, it is valid to invoke the second alternative without entering the first one. In other cases, *e.g.*, for input `'package Animals ...'`, it is valid to fail the whole choice without invoking any of the parsers.

Yet the choice parser invokes both alternatives creating superfluous intermediate collections, mementos and failures before it actually fails.

6.1.4 Context-Sensitivity Overhead

In the case of PetitParser, the context-sensitivity overhead appears when a context contains non-empty stacks (*e.g.*, an indentation stack, an invocation stack, *etc.*). When memoizing a parser, deep copies of stacks are created (see Listing 4.8).

This is the right approach in some cases, *e.g.*, the `body` sequence where `indent` modifies the indentation stack (see for example Listing 4.11). If any of the subsequent rules in `body` fails, the original stack has to be restored. However, in other cases, *e.g.*, the `identifier` sequence where none of the underlying combinators modifies the indentation stack, the deep copy of a context is superfluous.

6.2 A Parser Combinator Compiler

The goal of a parser compiler is to provide a high-performance parser from a parser combinator while preserving all the advantages of parser combinators. To achieve its goal, a parser compiler analyzes the given PEG grammar, selects the most appropriate parsing strategy for each of its rules and creates a top-down parser where each method represents a rule of the grammar with the selected strategy implemented in the method body.

Figure 6.2 shows the work-flow of parser development with the parser compiler we present. First, flexibility and comprehensibility of combinators is utilized (*prototype phase*). Then, a parser compiler applies domain-specific optimizations, builds a top-down parser (*compile phase)* and allows an expert to further modify the compiled parser at her will[3] to further improve the performance (*hand-tune phase)*. In the end, the resulting parser can be deployed as an ordinary class and used for parsing with peak performance (*deployment phase)*.

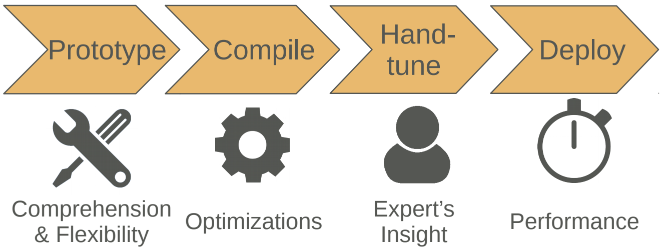


Figure 6.2: In the prototype phase a language engineer utilizes the advantages of parser combinators that are later compiled to a high performance parser, possibly tuned by the engineer and deployed.

The idea of a parser compiler fits well into agile modeling and addresses the performance problems of a parser for agile modeling. The parser compiler does not restrict grammars it can optimize (in the worst case it simply does not optimize). It works automatically without user interaction and the output can be further customized if needed.

6.2.1 Adaptable Strategies

As mentioned in section 6.1 there are four different kinds of overhead: (i) composition overhead; (ii) superfluous intermediate objects; (iii) backtracking overhead; and (iv) context-sensitivity overhead. The different parsing strategies target these four kinds of overhead and reduce it.

The parser compiler can work in two different modes: a scannerless mode and a scanning mode. The scanning mode provides better performance but cannot be used for all the grammars,[4] therefore the scannerless mode serves as a fallback option. The parsing strategies recognized and utilized by a parser compiler in a scannerless mode are visualized in Figure 6.3 and the strategies utilized in a scanning mode are in Figure 6.4.

[3] A parser compiler recognizes a hand-tuned code and does not override it unless explicitly stated.

[4] As we explain in Appendix E.

Different languages are parsed by different strategies. For example, some combinators describing regular languages can be optimized by specializations, parsing expressions and recognizers in the scannerless mode and by regular parsing expressions in the scanning mode. The recognizers strategy of a scannerless mode can be combined with both; specializations and parsing expressions.

The combinators describing context-free languages are optimized by LL(1) choices, nondeterministic choices and, again, by parsing expressions. Furthermore, combinators can be optimized to use only context-free memoizations, thus there are also context-free combinators.

There is not much space to optimize combinators describing context-sensitive languages; some of them can be optimized using context-sensitive parsing expressions.

A parser compiler in a scanning mode utilizes a scanner (see Appendix E) to parse regular parsing expressions (see section E.2) and to guide choices to choose the correct alternative based on the next token. In case a scanner cannot be used a parser compiler falls back to a scannerless mode with character-based alternatives. A short description of all the strategies applied by a parser compiler follows:

Combinators serve as a fallback option. Whenever a parser compiler does not identify a suitable strategy, the original combinator is used. This way it is ensured that the parser compiler works even for unknown combinators.

Context-Sensitive Parsing Expressions (CS-PEs) reduce composition overhead, because they perform loop unrolling[5] of choices (see the implementation in Listing C.4) and sequences (see the implementation in Listing C.6).

Parsing Expressions (PEs) reduce composition overhead of context-sensitive combinators and context-sensitivity overhead. They reduce composition overhead because they perform loop unrolling. They reduce context-sensitivity overhead because they use only context-free mementos instead of more complex context-sensitive ones.

Nondeterministic Choices reduce backtracking overhead, because they reject an alternative based on the next character (or token) of input.[6]

Deterministic Choices reduce backtracking overhead, because they choose the correct alternative based on the next character (or token) of input.

Regular Parsing Expressions (RPEs) reduce composition overhead by replacing a hierarchy of combinators by a finite state automaton, which can be implemented more efficiently than a recursive top-down parser.

Specializations reduce composition overhead by replacing a hierarchy of combinators by a simple programming construct such as a loop or a comparison, which are more efficient than interactions of combinators.

Recognizers reduce superfluous intermediate allocations because they avoid intermediate representations. Recognizers return only `true` or `false` instead of CST. This improves performance since object initialization methods do not need to be run and because it improves the efficiency of a typical Smalltalk garbage collector [Wil92].

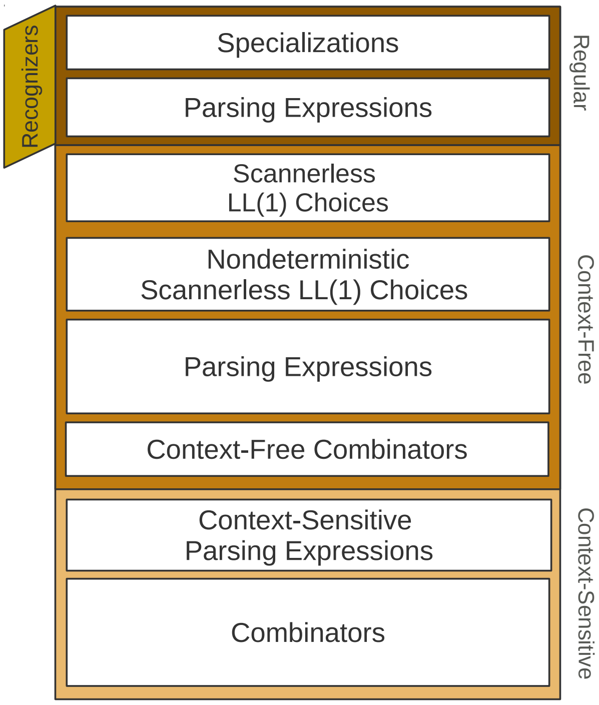


Figure 6.3: Parsing strategies applied in a scannerless mode. The recognizer strategy can be combined with both, specializations and recognizers.

While Figure 6.3 and Figure 6.4 serve as an overview, they are neither precise nor exhaustive. For example, scannerless choices can exist in a context-free and a context-sensitive variant. In reality the borders between regular, context-free and context-sensitive parsing expressions are not as clear as visualized. We avoid such details to improve comprehensibility of the overview. In the following section we describe in detail when and how the parser compiler applies these strategies and how the final code looks like.

6.3 Parser Optimizations

Parser combinators form a graph with cycles. A parser compiler uses this graph of combinators as its intermediate PEG-aware representation. The optimizations themselves are implemented as a series of passes over the graph, each performing a transformation using pattern matching. Particular nodes are moved, replaced with more appropriate alternatives, or changed and extended with additional information. In the final phase, these nodes are visited by a code generator to implement the selected strategy in the

[5] `http://www.webcitation.org/6k65AygCX`

[6] They are not deterministic because even if an alternative is rejected, it is not clear which of the remaining alternatives should be selected.

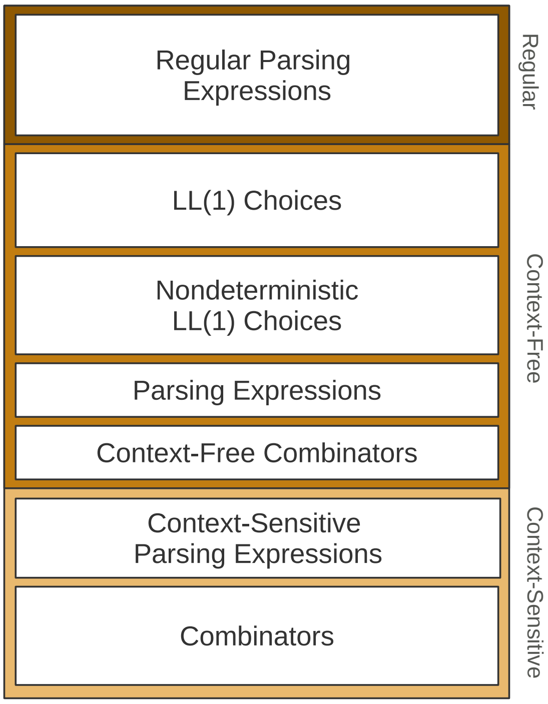


Figure 6.4: Parsing strategies applied in a scanning mode.

host code. Code generation results in a class where each method represents a combinator in the modified graph of parser combinators.

Contrary to other representations used for performance optimizations (*e.g.*, AST,[7] Bytecode,[8] SSA[9] or LLVM IR[10]) the intermediate representation of a parser compiler is high-level, directly represents the target domain and therefore allows for domain-specific optimizations. For example, a parser compiler can directly check if an expression is nullable or a choice is deterministic. This would be close to impossible with low-level representations.

The following section describes the optimizations in detail. We use the following syntax for rewriting rules. The specific class of combinator is in angle brackets `<>`, *e.g.*, all the character class combinators are marked as `<CharClass>`. Any parser combinator is `<Any>`. A parser combinator that is a child of a parent `P` is marked `P→*<Any>`. We use this syntax to refer to all the successors of a given parent `P`. For example, `program →*<Any>` refers to all the parsing expressions in the grammar from Listing 6.1.

A parser combinator with a property is marked with `:` and the property name, *e.g.*,

[7] http://www.webcitation.org/6k64zYXWe
[8] http://www.webcitation.org/6k650GihL
[9] http://www.webcitation.org/6k651b8Pg
[10] http://www.webcitation.org/6k652NVlc

`<Any:nullable>`. Adding a property to a parser combinator is marked with `+` sign and the property name, for example `<Any+nullable>` adds a `nullable` property.

Delegating parsers embed the parser which they delegate to in angle brackets, *e.g.*, `<Sequence<Any><Any>>` represents a sequence of two arbitrary combinators. An alternative syntax for sequences and choices and other delegating operators is to re-use the PEG syntax, *e.g.*, `<Any> <Any>` is also a sequence of two arbitrary combinators. The rewrite operation is ⇒. Merging a choice of two character classes into a single one is written as:

```
  <CharClass> / <CharClass> ⇒ <CharClass>
```

6.3.1 Regular Optimizations

Intuitively, regular optimizations are performed on a token level, *i.e.*, on expressions recognizing identifiers, numbers or keywords. Some of these expressions can be expressed by finite state automata, but since PEGs are scannerless and have different semantics than regular expressions this is not always possible.

Regular Parsing Expressions

Regular parsing expressions (see section E.2) are expressions recognizable by finite state automata (FSAs). FSAs can be implemented more efficiently than parsing expressions, without backtracking, composition overhead, or superfluous object allocations.

During a dedicated optimization phase, all expressions are analyzed for their *"regularity"* and marked as regular `+regular`. All the regular expressions are wrapped with a `<Scanner>` combinator.

```
<Any:regular> ⇒ <Scanner<Any>>
```

A `<Scanner>` combinator represents a traditional scanner that uses an implementation of an FSA to parse input. For example, `identifier` is a regular parsing expression and can be parsed with `<Scanner>`:

```
identifier ⇒ <Scanner<#letter <#letter / #digit>*>
```

In the final phase, the code generator produces the following code from the `identifier` sequence:

```
scanner scan_identifier
```

Details about the scanner and its integration into a parser are provided in Appendix E.

Specializations

Specializations reduce composition overhead by replacing a hierarchy of combinators by a simple programming construct such as a loop or a comparison.

Returning to the problem with `letterOrDigit*` in subsection 6.1.1 (let us suppose that a scanner is not utilized), the whole rule is specialized as an instance of the `<CharClassStar>` combinator. The `#digit / #letter` rule is specialized using a single `CharClass` combinator `[a-zA-Z0-9]`, and a repetition of the character class is replaced by a specialized `CharClassStar` combinator, which can be implemented as a while loop. The `letterOrDigit∗` rule is rewritten to the following:

```
letterOrDigit⋆ ⇒ <CharClassStar[a-zA-Z0-9]>
```

In the final phase, the code generator produces the following code, which contains only three lines of code per consumed character:

```
letterOrDigitStar
    | retval |
    retval ← OrderedCollection new.
    [context peek isLetter or:
    [context peek isDigit]] whileTrue: [
        retval add: context next.
    ].
    ↑ retval
```

Returning to the problem with `&#space` in subsection 6.1.1, the whole rule is specialized as a single `AndCharClass` combinator. The `classToken` rule is rewritten as follows:

```
classToken ← 'class' <AndCharClass[\t\n␣]
```

In the final phase, the code generator produces for `AndCharClass` the following code, which does not create any mementos and does not invoke any extra methods:

```
↑ (context peek isSpace) ifFalse: [
  Failure message: 'space expected'.
]
```

We implement several similar specializations, including the following:

- A new `<CharClass>` is created from a choice of char classes:

  ```
  <CharClass> / <CharClass> ⇒ <CharClass>
  ```

- A new `CharClass` is created from the negation of a char class:

  ```
  <CharClass> negate ⇒ <CharClass>
  ```

- `CharClassStar` is created from a star repetition of a char class (as we show in the example):

  ```
  <CharClass>⋆ ⇒ <CharClassStar>
  ```

- `CharClassPlus` is created from a plus repetition of a char class:

  ```
  <CharClass>+ ⇒ <CharClassPlus>
  ```

- `AndCharClass` (or `NotCharClass`) is created from char class predicates:

```
&<CharClass>  ⇒ <AndCharClass>
!<CharClass>  ⇒ <NotCharClass>
```

- `AndLiteral` (or `NotLiteral`) is created from literal predicates:

```
&<Literal>  ⇒ <AndLiteral>
!<Literal>  ⇒ <NotLiteral>
```

- `TokenCharClass` is created from a single-character token:

```
<CharClass> token  ⇒ <TokenCharClass>
```

Recognizers

Recognizers target the problem of superfluous object allocations. Combinators forming a `Token` parser are marked to avoid generating intermediate representations (`+recognizer`) because tokens do not have an internal structure (as described in section 3.2). The same happens in the case of and `&` and not `!` predicates, which return only `true` or `false` and discard their result. Recall that `P →*<Any>` refers to all the children of `P` .

```
<Token>→*<Any>          ⇒ <Token>→*<Any+recognizer>
<AndPredicate>→*<Any>  ⇒ <AndPredicate>→*<Any+recognizer>
<NotPredicate>→*<Any>  ⇒ <NotPredicate>→*<Any+recognizer>
```

As an example, the `CharClassStar` parser specialized from `letterOrDigit*` inside the `idToken` is marked to avoid generating an intermediate representation:

```
letterOrDigit⋆ ← <CharClassStar[a-zA-Z0-9]:recognizer>
```

In the final phase, the code generator produces the following code:

```
letterOrDigitStar
    [context peek isLetter or:
    [context peek isDigit]] whileTrue: [
        context next.
    ].
```

6.3.2 Context-Free Optimizations

Optimizations on this level focus primarily on lookahead and backtracking. PEG choices are analyzed and backtracking reduced, if possible.

Deterministic choices

Deterministic choices limit invocations and allocations caused by backtracking. During a dedicated optimization phase, character-based or token-based *first sets* [Red09, GJ08b] are computed. If all n choice alternatives $a_1/a_2/.../a_n$ have distinct first set (*i.e.*, their first sets do not overlap) a choice is marked as a deterministic choice (+dch) and choices with dch property are replaced with a deterministic choice combinator.

```
<Choice <Any> <Any>:dch> ⇒ <DeterministicChoice <Any> <Any>>
```

For example the class/method choice is marked as dch and the whole body is rewritten to:

```
body ⇒ indent
            <DeterministicChoice <class> <method>>*
        dedent
```

In the final phase, the code generator produces the following code from this choice:

```
classOrMethod
    | result |
    (context peek == $c) ifTrue: [ ↑ self class ].
    (context peek == $d) ifTrue: [ ↑ self method ].
```

Or if a scanner is utilized:

```
classOrMethod
    | token |
    token ← scanner scan_classOrMethod.
    (token == #class) ifTrue: [ ↑ self class ].
    (token == #def    ifTrue: [ ↑ self method ].
```

Nondeterministic choices

Nondeterministic choices partially prevent invocations and allocations caused by backtracking. In case alternatives of a choice overlap and the choice is not deterministic, it still might be optimized using guards. Guards allow for an early failure of a parse attempt using the peek character or the next token. When suitable (*e.g.*, the character-based first set is reasonably small) choice alternatives are marked for a guard (+guard). Any choice alternative marked for guarding <Any:guard> is wrapped with Guard. Some alternatives do not need to be guarded:

```
<Any:guard> / <Any:guard> ⇒ <Guard<Any>> / <Guard<Any>>
<Any:guard> / <Any>       ⇒ <Guard<Any>> / <Any>
<Any> / <Any:guard>       ⇒ <Any> / <Guard<Any>>
```

Guard is a combinator that prepends an underlying combinator with a code that fails immediately, without entering the underlying combinator. As an example, let us slightly modify class and method from Listing 6.1 to allow for private definitions:

```
method       ← privateToken? defToken idToken ...
class        ← privateToken? classToken  idToken ...

privateToken ← 'private' token
```

In such a case, the alternatives of a class/method are wrapped with Guard.

```
body ← indent
         <Choice <Guard<class>> <Guard<method>)*
       dedent
```

In the final phase, the code generator produces the following code from the choice of `class` and `method`:

```
classOrMethod
    | result |
    (context peek == $c or: [ context peek == $p]) ifTrue: [
        (result ← self class) isFailure ifFalse: [
            ↑ result
        ]
    ].
    (context peek == $d or: [ context peek == $p]) ifTrue: [
        (result ← self method) isFailure ifFalse: [
            ↑ result
        ]
    ].
    ↑ Failure message: 'no suitable alternative found'
```

Or if a scanner is utilized:

```
classOrMethod
    | result memento |
    memento ← scanner remember
    (scanner guard_privateOrClass) ifTrue: [
        (result ← self class) isFailure ifFalse: [
            ↑ result
        ]
        scanner restore: memento.
    ].
    (scanner guard_privateOrMethod) ifTrue: [
        (result ← self method) isFailure ifFalse: [
            ↑ result
        ]
        scanner restore: memento.
    ].
    ↑ Failure message: 'no suitable alternative found'
```

6.3.3 Context-Sensitive Optimizations

Optimization on this level addresses performance problems of parser combinators that are too generic, and superfluous context-sensitive memoizations.

Context-Sensitive Parsing Expressions

In a combinator implementation the children of sequences and choices are called in a loop (see Listing C.6 and Listing C.4). The parser compiler improves performance of sequences and choices by loop unrolling.

The choices are simply unrolled; no further analysis is performed. However, the children of sequences are analyzed for the nullability property (see Definition A.7) and marked as nullable (`+nullable`) if so. Nullable expressions never fail (see Definition A.6) and error handling can be omitted. The restore after the first element of a sequence is also omitted. It is superfluous, because in case of failure the underlying code will have restored the context already (see the `parseOn:` contract in section 3.2). We replace the sequences with their nullable variants:

```
<Sequence <Any> <Any:nullable>>
    ⇒ <SecondNullableSequence <Any> <Any>>
```

There is no `<FirstNullableSequence>` because a restore after the first element of a sequence is superfluous. In practice, the sequences can have more children. The rewrite rules for sequence with three children extend straightforwardly:

```
<Sequence <Any> <Any:nullable> <Any>>
    ⇒ <SecondNullableSequence <Any> <Any>>
<Sequence <Any> <Any> <Any:nullable>>
    ⇒ <ThirdNullableSequence <Any> <Any>>
<Sequence <Any> <Any:nullable> <Any:nullable>>
    ⇒ <SecondAndThirdNullableSequence <Any> <Any>>
```

For example, the repetition in the `body` rule is marked as nullable:

```
body ⇒ indent
             <Star <class/method>:nullable>
       dedent
```

and the `body` rule is rewritten as follows:

```
body ⇒ <SecondNullableSequence <indent>
             <Star <class/method>:nullable>
       <dedent>>
```

However, in practice when sequences have any number of children, we do not rewrite sequences. Instead the parser compiler checks the `nullable` property of each child and omits the restore code if the `nullable` property is set. In the final phase, the code generator produces the code in Listing 6.2.

Context-Free Memoization

Context-free memoization reduces the overhead of context-sensitive expressions by turning them into the context-free expressions. The context-free expressions use only a position in a stream as a memento. The deep copy of a context is performed only when necessary, *i.e.*, for the context-sensitive parts of a grammar.

We describe two approaches. The first one is more universal and can be applied to a context-sensitive grammar, *e.g.*, a grammar utilizing grammar rewriting. The other one is tailored to our context-sensitive extension utilizing the push `▽` and `△` operators and is based on the *push–pop* analysis (see Definition 4.8).

The *context-sensitive* analysis traverses the combinators and marks a combinator as context-sensitive (`+cs`) whenever a combinator performs context-sensitive operations. This might be, for example, a combinator depending on an external context or performing some sort of a grammar modification. If a combinator refers to a context-sensitive combinator, the combinator is marked as context-sensitive as well:

```
body
    | memento indent dedent classOrMethodCollection |
    memento ← context remember.

    [ indent ← self indent ] isFailure ifTrue: [
        "no context restore needed here"
        ↑ indent
    ]

    "classOrMethodStar is nullable, no error handling needed"
    classAndMethodCollection ← self classOrMethodStar.

    [ dedent ← self dedent ] isFailure ifTrue: [
        context restore: memento.
        ↑ dedent
    ]

    ↑ Array with: indent
            with: classAndMethodCollection
            with: dedent
```

Listing 6.2: The code produced from the `body` sequence after applying the context-sensitive parsing expressions optimization.

```
<Any>→*<Any:cs> ⇒ <Any+cs>→*<Any:cs>
```

The *push–pop* analysis also traverses the combinators, but it performs the *push–pop* analysis and marks each combinator as push `+push`, pop `+pop`, context-sensitive `+cs` or context-free `+cf`.

With *push–pop* analysis more expressions are marked as context-free, because consecutive `▽` and `△` result in a context-free expression (in a sense the expression does not change the context). Furthermore, a sequence can restore after push `▽` by calling pop `△`. Only expressions in sequences after the pop must be restored using the full context-sensitive memento.For example, consider `INDENT` and `DEDENT` from Listing 4.11. After `INDENT`, which pushes to the indentation stack, the indentation stack can be restored to the state before `INDENT` by popping the top of the indentation stack. On the other hand, after `DEDENT`, which pops an element from the indentation stack, the popped element cannot be be restored by push, because we don't know what to push. The solution we use is to restore from the full context-sensitive memento.

As an example of a *push–pop* analysis, consider the `body` sequence, which is marked as context-free by *push–pop* analysis:

```
body    ← <<indent:push>
             <Star<classOrMethod>:cf>
         <dedent:pop>:cf>
```

Even though the `body` sequence contains the context-sensitive rules `indent` and `dedent`, no context-sensitive memento is used in the generated code:

```
body
    | memento indent dedent classAndMethodCollection |
    memento ← context position.

    [ indent ← self indent ] isFailure ifTrue: [
        "no context restore needed here, indent did it"
        ↑ indent
    ]

    "classOrMethodStar is nullable, no error handling needed"
    classAndMethodCollection ← self classOrMethodStar.

    ( dedent ← self dedent ) isFailure ifTrue: [
        context indentationStack pop.
        context position: memento.
        ↑ dedent
    ]

    ↑ Array with: indent
            with: classAndMethodCollection
            with: dedent
```

Listing 6.3: The code produced from the `body` sequence after applying the context-free memoizaton optimization.

For this optimization, we don't use any rewrite rules, because the context-free memoization can be combined with almost any combinator and we would have to introduce a context-free and context-sensitive variants for each of the combinators. This is a typical use case for the strategy pattern [Gam97], which we use to implement the context-free memoizations. We use two memoization strategies (context-free and context-sensitive) that are assigned to each of the combinators based on the result of a *push–pop* analysis. The memoization strategies are used by a parser compiler to generate appropriate remember and restore code.

Combinators

Combinators are the last resort if no other parsing strategy can be applied. This can happen, for example, if a parser compiler optimizes a bounded sea combinator (see Listing C.11) whose semantics is unknown to it.

The semantics of the unknown combinator is performed by the combinator itself. Because many of combinators delegate to their children, the parser compiler optimizes children of the unknown combinator and replaces them with a dedicated `Bridge` combinator. The `Bridge` combinator is a standard parser combinator that forwards the parsing logic to the compiled and optimized code. This way some parts of a parser are executed using the compiled and optimized code while the unknown parts remain unchanged.

If `<Unknown>` represents a combinator unknown to a parser compiler, the rewrite rule is

```
<Unknown <Any>> ⇒ <Unknown <Bridge<Any>>>
```

As an example, let us suppose we define `body` as a bounded sea of classes or methods:

```
body ← ∼(class/method)⋆∼
```

This definition of `body` is transformed to the following combinator graph (the rule `class/method` is compiled as usually):

```
body ← <BoundedSea<Bridge<(class/method)⋆>>>
```

In the final phase, the code generator produces the following code, where `unknown` is an instance of a `BoundedSea` parser:

```
body
    ↑ unknown parseOn: context
```

Inside the `unknown>>parseOn:` an island is called (see Listing C.11). The island is replaced by a `Bridge` combinator that forwards to the compiled code:

```
Bridge>>parseOn: context
    "in our example, the selector is 'classOrMethodStar'"
    ↑ compiledParser perform: selector
```

6.4 Performance analysis

In this section we briefly introduce the *PetitParser compiler*, our implementation of a parser compiler for PetitParser. We report on performance of parsers compiled by the PetitParser compiler compared to the performance of plain PetitParser. We also report on impact of a particular parsing strategy on the overall performance, we analyze performance of bounded seas and last but not least we compare several different implementations of a Smalltalk parser with that compiled by the PetitParser compiler.

6.4.1 PetitParser compiler

PetitParser compiler is an implementation of a parser compiler for PetitParser. The PetitParser compiler applies the parser compiler techniques and outputs a class that serves as a top-down parser equivalent to the input combinator.

The PetitParser compiler is available online[11] for Pharo and Smalltalk/X.[12] It is being used in real environments, for example a language for Live Robot Programming[13] and the Pillar markup language.[14]

[11] `http://www.webcitation.org/6k65PNNfm`

[12] `http://www.webcitation.org/6k62sGRlg`

[13] `http://www.webcitation.org/6k65Q8jg9`

[14] `http://smalltalkhub.com/#!/~Pier/Pillar`

Validation. The PetitParser compiler is covered by three thousand tests. Furthermore, it is validated against several existing PetitParser combinators and comparing their results with the results produced by the compiled variant of a particular parser. In particular, the results are validated for:

- Java parser[15] on OpenJDK 6 source code,[16]
- Smalltalk parser[17] on Pharo source code,[18]
- Ruby and Python semi-parsers[19] on several GitHub projects: Cucumber, Diaspora, Discourse, Rails, Vagrant, Django, Reddit and Tornado.

6.4.2 Benchmarks

We focus on performance of the following grammars, most of which we have already analyzed in the previous chapters:

Expressions is a benchmark measuring performance of parsing arithmetic expressions used in the performance analysis of parsing contexts (see section 4.4). Input consists of expressions with operators `(`, `)`, `*`, `+` and integer numbers. The brackets must be balanced. The operator priorities are considered. The grammar is in a deterministic form. The parser contains eight rules.

Indentation-Sensitive Expressions (IS Expressions) is an indentation-sensitive version of the previous used in the performance analysis of parsing contexts (see section 4.4). Input consists of expressions with operators `*`, `+`, integer numbers and `indent`, `dedent` instead of brackets. The parser utilizes parsing contexts and the offside line rule as described in chapter 4. For example, `'2*(3+4)'` is represented as follows:

```
2*
  3+4
```

Context-Free Python (CF Python) is a benchmark measuring performance of the Python parser used in the Python case study (see section 4.5). Input consists of several GitHub Python projects[20] that are preprocessed and explicit `indent` and `dedent` tokens are inserted. The parser contains approximately forty rules. The parser utilizes islands [Moo01] but not bounded seas — it extracts structural elements and skips the rest.

Python is a benchmark measuring performance of an indentation-sensitive Python parser used in the Python case study (see section 4.5). It does not require preprocessing; the parser determines `indent` and `dedent` on its own. The parser utilizes islands [Moo01], parsing contexts and the offside line rule.

[15] http://smalltalkhub.com/#!/Moose/PetitJava/
[16] http://download.java.net/openjdk/jdk6
[17] http://smalltalkhub.com/#!/~Moose/PetitParser
[18] http://files.pharo.org/get-files/50/sources.zip
[19] http://smalltalkhub.com/#!/~JanKurs/PetitParser
[20] Django, Tornado and Reddit.

Smalltalk is a benchmark measuring performance of a Smalltalk parser. Input consists of a source code from a Pharo 5 image.[21] The parser contains approximately eighty rules.

Java is a benchmark measuring performance of a Java parser provided by the Moose analysis platform community [NDG05] used in the bounded seas Java case-study (see section 5.5). Input consists of standard JDK library files. The parser contains approximately two hundred rules.

Java Sea is a benchmark measuring performance of a Java sea parser utilizing bounded seas used in the bounded seas Java case-study (see section 5.5). The parser contains approximately twenty rules.

The presented benchmarks cover a variety of grammars varying from small ones to complex ones with size from eight to two hundred rules. Some grammars are deterministic (arithmetic expressions) and some utilize longer lookahead (Smalltalk and Java).[22] They also cover standard grammars (Expressions, Smalltalk and Java), context-sensitive grammars (IS Expressions and Python), island grammars (CF Python, Python) and a bounded sea grammar (Java Sea).

How we measure. We run each benchmark ten times using the latest release of the Pharo VM for Linux.[23] All the parsers and inputs are initialized in advance, then we measure time to parse.

We report on speedup (the ratio between original PetitParser serving as a baseline and its compiled version) and time per character. When measuring speedup, we consider the best time. To estimate impact of a garbage collector, we collect both times, with and without the garbage collection. When showing time per character, we visualize five-number summary, median is represented by a bar, lowest value, first, third quartiles and highest value are represented by a box with whiskers. In this chapter we show only the most important graphs; all the remaining graphs with more details can be found in Appendix F.

Results. The speedup of a compiled version compared to the original version is in Figure 6.5 (its zoomed-in version is in Figure F.2). The speedup ranges from two to twenty-five. Most of the parsers have speedup in a range from two to five. The exceptions are the Python parser and the Java Sea parser, for which we measure twenty-five times and ten times better performance.

Surprisingly, the speedup of IS Expressions is significantly lower compared to the speedup of other-context-sensitive parsers, *i.e.*, Python and Java Sea. This is caused by a significantly larger number of context-sensitive mementos created in Python and Java Sea parsers.

In terms of time per character as visualized in Figure 6.6 (its zoomed-in version is in Figure F.4) all the compiled parsers are in a range from 0.3 to $1.1 \mu s$ per character. The Python parser is performing almost as well as its context-free variant and the performance of the Java Sea parser is close to the performance of the Java parser. In case of IS Expressions, the difference is still rather high. This is caused by the simplicity of

[21] `http://files.pharo.org/get-files/50/sources.zip`

[22] The implementors did not bother to make it deterministic.

[23] from October 7, 2016

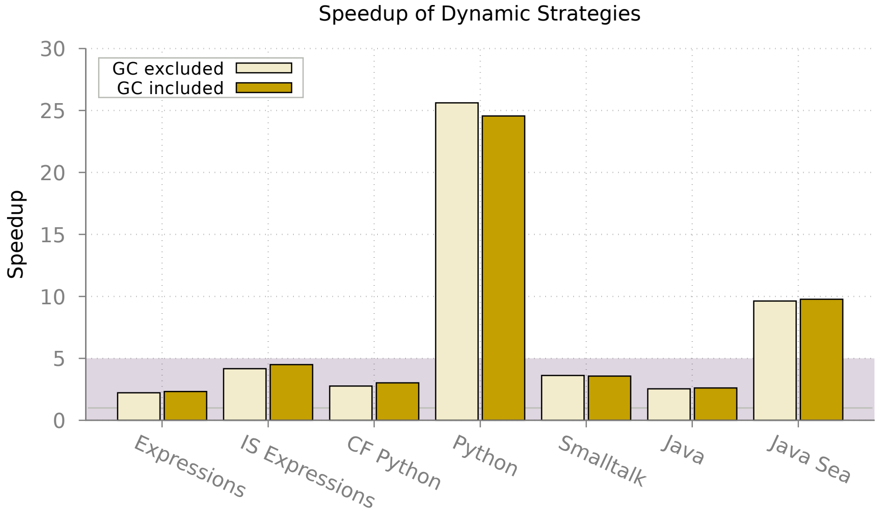


Figure 6.5: The speedup of compilation for different grammars.

the Expressions grammar where the overhead caused by indentation (even though optimized) is still significant because indentation context-sensitive rules consume a large part of input.

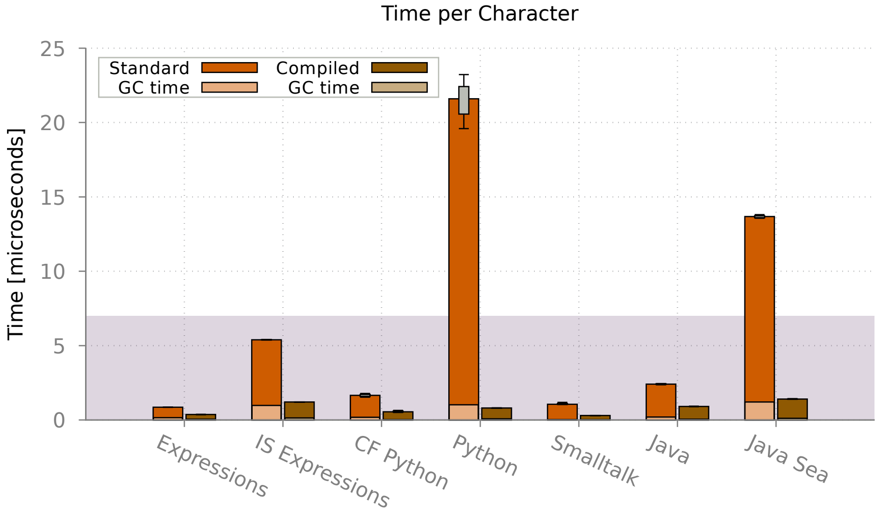


Figure 6.6: The time-per character for different grammars.

6.4.3 Parsing Strategies Impact

We report on speedup (ratio between original PetitParser and the compiled version) and time per character. When compiling we enable different parsing strategies and we report on these. Because of the way the PetitParser compiler is implemented, some parsing strategies cannot be applied without transforming the parser into the context-sensitive parsing expressions (PE). Therefore the context-sensitive parsing expressions are always included. We use the following configurations:

PE transforms combinators of PetitParser to context-sensitive parsing expressions performing loop unrolling as described in section 6.3.3.

PE+RE applies all the transformations from subsection 6.3.1 to the context-sensitive parsing expressions (PE). If possible a scanner is utilized, otherwise specializations and recognizer strategies are utilized. The context-sensitive mementos are used, choices are not optimized.

PE+CF applies all the transformations from subsection 6.3.2 to the parsing expressions (PE). Only character-based deterministic choices and guards are utilized because regular parsing expressions are not detected and a scanner cannot be used to optimize choices. The context-sensitive mementos are used.

PE+CS applies all the transformations from subsection 6.3.3 to the parsing expressions (PE). The parsing expressions are analyzed with the *push–pop* analysis and a context-free memoization is used whenever possible. Regular expressions and choices are not optimized.

PE+RE+CF applies transformations from subsection 6.3.1 and subsection 6.3.2 to the parsing expressions (PE). If possible a scanner, token-based deterministic choices and guards are utilized. Otherwise specializations, recognizers, character-based deterministic choices and guards are utilized. The context-sensitive mementos are created.

PE+RE+CS applies transformations from subsection 6.3.1 and subsection 6.3.3 to the parsing expressions (PE). If possible a scanner is utilized, otherwise specializations and recognizers are utilized. The parsing expressions are analyzed with the *push–pop* analysis and a context-free memoization is used whenever possible. Choices are not optimized.

PE+CF+CS applies transformations from subsection 6.3.2 and subsection 6.3.3 to the parsing expressions (PE). Only character-based deterministic choices and guards are utilized because regular parsing expressions are not detected and a scanner cannot be used to optimize choices. The parsing expressions are analyzed with the *push–pop* analysis and a context-free memoization is used whenever possible. The regular expressions are not optimized.

All applies all the transformations. If possible a scanner and token-based deterministic choices and guards are utilized. Otherwise specializations, recognizers, character-based deterministic choices and guards are utilized. The parsing expressions are analyzed with the *push–pop* analysis and a context-free memoization is used whenever possible.

The speedup of a particular configuration is shown in Figure 6.7 (its zoomed-in version is in Appendix, Figure F.8). The baseline is the plain PetitParser version. The graphs illustrate how much a particular configuration contributes to the overall performance. Times per character and detailed graphs for each of the grammars are in Appendix F.2.

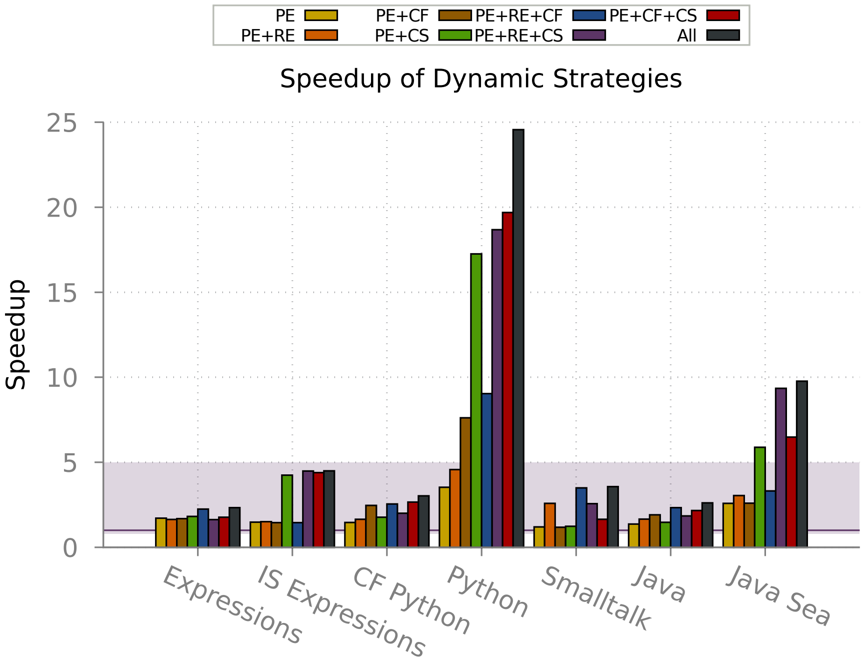


Figure 6.7: Speedup against plain PetitParser for different configurations.

Different strategies have different impact on each of the parsers. Regular expression strategies (PE+RE), for example, optimize the Smalltalk and Java Sea parsers well, on the other hand, they are worse than standalone parsing expressions (PE) in the case of Expressions. This is caused by a scanner that provides the best performance when combined with the context-free optimizations as we illustrate in subsection 6.4.4. Context-free strategies (PE+CF) affect the most the CF Python, Python and Java parsers. Context-sensitive strategies (PE+CS) significantly affect the context-sensitive parsers such as the IS Expressions, Python and Java Sea parsers. In other cases context-sensitive strategies slightly improve performance because mementos are integers and not objects.

6.4.4 Scanner Impact

The current implementation of a scanner limits its usage to the Expressions parser and the Smalltalk parser. Other grammars either do not use `token` to consume input (CF Python and Java Sea), are context-sensitive sensitive (IS Expressions and Python), or do not contain regular expressions in tokens (Java, which uses tokens composed of other tokens). The speedup using a scanning strategy is approximately 10-15% (see Figure 6.8). The baseline is a parser that does not utilize a scanning strategy.

In order to investigate in detail the impact of a scanner on Expressions and Smalltalk, we measure performance in the following configurations with and without a scanner:

PE transforms combinators of PetitParser to context-sensitive parsing expressions performing loop unrolling as described in section 6.3.3.

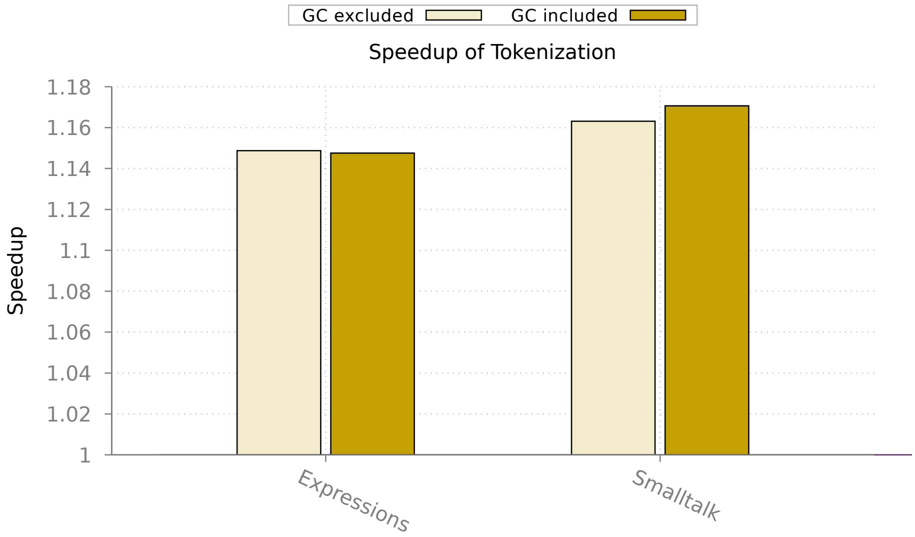


Figure 6.8: Speedup of a scanning strategy when applied on the Expressions and Smalltalk parsers.

PE+SRE applies the scannerless strategies from subsection 6.3.1 to the parsing expressions (PE). There is no token analysis, only specializations and recognizers are applied. Choices are not optimized.

PE+TRE applies the tokenizing (scanning) strategy from subsection 6.3.1 to the parsing expressions (PE). Expressions are analyzed for tokens and a scanner is utilized. The context-sensitive mementos are used, choices are not optimized.

PE+SRE+CF applies the scannerless strategies from subsection 6.3.1 and choice strategies subsection 6.3.2 to the parsing expressions (PE). There is no token analysis, only specializations and recognizers are applied. Scannerless character-based deterministic choices and guards are utilized.

PE+TRE+CF applies the tokenizing (scanning) strategy from subsection 6.3.1 and choice strategies subsection 6.3.2 to the parsing expressions (PE). Expressions are analyzed for tokens and a scanner is utilized, as well as the token-based deterministic choices and guards.

The results for Expressions are in Figure 6.9 and for Smalltalk in Figure 6.10. The baseline is plain PetitParser. In the case of the Expressions parser, a standalone tokenization (PE+TRE) provides worse performance than its scannerless variant (PE+SRE), but when combined with the context-free optimizations (PE+TRE+CF), the token strategy outperforms the scannerless variant (PE+SRE+CF). In the case of the Smalltalk parser, a standalone tokenization (PE+TRE) is better than its scannerless variant (PE+SRE). The probable reason being the higher complexity of tokens that are better optimized by the scanner than by the specializations and recognizers.

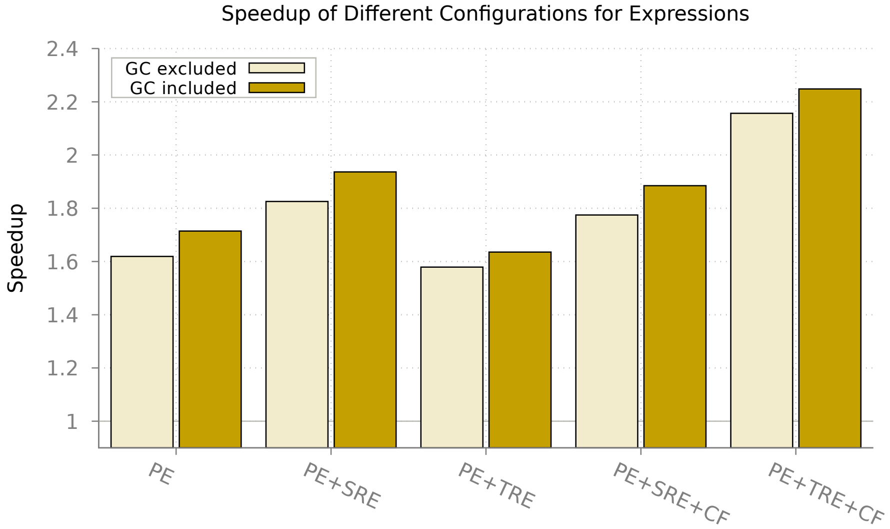


Figure 6.9: Speedup of Expressions against the plain PetitParser for different configurations with scannerless or tokenizing strategies.

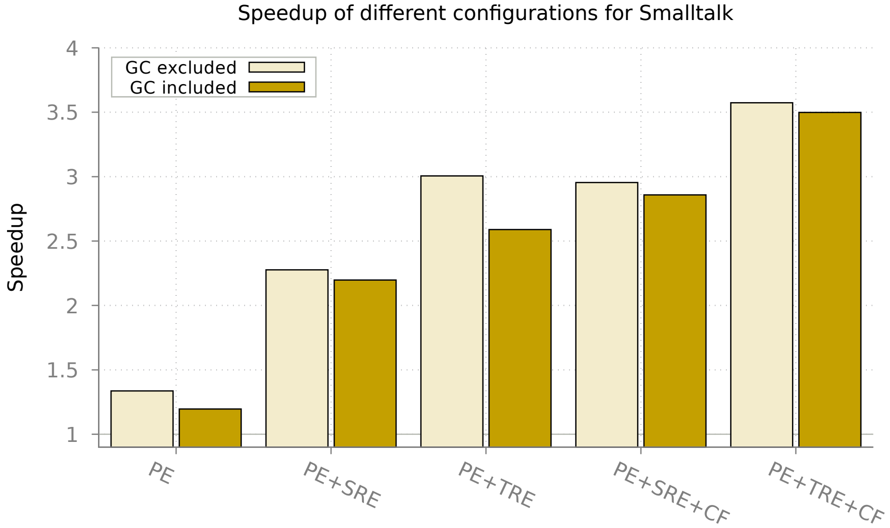


Figure 6.10: Speedup of Smalltalk against the plain PetitParser for different configurations with scannerless or tokenizing strategies.

6.4.5 Memoization Impact

The context-sensitive optimizations affect the most the context-sensitive parsers: IS Expressions, Python and Java Seas. In order to investigate in detail the impact of the *context-sensitive* analysis and *push–pop* analysis as described in subsection 6.3.3 we compare performance of parsers based on both of these analyses. The baseline is a configuration without any context-sensitive optimizations, *i.e.*, the PE+RE+CF configuration.

The impact of the *context-sensitive* analysis and the *push–pop* analysis on performance is visualized in Figure 6.11. In the case case of IS Expressions and Python, the *push–pop* analysis doubles the speedup of the *context-sensitive* analysis. The *push–pop* analysis does not improve performance of the Java Sea parser, the *context-sensitive* analysis is sufficient since Java Sea does not utilize push and pop operators.

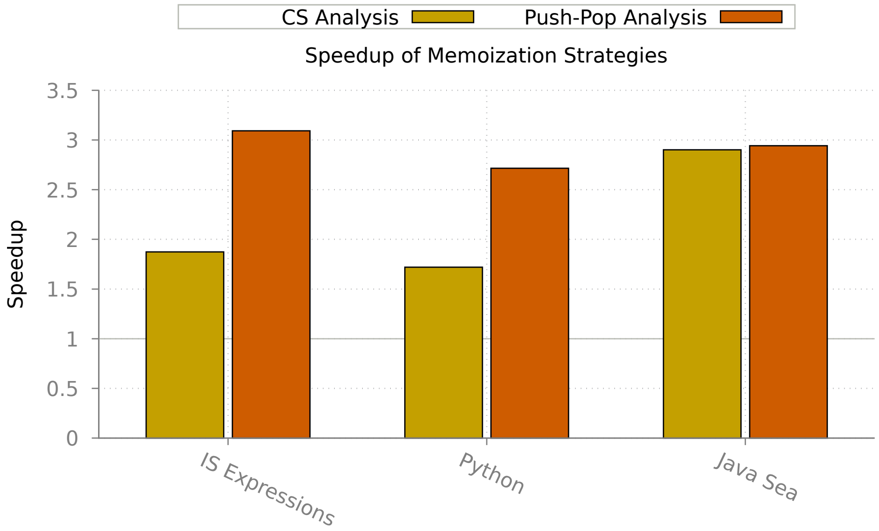


Figure 6.11: The impact of *context-sensitive* analysis and *push–pop* analysis on context-sensitive memoization.

6.4.6 Java Parsers Comparison

In the Java case study in chapter 5 the Java Sea parser performs several times worse than its island variants. The overhead is caused by the context-sensitive mementos and can be avoided by applying the context-free memoizations.

We investigate the impact of context-free memoization strategy on Java Seas in the CS configuration, *i.e.*, in the configuration we apply only the context-free memoization from subsection 6.3.3 to parser combinators (not to parsing expressions, thus CS, not PE+CS).

There is performance of a Java Sea parser, a Java Sea parser with the context-free memoization strategy (CS), and a Refined parser compared to performance of a Java parser, which serves as a baseline, in Figure 6.12. All the parsers are in their plain form, not transformed by PetitParser compiler. The Java Sea parser in the CS configuration uses a special version of combinators that do not deep-copy the data for bounded-seas unless invoked from a bounded sea combinator. As can be seen, the Java Sea parser with context-free memoizations is as fast as its island variant.

6.4.7 Smalltalk Parsers Comparison

In this section we compare performance of a Smalltalk parser compiled by a PetitParser compiler (serves as a baseline) with other implementation of Smalltalk parser available

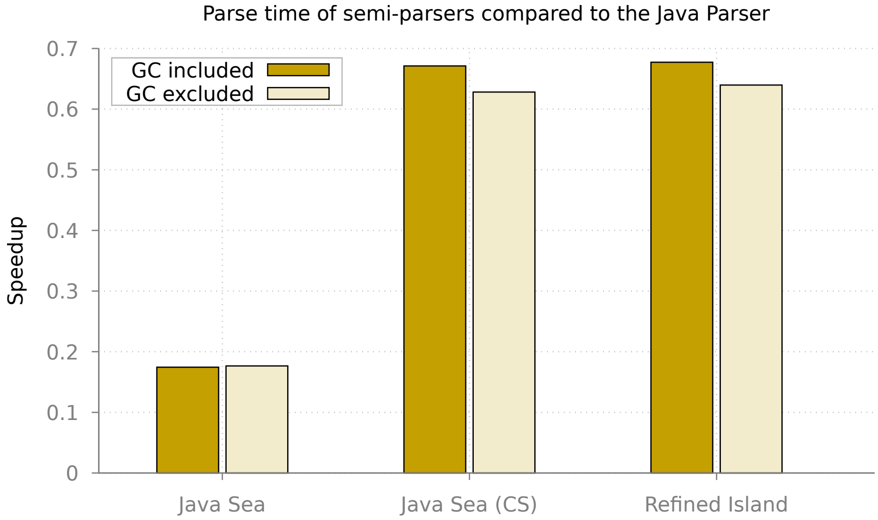


Figure 6.12: Performance of Java semi-parsers compared to the Java parser

in the Pharo environment. All of the parsers create an identical abstract syntax tree from the given Smalltalk code. The parsers are:

1. **PetitParser** is an implementation of a Smalltalk parser in PetitParser.

2. **PetitParser Compiled** is version of the above parser compiled with the PetitParser compiler. This parser serves as a baseline.

3. **Smalltalk SmaCC** is a scanning table-driven parser compiled by SmaCC [BR] from a LALR(1) Smalltalk grammar.

4. **Hand-written parser** is a parser used natively by Pharo. It is a hand-written and optimized parser and utilizes a scanner. It is probably close to the optimal performance of a hand-written parser as it is heavily used throughout the system and has therefore been extensively optimized by Pharo developers.

The speedup comparison is shown in Figure 6.13. Average time per character for each of the parsers is shown in Appendix, Figure F.35. The PetitParser is approximately three times slower than the parser compiled by a PetitParser compiler. The hand-written parser is approximately 10% faster than the parser compiled by a PetitParser compiler. The SmaCC parser is approximately two times slower than the parser compiled by a PetitParser compiler. Time per character of the compiled parser is 0.29μs, 1.04μs for the PetitParser, 0.26μs for the hand-written parser and 0.59μs for a parser generated by SmaCC.

6.5 Related Work

There has been recent research in Scala parser combinators [Ode07, MPO08] that is closely related to our work. The general idea is to perform compile-time optimizations

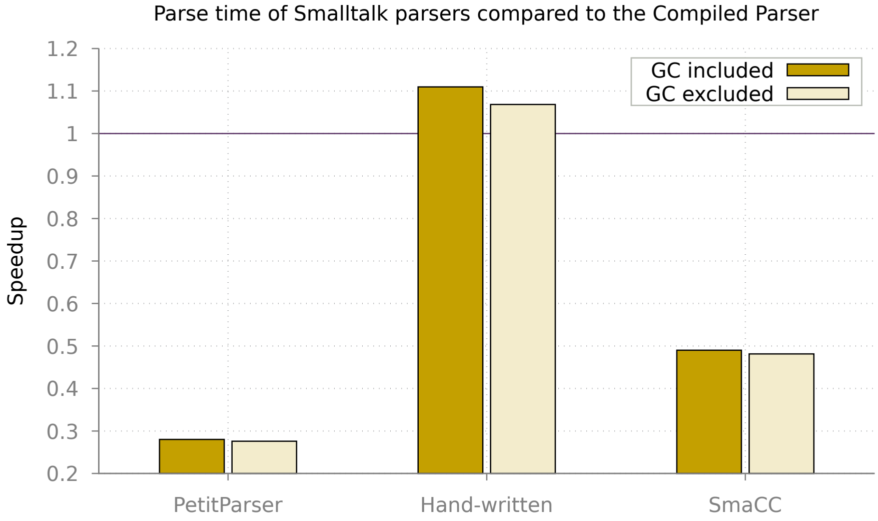


Figure 6.13: Performance speedup of Smalltalk parsers

to avoid unnecessary overhead of parser combinators at run-time. In the work *Accelerating Parser Combinators with Macros* [BJ14] the authors argue to use macros [Bur13] to remove the composition overhead. In *Staged Parser Combinators for Efficient Data Processing* [JCS$^+$14] the authors use a multi-stage programming [Tah03] framework LMS [RO10] to eliminate intermediate data structures and computations associated with a parser composition. Both works lead to a significant speedup at least for the analyzed parsers: an HTTP header parser and a JSON parser.

Similarly to our approach, ahead-of-time optimizations are applied to improve the performance. In contrast, our work does not utilize meta-programming to manipulate compiler expression trees in order to optimize parser combinators. Instead we implemented a dedicated tool from scratch. In our work, we consider several types of optimizations guided by a need to produce fast and clean top-down parsers.

Other approaches leading to better combinator performance are memoization [FS96] and Packrat Parsing [For02b] (already utilized by PetitParser). In *Efficient combinator parsers* [KP98] Koopman *et al.* use the continuation-passing style to avoid intermediate list creation.

There are table-driven or top-down parser generators such as Yacc [Joh75], Bison [Lev09], ANTLR [PQ95] or Happy [Hap10] that provide excellent performance but they do not easily support context-sensitivity.

Our work is also related to compilers supporting custom DSLs and providing interfaces for optimizations, *e.g.*, Truffle [HWW$^+$14]. Yet our approach is focused on concrete optimization techniques for parser combinators and we do not aspire for general DSL support.

6.6 Conclusion

In this chapter, we introduced a concept of adaptable parsing strategies, which we implemented in the PetitParser compiler. Based on a static analysis of a grammar the

PetitParser compiler applies different parsing strategies for different parts of a grammar. This approach leads to a significant performance speedup ranging from two to twenty-five, depending on a grammar. According to our performance analyses, performance overhead of parsing context and bounded seas can be eliminated by applying context-sensitive aware memoizations. Furthermore, by applying the techniques described in this chapter, performance of a parser based on the PEG definition can reach the performance of a hand-written scanning parser.

The implementation, tests, validations, benchmarks and measurements this chapter can be re-run. The prepared image is available online.[24]

[24] `http://scg.unibe.ch/research/parsingForAgileModeling`

7 Ruby Case study

It is generally agreed that parsing Ruby is everything but an easy task. There is no existing grammar and some rules are context sensitive. It is a major engineering task to extract information from Ruby code.[1]

In this case study we utilize concepts presented in this work to trade precision for time while extracting information from Ruby code. We prototype a Ruby semi-parser while utilizing techniques of agile modeling and extensions presented in this work: (i) we implement the parser in multiple iterations and for each of them we measure precision and recall of extracted information;[2] (ii) we use indentation to recognize structural elements; (iii) we utilize a parsing context to define context-sensitive rules; (iv) we use bounded seas to skip over unknown parts; and (v) we use a parser compiler to speedup the development cycle.

Our study confirms that indentation serves as a good proxy [HGH08] and we reach high precision and recall quickly. Adding context-sensitive rules for strings and comments we reach the limits of our indentation-based approach. We also extract method calls with high precision as well. However, several iterations are needed to improve recall. The parser based on PetitParser is relatively slow and the performance is improved by a factor of nine when utilizing dynamic strategies.

The chapter is organized as follows: In section 7.1 we implement a parser to recognize structural elements of Ruby code (modules, classes and methods) utilizing indentation. In section 7.2 we iteratively extend the parser to recognize method calls and we iteratively improve its results. In section 7.3 we investigate performance of the implemented parser and finally section 7.4 concludes this paper.

7.1 Ruby Structure

The standard approach to recognize structure of input is to track all the language elements that affect the structure. We use this approach in the Java case study (see subsec-

[1] `http://www.webcitation.org/6k65YVVOG`

[2] We use the JRuby parser as an oracle.

tion 6.4.6) where we define a rule for blocks. It turns out to be a much harder problem in the case of Ruby, because of the *dangling end problem*, which we describe in the following section.

7.1.1 The Dangling End Problem

Ruby poses interesting parsing challenges not only for a traditional parsers, but even for semi-parsers. The problem we faced when recognizing structure is the *dangling end problem*. Normally a control structure like an `if` statement terminates with `'end'`. However there is also an *if modifier* which does not require `'end'`:

```
return error if check?
```

There exist numerous such modifiers in Ruby,[3] which resemble conditional blocks, but have a different syntax. Such modifiers pose a problem for parsing. From the perspective of an imprecise parser, it is hard to distinguish between a modifier and a loop or a conditional block.

Ruby structures (such as classes, methods, blocks) end with the `'end'` keyword (see Listing 7.1). To capture the structure of Ruby code, we need to define rules for these structural elements, including conditional blocks and others, such as loops, do blocks, and brace pairs.

```
  class Shape
    def draw
      if (x > 0)
        do_something()
      end
    end
  end
```

Listing 7.1: Example of a Ruby code.

Ruby modifiers are not paired with any `'end'` as we can see in Listing 7.2. If we incorrectly pair `'end'`, we change the structure of a program. Unfortunately, it is hard to precisely recognize when `'if'` belongs to a modifier and when to a conditional block unless we specify an almost complete grammar to recognize all the constructs.

```
class Shape
  def draw
    return error if check?
    if (x > 0)
      do_something
    end
  end
end
```

Listing 7.2: Example of Ruby code where `'if'` is not paired with any `'end'`.

[3] `if`, `unless`, `while` and `until` (see `http://www.webcitation.org/6k7fivIGJ`)

Lookbehind

Iyadurai extracted structure with the help of an island parser, which consists of forty parsing rules [Iya16]. This approach is based on the observation that, in the test data we used, the only case where the starting keyword of a conditional block is in the middle of a line, is to assign its result with an equals sign. Therefore a lookbehind for `'='` suffices to differentiate a modifier keyword from a regular keyword.

Indentation

In this work we focus on indentation. We exploit the fact that indentation is a good proxy for structure. We define a context-sensitive parser that uses indentation to differentiate modifiers from a regular keyword. From the perspective of indentation, modifiers look like loops or conditional blocks with a single line scope. As we shall see, the use of indentation-sensitive rules simplifies the implementation of the parser.

We define a context-sensitive grammar that recognizes modules, classes, methods and class methods in Ruby code by utilizing indentation and bounded seas. The scope of a module, class or method extends as far as code appears to the right of the module, class or method declaration (*i.e.*, in the onside position). In the terminology of bounded seas, the right boundary of seas is code in the offside position.

The definition of onside content is in Listing 7.3. We reuse the layout-sensitive `onside` definition from Listing 4.7. The definition of `module` is in Listing 7.4. Similarly we define `class` and `method`. The final grammar can be expressed on a single page (see Listing 7.5). The advantage of the grammar utilizing indentation is that a language engineer does not need to specify the rules for block elements (*e.g.*, `for`, `if-then-else`, *etc.*) and can focus only on the parts of her interest. The disadvantage is that this approach expects code to be properly indented.

```
primary ← class /
          module /
          method

content ← ∼(onside, primary)∼+ / ∼ϵ∼
```

Listing 7.3: Indentation sensitive definition of a Ruby structure.

```
module    ← setOL 'module' cpath
              content
            end, removeOL

end         ← &#letter (aligns/offside) / #eof
```

Listing 7.4: Definition of a Ruby module.

```
primary    ← class /
             module /
             method

content    ← ~(onside, primary)~+ / ~ϵ~


module     ← setOL 'module' cpath
                content
             end, removeOL

class      ← setOL 'module' cpath
                content
             end removeOL

method     ← defn / defs

defs       ← setOL identifier '.' fname
                content
             end, removeOL

defs       ← setOL fname
                content
             end, removeOL

end        ← &#letter (aligns/offside) / #eof

identifier ← #leter (#leter / #digit / '_')*
cpath      ← identifier ('::' identifier)*
fname      ← (identifier ('?' / '!' / '=')) /
             '..' / '!' / '^' / '<=>' / '==' /
             ...
```

Listing 7.5: Indentation sensitive definition of a Ruby structure.

7.1.2 Measurements

To measure precision and recall, we use *jruby-parser*[4] as a reference parser. We compare the structure (modules, classes, methods and class methods) of Ruby code detected by jruby-parser with the structure detected by our parser. To compare structure, we list elements forming the structure (*i.e.*, modules classes and methods). Each element in the list is identified by a unique path determined by the placement of the element. For example

```
<module>Graphics
<class> Graphics::Shape
<defn>  Graphics::Shape.draw
```

refers to a method `draw` defined in a class `Shape`. `Shape` belongs to a `Graphics` module. Another example

[4] http://www.webcitation.org/6k65g9UGW

```
<class>Shape
<class>Shape::Renderer
<defs> Shape::Renderer.instance
```

refers to a class-side method `instance` of the inner class `Renderer` nested in a `Shape` class.

Test Data We perform our study on a sample of N =5055 files of five popular projects on GitHub: Rails[5], Discourse[6], Diaspora[7], Cucumber[8] and Valgrant.[9] The files contain in total 41522 classes and methods.

Results

The grammar from Listing 7.5 recognizes structure of code with 0.933 precision and 0.985 recall. The result can be further improved by specifying `string` (including heredoc as described in subsection 4.2.1) and `comment`.[10] This results in 0.997 precision and 0.995 recall. The perfect precision and recall cannot be reached, because some code is not indented properly and thus assigned to a wrong class. Moreover, a problem are strings representing regular expressions. They have the following syntax: `/regex/`). Such a string is, without further information, indistinguishable from two consecutive `/` operators, *e.g.*, `not/regex/but/path`, which misleads the parser.

7.2 Ruby Method Calls

Regarding code structure we reach promising results quickly. In the next step we try a more challenging task: extracting *method calls*. We start with a *naive* implementation of a method call:

```
call ← '.' identifier
```

However this does not allow a parser to detect receivers, so we extend it with a naive implementation of a call with *cpath* receiver:

```
call ← cpath '.' identifier
```

Unfortunately, this implementation has rather low recall, approximately 0.8. In the following steps, we iteratively identify elements that are missed by our implementation and extend the definition with more and more information. We (i) add support for array accessors (*e.g.*, `colors[2]`) and *special* selectors (*e.g.*, `>>`); (ii) add support for *prefix* selectors (*e.g.*, `!true`); (iii) add support for *functions* as receivers (*e.g.*, `getColor().asHex`); (iv) extend *function* arguments with seas of calls to detect method calls in between brackets `()` and `[]`; (v) add support for *assignments* to

[5] http://www.webcitation.org/6k65olp6x

[6] http://www.webcitation.org/6k65pVD7m

[7] http://www.webcitation.org/6k65qF4wC

[8] http://www.webcitation.org/6k65qupqe

[9] http://www.webcitation.org/6k65rjikI

[10] The correct implementation of all the variants of Ruby strings was actually the major tasks when developing the parser..

distinguish them from method calls (*e.g.*, to reject `color.red = 25`); and last but not least (vi) extend the definition of *strings* with seas of escaped code to detect method calls inside strings, for example:

```
"the value of color is: #{getColor().asHex}"
```

All these extensions result in fifteen additional rules to the parser, three of which[11] are implemented utilizing bounded seas.

7.2.1 Measurements

To measure precision and recall, we again use *jruby-parser*. We compare the call nodes of Ruby code detected by jruby-parser with the call nodes detected by our parser. To compare the call nodes, we describe it as a list of receivers and selectors. Each element of the list is identified by a unique path determined by the placement of the element. For example

```
<const> Graphics::Shape.draw+color
<call>  Graphics::Shape.draw->rgb
<call>  Graphics::Shape.draw->asCMYK
```

refers to a method call `color.rgb.asCMYK`, *i.e.*, two calls, `color` as a receiver and `rgb` as a selector. In the case of the `asCMYK` selector, the receiver is the method call `color.rgb`. The calls are in a `draw` method of a `Graphics::Shape` class.

Test Data We perform our study on the same sample of N =5015 files of the same projects on Github: Rails, Discourse, Diaspora, Cucumber andValgrant. The files contain 141537 receivers and 196758 selectors.

Results

When detecting selectors, the *naive* approach results in 0.964 precision and 0.798 recall. With all the extensions we describe, the precision and recall reach 0.987 and 0.966 respectively. All the intermediate stages are visualized in Figure 7.1. The intermediate stages improve recall except for the *cpath* step, which just detects receivers having no effect on selectors, and the *assign* step, which improves precision. The *function* intermediate step even decreases recall, but the recall is fixed in the following *arguments* step (it better describes function arguments). The reason for imperfect precision are errors in detected structure and mismatch between regex strings and consecutive `/` operators. The main cause of imperfect recall are features that we have not implemented yet or have been implemented only partially, *e.g.*, the parser misses method calls in strings escaped with `#[]` (we implemented only `#{}`).

When detecting receivers, the *cpath* step results in 0.968 precision and 0.789 recall. With all the described extensions, the precision and recall reach 0.984 and 0.973 respectively. All the intermediate stages are visualized in Figure 7.2. The intermediate stages improve recall except for the *assign* step, which improves precision of detected selectors and does not affect receivers. The reason for imperfect precision are errors

[11] round bracket arguments, square bracket arguments and strings

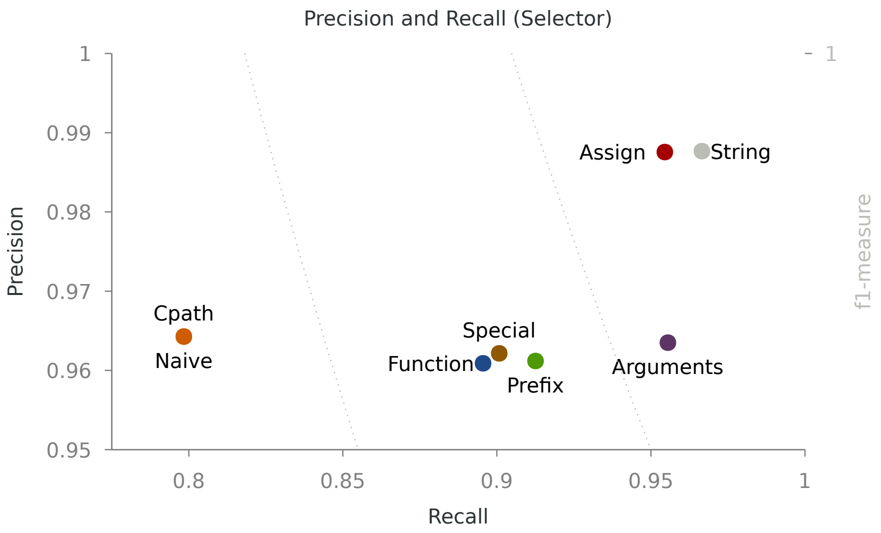


Figure 7.1: Precision and recall of detected selectors in various stages of our method call implementation.

in detected structure and improperly handled infix operators (*e.g.*, in arithmetic expressions). The main cause of imperfect recall are features that we have not implemented yet, for example calls with method nodes as receivers.

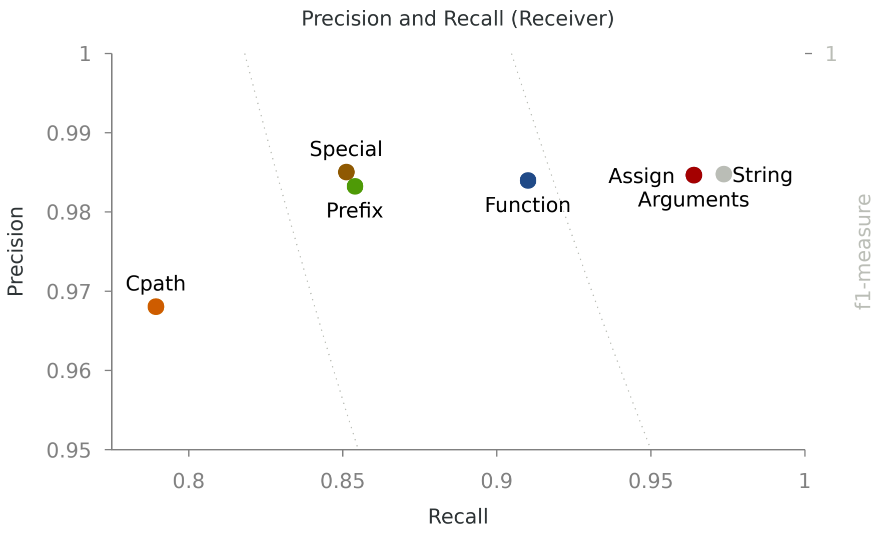


Figure 7.2: Precision and recall of detected receivers in various stages of our method call implementation.

7.3 Performance

In this section we focus on performance of the implemented parser. We investigate the impact of additional rules on the performance of the parser as well as the impact of a parser compiler. We also visualize dependency between the input size and time to parse.

How we measured We measure performance on a random sample of N =312 files of the same projects on Github: Rails, Discourse, Diaspora, Cucumber, Valgrant, Typhoeus. We run each benchmark ten times using the latest release of the Pharo VM for Linux.[12] All the parsers and inputs are initialized in advance, then we measure time to parse.

We report on speedup (the ratio between original PetitParser serving as a baseline and its compiled version) and time per character. When measuring speedup, we consider the best time. To estimate impact of a garbage collector, we collect both times, with and without the garbage collection. When showing time per character, we visualize five-number summary, median is represented by a bar, lowest value, first, third quartiles and highest value are represented by a box with whiskers.

Results Average time per character of the Ruby parser in PetitParser is 50μs. The parser compiler offers approximately nine times speedup resulting in 8μs per character. Majority of time of the compiled version is spent in the sea memoization and in the search for a boundary. Speedup and time per character are in Figure 7.3 and Figure 7.4 respectively.

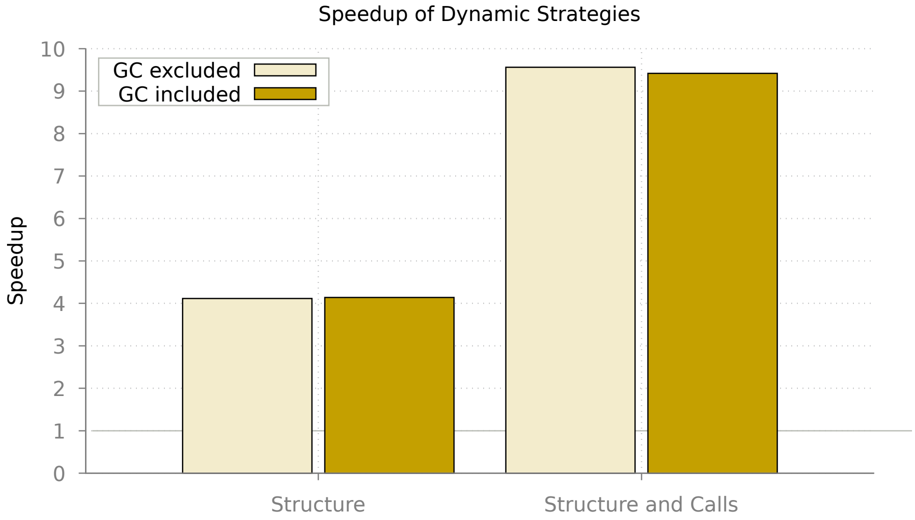


Figure 7.3: Speedup of dynamic strategies against plain PetitParser for the Ruby parsers from section 7.1 (Structure) and section 7.2 (Structure and calls).

The performance of the Ruby parser that extracts only structure is better than the performance of the parser that extracts structure and calls. On the other hand, the Ruby

[12] from October 7, 2016

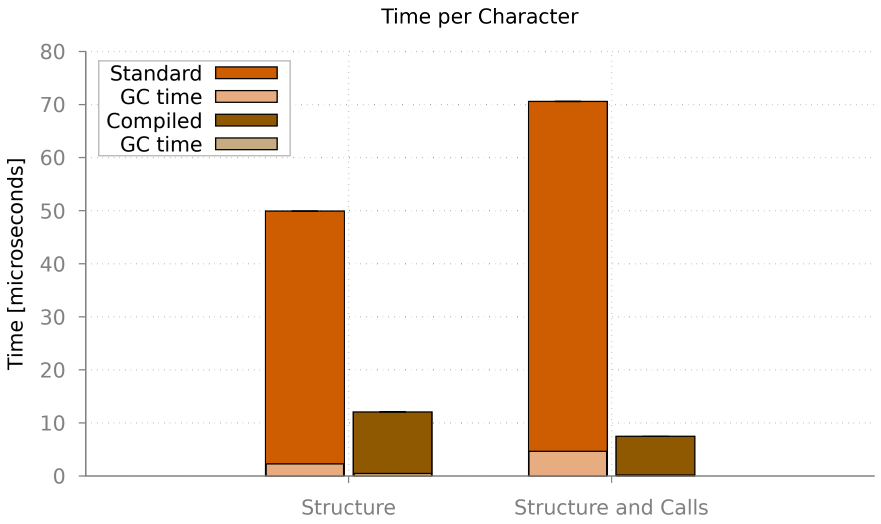


Figure 7.4: Time per character of Ruby parsers from section 7.1 (Structure) and section 7.2 (Structure and calls) for both plain PetitParser and its compiled variant with dynamic strategies.

parser that extracts only structure and that is compiled with dynamic strategies has worse performance than the parser that extracts all the information and that is compiled with dynamic strategies. The extra information allows to apply more efficient parsing strategies, which results in better overall performance. The dependency between parse time and input size is linear as shown in Figure 7.5.

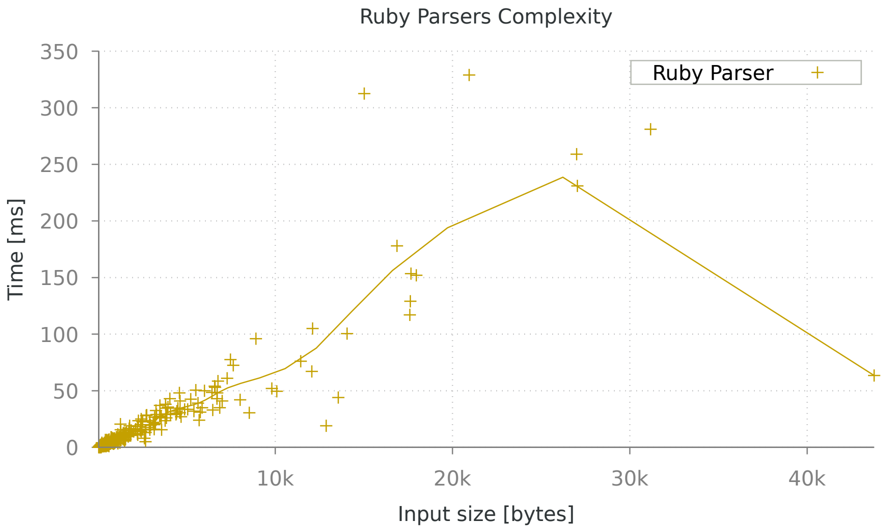


Figure 7.5: The dependency between the input size and time to parser for the Ruby parser.

Compared to the Python island parser and the Java Sea parser (used in case studies in chapter 4 and chapter 5 respectively), the Ruby parser is significantly slower (see Figure 7.6). Based on our analysis, this is caused by three to four times more full mementos created per character when compared to the Python and Java parsers. Furthermore, the amount of seas invoked per character is also three times higher compared to the Java parser, which results in more memoizations in the sea implementation. Last but not least, the Ruby parser is backtracking more than the Python and Java parsers, which again negatively affects performance.

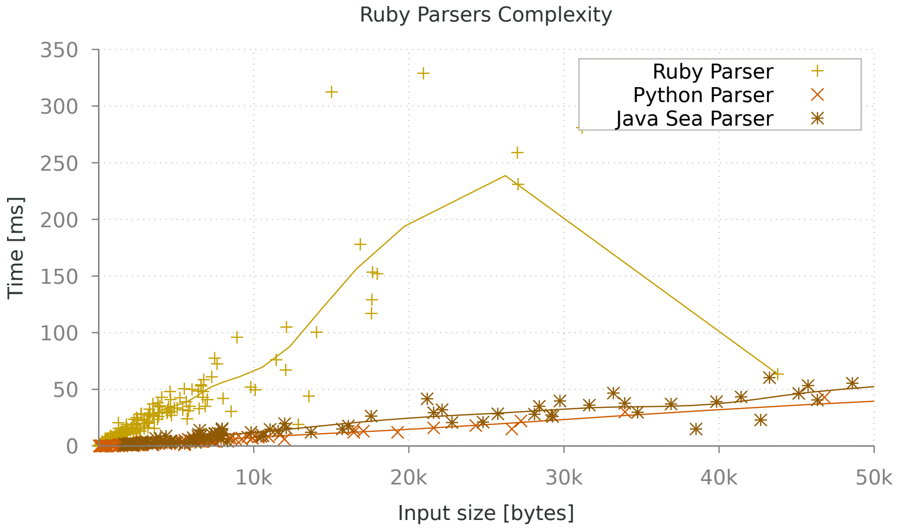


Figure 7.6: Dependency between the input size and parse time of the Ruby bounded sea parser, the Python island parser and the Java bounded sea parser.

7.4 Conclusion

In this chapter we implemented a semi-parser for Ruby that utilizes indentation, bounded seas and context-sensitive definitions of Ruby strings. We extract structure with 0.99 precision and 0.99 recall and we recognize receivers and selectors with precision above 0.95. We improve the initially low recall of extracted method calls in several iteration by adding more information. This results in 0.95 recall. With dynamic strategies we reach speedup by the factor of nine. According to our observations, precision, recall and performance are better, the more grammar rules are specified.

The resulting parser contains approximately ten rules to describe structural elements (modules, classes and method definitions), fifteen rules to describe method calls and another ten rules that describe comments, strings and other helper rules. Though the parser described in this chapter is far from complete, with the help of technologies introduced in this work it is able to provide feedback from the very early stages of the development.

We do not apply any heuristics or semi-automation techniques to specify the parser. All the work is done by the parser developer. We reduce the burden placed on the developer with indentation-sensitive rules and bounded seas and we speedup the development cycle thanks to the dynamic parsing strategies.

The implementation, tests, validations and measurements from this chapter can be re-run. The prepared image is available online.[13]

[13] `http://scg.unibe.ch/research/parsingForAgileModeling`

8
Conclusion

Agile modeling imposes specific requirements on the parsing technology used, especially in the areas of expressiveness, composability, tolerance and performance. Unfortunately, these requirements are not met by the current state of the art parsing technology. The main problem lies in the fact that current parsing algorithms try to find a compromise between flexibility and performance. They limit the expressiveness of grammars to improve performance while using relatively simple parsing algorithms.

In this work we (i) extend PEGs with context-sensitive definitions to improve the expressiveness of PEGs; (ii) extend PEGs with bounded seas, a novel and composable approach to tolerant parsing; and finally (iii) we introduce a parser compiler; a source-to-source code translator that combines multiple parsing strategies in a single parser.

Parsing contexts (see chapter 4) allow for context-sensitive grammar definitions while preserving understandability and semantic suitability of context-free grammars. Parsing contexts manipulate elements in a context via push and pop operations thanks to which the parsing contexts are easy to optimize; based on our measurements the *push–pop* analysis doubles the speedup of context-free optimizations (see Figure 6.11). Parsing contexts allow a grammar engineer to specify layout-sensitive definitions (see subsection 4.5.1) as well as to formalize the CommonMark grammar (see Appendix G).

Bounded seas (see chapter 5) introduce a non-ambiguous operator to PEGs that can skip an unknown input until a desired input is found. Bounded seas are highly composable and can be easily integrated into any grammar structure. Parsers based on bounded seas are robust and can extract information from source code with high precision and recall, low effort and can be easily extended. When extracting structure from Java code, the sea parser even outperforms a traditional parser (in precision and recall as well as in effort required by a grammar engineer) as demonstrated in section 5.5. Moreover, bounded seas can be used with layout-sensitive parsing to simplify a grammar definition and to further improve precision and and recall of extracted data (see section 7.1).

The novelty of a parser compiler (see chapter 6) is that to parse input a sophisticated runtime environment is used instead of a single parsing algorithm. The parser compiler analyzes a grammar, chooses the most appropriate parsing strategy and generates a top-down parser from a PEG definition. The parser compiler significantly reduces overhead of parser combinators, scannerless parsing and context-sensitive definitions while preserving their expressiveness. In the case of PetitParser the speedup of a parser compiled by a parser compiler ranges from two to five for context-free grammars and five to twenty-five for context-sensitive grammars. Based on our case study, the performance of a parser compiled from a PEG-based Smalltalk grammar is comparable to a hand-written optimized scanner-parser pipeline (see subsection 6.4.7).

To validate our ideas, we integrate parsing contexts, bounded seas and a compiler into the PetitParser framework.[1] We use the extended version of PetitParser in the Ruby case study (see chapter 7).

We use parsing contexts to define a layout-sensitive grammar that detects structure of Ruby code. We also utilize parsing contexts to specify context-sensitive parts of Ruby (*e.g.*, here documents). To save time we utilize bounded seas to extract only information of our interest, without specifying the complete grammar. Based on the ideas of agile modeling, we incrementally refine the grammar and extract more information with better precision, recall and even performance. To increase the grammar engineer's comfort, we use a parser compiler, which significantly reduces parse time and speeds up the development cycle. In the end we are able to extract structure with both precision and recall over 0.99 and to extract method calls and their receivers with both precision and recall over 0.95 while the complete grammar definition does not contain more than thirty rules.

[1] `http://scg.unibe.ch/research/parsingForAgileModeling`

Bibliography

[AA14] Michael D. Adams and Ömer S. Ağacan. Indentation-sensitive parsing for Parsec. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 121–132, New York, NY, USA, 2014. ACM.

[Ada13] Michael D. Adams. Principled parsing for indentation-sensitive languages: Revisiting Landin's offside rule. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 511–522, New York, NY, USA, 2013. ACM.

[AI15] Ali Afroozeh and Anastasia Izmaylova. Faster, practical GLL parsing. In Bjorn Franke, editor, *Compiler Construction*, volume 9031 of *Lecture Notes in Computer Science*, pages 89–108. Springer Berlin Heidelberg, 2015.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, Mass., 1986.

[Asv95] P.R.J. Asveld. A fuzzy approach to erroneous inputs in context-free language recognition. In *Proceedings of the Fourth International Workshop on Parsing Technologies IWPT'95*, pages 14–25, Prague, Czech Republic, 1995. Institute of Formal and Applied Linguistics, Charles University.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling Volume I: Parsing*. Prentice-Hall, 1972.

[Bac79] Roland C. Backhouse. *Syntax of Programming Languages: Theory and Practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.

[BBC$^+$10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[BDH$^+$01] M. G. J. Brand, A. Deursen, J. Heering, H. A. Jong, M. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. *Compiler Construction: 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April*

2–6, 2001 Proceedings, chapter The ASF+SDF Meta-environment: A Component-Based Language Development Environment, pages 365–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[BG82] Philip A. Bernstein and N. Goodman. A sophisticate's introduction to distributed concurrency control. In *Proceedings of the Eighth International Conference on Very Large Data Bases*, pages 62–76, 1982.

[Bis92] Walter R. Bischofberger. Sniff: A pragmatic approach to a C++ programming environment. In *C++ Conference*, pages 67–82, 1992.

[BJ14] Eric Béguet and Manohar Jonnalagedda. Accelerating parser combinators with macros. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA '14, pages 7–17, New York, NY, USA, 2014. ACM.

[BM06] Leonhard Brunauer and Bernhard Mühlbacher. Indentation sensitive languages. http://www.cs.uni-salzburg.at/ ck/content/classes/TCS-Summer-2006/index.html, 2006.

[BR] John Brant and Don Roberts. SmaCC, a Smalltalk Compiler-Compiler. http://www.refactory.com/Software/SmaCC/.

[BS10] John Boyland and Daniel Spiewak. Tool paper: ScalaBison recursive ascent-descent parser generator. *Electron. Notes Theor. Comput. Sci.*, 253:65–74, September 2010.

[Bur75] William H. Burge. *Recursive programming techniques*. The systems programming series. Addison-Wesley, Reading (Mass.), 1975.

[Bur13] Eugene Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[Cho57] Noam Chomsky. *Syntactic Structures*. Mouton and Co, The Hague, 1957.

[CHP88] James R. Cordy, Charles D. Halpern, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings of The International Conference of Computer Languages*, pages 280–285, Miami, FL, October 1988.

[Chr09] Henning Christiansen. Adaptable grammars for non-context-free languages. In Joan Cabestany, Francisco Sandoval, Alberto Prieto, and JuanM. Corchado, editors, *Bio-Inspired Systems: Computational and Ambient Intelligence*, volume 5517 of *Lecture Notes in Computer Science*, pages 488–495. Springer Berlin Heidelberg, 2009.

[Cor06] James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[Cox07] R. Cox. Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), 2007. `http://swtch.com/~rsc/regexp/regexp1.html`.

[DCMS03] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Agile parsing in TXL. *Autom. Softw. Eng.*, 10(4):311–336, 2003.

[dJNNKV09] Maartje de Jonge, Emma Nilsson-Nyman, Lennart C. L. Kats, and Eelco Visser. Natural and flexible error recovery for generated parsers. In Mark G. J. van den Brand and Jeff Gray, editors, *Software Language Engineering (SLE 2009)*, Lecture Notes in Computer Science, Heidelberg, oct 2009. Springer.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[EKV09] Giorgios Economopoulos, Paul Klint, and Jurgen Vinju. *Compiler Construction: 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, chapter Faster Scannerless GLR Parsing, pages 126–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[ERKO12] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In *SLE*, pages 244–263, 2012.

[EV06] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, pages 1–4, September 2006.

[FHC07] Richard A. Frost, Rahmatullah Hafiz, and Paul C. Callaghan. Modular and efficient top-down parsing for ambiguous left-recursive grammars. In *Proceedings of the 10th International Conference on Parsing Technologies*, IWPT '07, pages 109–120, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics.

[FM13] Roberto Ierusalimschy Fabio Mascarenhas, Sérgio Medeiros. On the relation between context-free grammars and parsing expression grammars. *CoRR*, abs/1304.3177, 2013.

[For02a] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.

[For02b] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37/9, pages 36–47, New York, NY, USA, 2002. ACM.

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[FS96] Richard A. Frost and Barbara Szydlowski. Memoizing purely functional top-down backtracking language processors. *Science of Computer Programming*, 27(3):263–288, November 1996.

[GÎ0] Tudor Gîrba. The Moose book, 2010.

[Gam97] Erich Gamma. Extension object. In *Pattern languages of program design 3*, pages 79–88. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[GHVJ93] Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Giv13] Attieh Sadeghi Givi. Layout sensitive parsing in the PetitParser framework. Bachelor's thesis, University of Bern, October 2013.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.

[GJ08a] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques — A Practical Guide.* Springer, 2008.

[GJ08b] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques — A Practical Guide*, chapter 8: Deterministic Top-Down Parsing, pages 235–361. Volume 1 of Gries and Schneider [GJ08a], 2008.

[GM95] Andy Gill and Simon Marlow. Happy: the parser generator for haskell. *University of Glasgow*, 1995.

[Gra] Grace. http://gracelang.org/documents/grace-spec031303.pdf.

[Gug15] Joël Guggisberg. Automatic token classification — an attempt to mine useful information for parsing. Bachelor's thesis, University of Bern, December 2015.

[Hap10] Happy — the parser generator for Haskell, 2010. http://tfs.cs.tu-berlin.de/agg/index.html.

[Has] Haskel 98 Report.

[HGH08] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Indentation as a proxy for complexity metrics. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 133–142, Washington, DC, USA, 2008. IEEE Computer Society.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual–. *SIGPLAN Not.*, 24(11):43–75, November 1989.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[HS91] Stephan Heilbrunner and Lothar Schmitz. An efficient recognizer for the boolean closure of context-free languages. *Theoretical Computer Science*, 80(1):53 – 75, 1991.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

[HWW$^+$14] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing AST interpreters. *SIGPLAN Not.*, 50(3):123–132, September 2014.

[ISO96] Information Technology – Syntactic metalanguage – Extended BNF, 1996.

[Iya16] Rathesan Iyadurai. Parsing Ruby with an island parser. Bachelor's thesis, University of Bern, April 2016.

[JCS$^+$14] Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. Staged parser combinators for efficient data processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14*, pages 637–653, New York, New York, USA, 2014. ACM Press.

[JKSV12] Maartje de Jonge, Lennart C. L. Kats, Emma Soderberg, and Eelco Visser. Natural and flexible error recovery for generated modular language environments. *ACM Transactions on Programming Languages and Systems*, 34(4), December 2012. Article No. 15, 50 pages.

[Joh75] S.C. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

[Jos85] Aravind K.. Joshi. *Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?* Cambridge University Press, 1985.

[JS97] Aravind Joshi and Yves Schabes. Tree-adjoining grammars, 1997.

[KdJNNV09] C. L. Lennard Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In Gary T. Leavens, editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2009)*, ACM SIGPLAN Notices, New York, NY, USA, October 2009. ACM Press.

[KL03] Steven Klusener and Ralf Lämmel. Deriving tolerant grammars from a base-line grammar. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 179–188. IEEE Computer Society, September 2003.

[Kli93] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, 1993.

[KLN14a] Jan Kurš, Mircea Lungu, and Oscar Nierstrasz. Bounded seas: Island parsing without shipwrecks. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 62–81. Springer International Publishing, 2014.

[KLN14b] Jan Kurš, Mircea Lungu, and Oscar Nierstrasz. Top-down parsing with parsing contexts. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014.

[KLR$^+$13] Jan Kurš, Guillaume Larcheveque, Lukas Renggli, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. PetitParser: Building modular parsers. In *Deep Into Pharo*, page 36. Square Bracket Associates, September 2013.

[KLV05] A.S. Klusener, R. Lämmel, and C. Verhoef. Architectural modifications to deployed software. *Sci. Comput. Program.*, 2005.

[Knu68] DonaldE. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[Knu90] Donald E. Knuth. The genesis of attribute grammars. In *Proceedings of the International Conference WAGA on Attribute Grammars and their Applications*, pages 1–12, London, UK, UK, 1990. Springer-Verlag.

[Kob05] Markus Kobel. Parsing by example. Diploma thesis, University of Bern, April 2005.

[Kop97] Rainer Koppler. A systematic approach to fuzzy parsing. *Software: Practice and Experience*, 27(6):637–649, 1997.

[Kos91] C. H. A. Koster. *Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings*, chapter Affix grammars for programming languages, pages 358–373. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[KP98] Pieter Koopman and Rinus Plasmeijer. Efficient combinator parsers. In *In Implementation of Functional Languages, LNCS*, pages 122–138. Springer-Verlag, 1998.

[KRS08] Richard Kelsey, Jonathan Rees, and Mike Sperber. The incomplete Scheme 48 reference manual for release 1.8, February 2008.

[KV10] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *OOPSLA'10: Proceedings of the 25th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 444–463, Reno/Tahoe, NV, USA, October 2010.

[KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 168–177, 2009.

[KVG+16] Jan Kurš, Jan Vraný, Mohammad Ghafari, Mircea Lungu, and Oscar Nierstrasz. Optimizing parser combinators. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2016)*, 2016. To Appear.

[Lan66] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[Lev09] J. Levine. *Flex & Bison: Text Processing Tools*. O'Reilly Media, 2009.

[LM01] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world, 2001.

[LS75] M.E. Lesk and E. Schmidt. Lex — A lexical analyzer generator. Computer Science Technical Report #39, Bell Laboratories, Murray Hill, NJ, 1975.

[LT93] Alon Lavie and Masaru Tomita. GLR* — an efficient noise-skipping parsing algorithm for context free grammars. In *In Proceedings of the Third International Workshop on Parsing Technologies*, pages 123–134, 1993.

[LV01] Ralf Lämmel and Chris Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, November 2001.

[LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings Eighth Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society, October 2001.

[MPO08] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical report, Department of Computer Science, K.U. Leuven, February 2008.

[ND04] Oscar Nierstrasz and Stéphane Ducasse. Moose – a language-independent reengineering environment. *European Research Consortium for Informatics and Mathematics (ERCIM) News*, 58:24–25, July 2004.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper.

[NK15] Oscar Nierstrasz and Jan Kurš. Parsing for agile modeling. *Science of Computer Programming*, 97, Part 1(0):150–156, 2015.

[NL12] Oscar Nierstrasz and Mircea Lungu. Agile software assessment. In *Proceedings of International Conference on Program Comprehension (ICPC 2012)*, pages 3–10, 2012.

[NNEH09] Emma Nilsson-Nyman, Torbjörn Ekman, and Görel Hedin. Practical scope recovery using bridge parsing. In Dragan Gašević, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 95–113. Springer Berlin Heidelberg, 2009.

[Ode07] Martin Odersky. Scala language specification v. 2.4. Technical report, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, March 2007.

[Okh04] Alexander Okhotin. Boolean grammars. *Information and Computation*, 194(1):19 – 48, 2004.

[Okh05] Alexander Okhotin. On the existence of a boolean grammar for a simple programming language. In *Proceedings of AFL 2005 (May 17-20, 2005, Dobogoko*, 2005.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers, May 2007.

[Pen86] Thomas J. Pennello. Very fast LR parsing. *SIGPLAN Not.*, 21(7):145–151, July 1986.

[PF11] Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

[PQ94] Terence J. Parr and Russell W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *CC '94: Proceedings of the 5th International Conference on Compiler Construction*, pages 263–277, London, UK, 1994. Springer-Verlag.

[PQ95] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25:789–810, 1995.

[PW86] F Pereira and D Warren. Readings in natural language processing. In Barbara J. Grosz, Karen Sparck-Jones, and Bonnie Lynn Webber, editors, *Readings in Natural Language Processing*, chapter Definite Clause Grammars for Language Analysis, pages 101–124. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

[Pyt] Python 3.0 Grammar. https://docs.python.org/3/reference/grammar.html.

[RDGN10] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, pages 1–4, Malaga, Spain, June 2010.

[Red09] Roman R. Redziejowski. Applying classical concepts to parsing expression grammar. *Fundam. Inf.*, 93(1-3):325–336, January 2009.

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[RVB$^+$12] dos Santos Reis, Leonardo Vieira, da Silva Bigonha, Roberto, Di Iorio, Vladimir Oliveira, de Souza Amorim, and Luis Eduardo. Adaptable parsing expression grammars. In FranciscoHeron de Carvalho Junior and LuisSoares Barbosa, editors, *Programming Languages*, volume 7554 of *Lecture Notes in Computer Science*, pages 72–86. Springer Berlin Heidelberg, 2012.

[SC89] D. J. Salomon and G. V. Cormack. Corrections to the paper: Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Not.*, 24(11):80–83, nov 1989.

[SD96] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*, pages 184–207, London, UK, UK, 1996. Springer-Verlag.

[SJ10] Elizabeth Scott and Adrian Johnstone. GLL parsing. *Electron. Notes Theor. Comput. Sci.*, 253(7):177–189, September 2010.

[SJC02] Frederick T Sheldon, Kshamta Jerath, and Hong Chung. Metrics for maintainability of class inheritance hierarchies. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(3):147–160, 2002.

[SMDV06] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.

[SMFK91] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191 – 229, 1991.

[Swi01] S.D. Swierstra. Combinator parsers: From toys to tools. *Electronic Notes in Theoretical Computer Science*, 41(1):38 – 59, 2001. 2000 ACM SIGPLAN Haskell Workshop (Satellite Event of PLI 2000).

[Tah03] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

[Ter05] P. Terry. *Compiling with C# and Java*. Pearson education. Pearson/Addison-Wesley, 2005.

[Tho68] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

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

[Tra10] Laurence Tratt. Direct left-recursive parsing expression grammars. Technical Report EIS-10-01, School of Engineering and Information Sciences, Middlesex University, oct 2010.

[Val75] Leslie G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308 – 315, 1975.

[VeaB98] Eelco Visser and Zine el-abidine Benaissa. A core language for rewriting. In *Electronic Notes in Theoretical Computer Science*, pages 1–4. Elsevier, 1998.

[Vis97a] Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, jul 1997.

[Vis97b] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[Vis02] Eelco Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, oct 2002. Springer-Verlag.

[VSWJ87] K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of the 25th Annual Meeting on Association for Computational Linguistics*, ACL '87, pages 104–111, Stroudsburg, PA, USA, 1987. Association for Computational Linguistics.

[WDM08] Alessandro Warth, James R. Douglass, and Todd 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, pages 103–110, New York, NY, USA, 2008. ACM.

[Wij69] A. van Wijngaarden. *Report on the Algorithmic Language ALGOL 68*. Printing by the Mathematisch Centrum, 1969.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques, 1992.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.

[WP07] Alessandro Warth and Ian Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA, 2007. ACM.

[You67] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189 – 208, 1967.

[Yu97] Sheng Yu. *Handbook of Formal Languages: Volume 1 Word, Language, Grammar*, chapter Regular Languages, pages 41–110. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[Zay14] Vadim Zaytsev. Formal foundations for semi-parsing. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 313–317, February 2014.

A
Formal development of PEGs

In section 3.1 we briefly introduced PEGs. In this chapter we recapitulate the formal definition of PEGs, their semantics and which expressions are syntactic sugar. We also recapitulate the abstract simulation of a PEG expression and define the first set for PEGs.

Definition A.1. (Parsing Expression Grammar (PEG)) A parsing expression grammar is a 4-tuple $G = (N, \Sigma, R, e_s)$ where N is a finite set of nonterminals, Σ is a finite set of terminal symbols, R is a finite set of rules, e_s is a starting expression.

Each rule $r \in R$ is a pair (A, e) which we write $A \leftarrow e$, $A \in N$ and e is a parsing expression. Parsing expressions (PEs) are defined inductively: if e_1 and e_2 are parsing expressions, then so are the following:

- ϵ , an empty string
- $'\mathtt{t}^+'$, any literal, $t \in \Sigma$
- $[\mathtt{t}^+]$, any character class, $t \in \Sigma$
- $\mathtt{A}$, any nonterminal, $A \in N$
- $\mathtt{e}?$, an optional expression
- $\mathtt{e_1 e_2}$, a sequence
- $\mathtt{e_1}/\mathtt{e_2}$, a prioritized choice
- $\mathtt{e}*$, aa zero-or-more repetitions
- $!\mathtt{e}$, a not-predicate
- $\&\mathtt{e}$, an and-predicate

□

Definition A.2 (PEG Semantics)**.** To formalize the semantics of a grammar $G = (N, \Sigma, R, e_s)$, we define a relation $\Rightarrow$ from pairs of the form (e, x) to the output pairs (o, y), where e is a parsing expression, $x \in \Sigma^*$ is an input string to be recognized, o indicates the result of a recognition attempt, and $y \in \Sigma^*$ is a remainder of input. The distinguished symbol f indicates failure.

Empty:

$$\frac{}{(\epsilon, x) \Rightarrow (\epsilon, x)}$$

Terminal *(success):*

$$\frac{a \in \Sigma}{(a, ax) \Rightarrow (a, x)}$$

Terminal *(failure):*

$$\frac{a \neq b}{(a, bx) \Rightarrow (f, bx)}$$

Nonterminal:

$$\frac{A \leftarrow e \in R \qquad (e, x) \Rightarrow (o, y)}{(A, x) \Rightarrow (o, y)}$$

Sequence *(success case):*

$$\frac{(e_1, x) \Rightarrow (o_1, y_1) \qquad (e_2, y_1) \Rightarrow (o_2, y_2)}{(e_1 e_2, x) \Rightarrow (o_1 o_2, y_2)}$$

Sequence *(failure 1):*

$$\frac{(e_1, x) \Rightarrow (f, x)}{(e_1 e_2, x) \Rightarrow (f, x)}$$

Sequence *(failure 2):*

$$\frac{(e_1, x) \Rightarrow (o, y) \qquad (e_2, y) \Rightarrow (f, y)}{(e_1 e_2, x) \Rightarrow (f, x)}$$

Choice *(option 1):*

$$\frac{(e_1, x) \Rightarrow (o, y)}{(e_1 / e_2, x) \Rightarrow (o, y)}$$

Choice *(option 2):*

$$\frac{(e_1, x) \Rightarrow (f, x) \qquad (e_2, x) \Rightarrow (o, y)}{(e_1 / e_2, x) \Rightarrow (o, y)}$$

Repetitions *(repetition):*

$$\frac{(e, x) \Rightarrow (o_1, y_1) \qquad (e*, y_1) \Rightarrow (o_2, y_2)}{(e*, x) \Rightarrow (o_1 o_2, y_2)}$$

Repetitions *(termination):*

$$\frac{(e, x) \Rightarrow (f, x)}{(e*, x) \Rightarrow (\epsilon, x)}$$

Not predicate *(success):*

$$\frac{(e, x) \Rightarrow (o, y)}{(!e, x) \Rightarrow (f, x)}$$

Not predicate *(failure):*

$$\frac{(e, x) \Rightarrow (f, x)}{(!e, x) \Rightarrow (\epsilon, x)}$$

□

Syntactic sugar The following expressions are syntactic sugar and can be expressed as follows [For04]: (i) an optional expression `e?` is equivalent to `e/ϵ`; (ii) one or more repetitions `e+` is equivalent to `ee*`; (iii) a character class is equivalent to a choice of one-character literals `'a'/'b'/'c'/...`; and (iv) an and-predicate `&e` is equivalent to a double negation `!(!e)`

Reductions of repetitions As in CFGs, repetition expressions can be eliminated from a PEG by converting them into right-recursive nonterminals [For04]. If expression `e` in repetiton `e*` accepts `ϵ`, the nonterminal is left-recursive and cannot be handled by PEGs in general.

Parsing Expression Languages Expression e *accepts* a string x if $\exists(e, x) \Rightarrow^+ (o, \epsilon)$. Expression e *succeeds* on a string xy if $\exists(e, xy) \Rightarrow^+ (o, y)$.

Definition A.3. (Parsing Expressions Languages (PELs)) A parsing expression language $\mathcal{L}(e)$ of a parsing expression e over the alphabet Σ is the set of strings $x \in \Sigma^*$ for which the e succeeds on x:

$$\mathcal{L}(e) = \{xy \mid (e, xy) \Rightarrow^+ (o, y)\}$$

□

Note that in this definition e does not necessarily consume all of xy, since even partially consumed strings are in the language. For example a language for a trivial expression `'a'` contains all the strings with `'a'` as a prefix (*e.g.*, `'aloha'`).

We use the following parent relation to compute the static next set (see Definition 5.8) used in bounded seas:

Definition A.4 (Parent Relation)**.** The parent function $\mathcal{P}(e, e')$ is a function that returns `true` if e' was directly constructed from e by applying one of the steps in Definition A.1, `false` otherwise. □

As an example of a parent relation, consider the following grammar snippet where `defn` is a parent of `'def'`, `id` and `body`:

```
defn ← 'def' id body
```

The following first set analysis is used to optimize choices of parsing expressions (see subsection 6.3.2):

Definition A.5 (First Set)**.** We define the first set *FIRST(e)* of an expression e as a set of expressions such that:

Nonterminal

$$\frac{a \in \Sigma}{\textit{FIRST}(a) = \{a\}}$$

Empty String

$$\frac{}{\textit{FIRST}(\epsilon) = \{\epsilon\}}$$

Sequence *(case 1):*

$$\frac{\epsilon \notin \mathit{FIRST}(e_1)}{\mathit{FIRST}(e_1 e_2) = \mathit{FIRST}(e_1)}$$

Sequence *(case 2):*

$$\frac{\epsilon \in \mathit{FIRST}(e_1)}{\mathit{FIRST}(e_1 e_2) = \mathit{FIRST}(e_1) \cup \mathit{FIRST}(e_2)}$$

Choice

$$\frac{}{\mathit{FIRST}(e_1/e_2) = \mathit{FIRST}(e_1) \cup \mathit{FIRST}(e_2)}$$

Repetition

$$\frac{}{\mathit{FIRST}(e*) = \mathit{FIRST}(e) \cup \{\epsilon\}}$$

Not Predicate *case 1*

$$\frac{\epsilon \notin \mathit{FIRST}(e)}{\mathit{FIRST}(!e) = \{!f_1\ !f_2\ \ldots\ !f_n \mid f_i \in \mathit{FIRST}(e)\}}$$

Not Predicate *case 2*

$$\frac{\epsilon \in \mathit{FIRST}(e)}{\mathit{FIRST}(!e) = \{\}}$$

□

Definition A.6 (Abstract Simulation)**.** We define a relation $\rightharpoonup$ consisting of pairs (e, o), where e is an expression and $o \in \{0, 1, f\}$. If $e \rightharpoonup 0$, then e can succeed on some input string while consuming no input. If $e \rightharpoonup 1$, then e can succeed on some input string while consuming at least one terminal. If $e \rightharpoonup f$, then e may fail on some input string. We will use variable s to represent an abstract result either 0 or 1. We will define the simulation relation $\rightharpoonup$ as follows:

1. $\epsilon \rightharpoonup 0$.
2. (a) $t \rightharpoonup 1, t \in T$.
   (b) $t \rightharpoonup f, t \in T$.
3. $A \rightharpoonup o$ if $e \rightharpoonup o$ and $A \leftarrow e$ is a rule of the grammar G.
4. (a) $e_1e_2 \rightharpoonup 0$ if $e_1 \rightharpoonup 0$ and $e_2 \rightharpoonup 0$.
   (b) $e_1e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup 1$ and $e_2 \rightharpoonup s$.
   (c) $e_1e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup s$ and $e_2 \rightharpoonup 1$.
   (d) $e_1e_2 \rightharpoonup f$ if $e_1 \rightharpoonup f$
   (e) $e_1e_2 \rightharpoonup f$ if $e_1 \rightharpoonup s$ and $e_2 \rightharpoonup f$.
5. (a) $e_1/e_2 \rightharpoonup 0$ if $e_1 \rightharpoonup 0$
   (b) $e_1/e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup 1$
   (c) $e_1/e_2 \rightharpoonup o$ if $e_1 \rightharpoonup f$ and $e_2 \rightharpoonup o$.
6. (a) $e* \rightharpoonup 1$ if $e \rightharpoonup 1$
   (b) $e* \rightharpoonup 0$ if $e \rightharpoonup f$
7. (a) $!e \rightharpoonup f$ if $e \rightharpoonup s$

(b) $!e \rightharpoonup 0$ if $e \rightharpoonup f$

□

For example, consider an abstract simulation of the following expressions:

```
'a' / ϵ
'b' 'c'
```

Because of the recursive nature of the definition, we first compute $\rightharpoonup$ for terminals and later we infer $\rightharpoonup$ for more complex expressions:

`'a'`	$\rightharpoonup 1$	(rule 2a), same for `'b'` and `'c'`
`'a'`	$\rightharpoonup f$	(rule 2b), same for `'b'` and `'c'`
`ϵ`	$\rightharpoonup 0$	(rule 1)
`'a'/ϵ`	$\rightharpoonup 0$	(rule 5b)
`'a'/ϵ`	$\rightharpoonup 1$	(rule 5c)
`'b' 'c'`	$\rightharpoonup 1$	(rule 4b)
`'b' 'c'`	$\rightharpoonup f$	(rule 4d)
`'a'*`	$\rightharpoonup 1$	(rule 6a)
`'a'*`	$\rightharpoonup 0$	(rule 6b)

Nullable and accepts epsilon analyses are utilized in a parser compiler to reduce overhead of backtracking. There might be different definitions of nullability [Bac79, FM13]; we define nullability with the help of an abstract simulation:

Definition A.7 (Nullable Expression)**.** We call an expression e *nullable*, if

$$e \rightharpoonup 0 \wedge e \not\rightharpoonup f$$

□

In other words, e is nullable if it can succeed on some input string while consuming no input and cannot fail. For example, zero or more repetitions of a Ruby class (`class*`) is a nullable expression. A repetition accepts an empty string (zero repetitions are allowed) and it never fails (see Definition A.2).

Definition A.8 (Accepts Epsilon)**.** We say that an expression e *accepts epsilon*, if

$$e \rightharpoonup 0$$

□

In other words, e accepts epsilon if it can succeed on some input string while consuming no input. For example, start of a line `^` is accepts epsilon. If invoked in start of a line position, it succeeds while consuming no input, but it can also fail when in other positions.

B
Bounded Seas Examples

In this chapter we provide detailed examples of definitions introduced in chapter 5.

B.1 Example of Dynamic *NEXT* computation

Let us compute the dynamic *NEXT* (see Definition 5.7) of the $\sim$`method`$\sim$ expression defined in the bounded sea grammar in Listing 5.5. Let us suppose a parser has aplready consumed `'class Foo'` in the input `'class Foo end'`. The invocation stack S_I looks as in Figure B.1.

3	($\sim$`method`$\sim$,9)
2	($\sim$`method`$\sim$ $+$ / $\sim$`nil`$\sim$,9)
1	(`classBody` $\leftarrow$ $\sim$`method`$\sim$ $+$ / $\sim\epsilon\sim$,9)
0	(`'class'` `id` `classBody` `'end'` ,0)

Figure B.1: State of a stack after parsing `'class Foo'` in the input `'class Foo end'`.

To parse $\sim$`method`$\sim$ we compute the dynamic next ($\mathcal{N}_D$) of $\sim$`method`$\sim$. We do this in the following steps:

1. First, $\mathcal{N}_D(S_I)$ is called.
2. This results in the repetition case as in Figure B.2. This case adds $\sim$`method`$\sim$ into the set and invokes $\mathcal{N}_D$ recursively on the remainder of S_I.
3. The new call results in the choice case as in Figure B.3, which again invokes the $\mathcal{N}_D$ function.

4. This results in the sequence case as in Figure B.5, which adds `'end'` into the set and terminates.

5. In the end, the complete $\mathcal{N}_D(\,\sim\!\texttt{method}\!\sim\,) = \{\,\sim\!\texttt{method}\!\sim\ \texttt{'end'}\,\}$.

Repetition (case 1):

$$\begin{array}{lcl}
1. & S = & \left|\begin{array}{rc}
{\scriptstyle 4} & (\ \sim\!\texttt{method}\!\sim\ ,9) \\
{\scriptstyle 3} & (\ \sim\!\texttt{method}\!\sim +\ ,9) \\
{\scriptstyle 2} & (\ \sim\!\texttt{method}\!\sim\ +\,/\ \sim\!\texttt{nil}\!\sim\ ,9) \\
{\scriptstyle 1} & (\ \texttt{classBody} \leftarrow\ \sim\!\texttt{method}\!\sim\ +\,/\ \sim\!\epsilon\!\sim\ ,9) \\
{\scriptstyle 0} & (\ \texttt{'class' id classBody 'end'}\ ,0)
\end{array}\right| \\
2. & S' = & \left|\begin{array}{rc}
{\scriptstyle 2} & (\ \texttt{classBody} \leftarrow\ \sim\!\texttt{method}\!\sim\ +\,/\ \sim\!\epsilon\!\sim\ ,9) \\
{\scriptstyle 1} & (\ \texttt{classBody}\ ,9) \\
{\scriptstyle 0} & (\ \texttt{'class' id classBody} * \texttt{'end'}\ ,0)
\end{array}\right| \\
3. & & \texttt{method} \rightharpoonup \{1, f\} \\
\hline
\multicolumn{3}{c}{\mathcal{N}_D(S) = \{\,\sim\!\texttt{method}\!\sim\,\} \cup \mathcal{N}_D(((\ \sim\!\texttt{method}\!\sim +\ , 9) : S'))}
\end{array}$$

Figure B.2: The repetition (case 1) inference when computing dynamic *NEXT*.

Choice (option 1):

$$\begin{array}{lcl}
1. & S = & \left|\begin{array}{rc}
{\scriptstyle 3} & (\ \sim\!\texttt{method}\!\sim +\ ,9) \\
{\scriptstyle 2} & (\ \texttt{classBody} \leftarrow\ \sim\!\texttt{method}\!\sim\ +\,/\ \sim\!\epsilon\!\sim\ ,9) \\
{\scriptstyle 1} & (\ \texttt{classBody}\ ,9) \\
{\scriptstyle 0} & (\ \texttt{'class' id classBody 'end'}\ ,0)
\end{array}\right| \\
2. & S' = & \left|\begin{array}{rc}
{\scriptstyle 1} & (\ \texttt{classBody} \leftarrow\ \sim\!\texttt{method}\!\sim\ +\,/\ \sim\!\epsilon\!\sim\ ,9) \\
{\scriptstyle 0} & (\ \texttt{'class' id classBody 'end'}\ ,0)
\end{array}\right| \\
\hline
\multicolumn{3}{c}{\mathcal{N}_D(S) = \mathcal{N}_D(((\ \sim\!\texttt{method}\!\sim\ +\,/\ \sim\!\texttt{nil}\!\sim\ , 9) : S'))}
\end{array}$$

Figure B.3: The choice (option 1) inference when computing dynamic *NEXT*.

B.2 Example of Static *NEXT* computation

Let us compute the static next (see Definition 5.8) of the `~method~` expression defined in the bounded sea grammar in Listing 5.5. The set of all the grammar rules PE

Nonterminal:

$$\begin{array}{lcc}
1. & S = & \left|\begin{array}{rc} {}_{2} & (\ \sim\texttt{method}\sim\ +/\sim\epsilon\sim\ ,9) \\ {}_{1} & (\ \texttt{classBody} \leftarrow\ \sim\texttt{method}\sim\ +/\sim\epsilon\sim\ ,9) \\ {}_{0} & (\ '\texttt{class}'\ \texttt{id}\ \texttt{classBody}\ '\texttt{end}'\ ,0) \end{array}\right| \\
2. & S' = & {}_{0}\left|(\ '\texttt{class}'\ \texttt{id}\ \texttt{classBody}\ '\texttt{end}'\ ,0)\right| \\
\hline
 & \multicolumn{2}{c}{\mathcal{N}_D(S) = \mathcal{N}_D(((\ \texttt{classBody}\ ,9) : S'))}
\end{array}$$

Figure B.4: The nonterminal inference when computing dynamic *NEXT*.

Sequence (case 1):

$$\begin{array}{lcc}
1. & S = & \left|\begin{array}{rc} {}_{1} & (\ \texttt{classBody}\ ,9) \\ {}_{0} & (\ '\texttt{class}'\ \texttt{id}\ \texttt{classBody}\ '\texttt{end}'\ ,0) \end{array}\right| \\
2. & & '\texttt{end}' \rightharpoonup \{1, f\} \\
\hline
 & \multicolumn{2}{c}{\mathcal{N}_D(S) = \{\ '\texttt{end}'\ \}}
\end{array}$$

Figure B.5: The sequence (case 1) inference when computing dynamic *NEXT*.

looks as in Figure B.6. To parse $\sim\texttt{method}\sim$ we compute the static next ($\mathcal{N}_S$) of

$$PE = \left\{\begin{array}{c}
\texttt{start} \leftarrow \texttt{class}* \\
\texttt{class} \leftarrow'\ \texttt{class}'\ \texttt{id}\ \texttt{classBody}\ '\texttt{end}' \\
'\texttt{class}'\ \texttt{id}\ \texttt{classBody}\ '\texttt{end}' \\
\texttt{classBody} \leftarrow\ \sim\texttt{method}\sim\ +/\sim\epsilon\sim \\
\sim\texttt{method}\sim\ +/\sim\epsilon\sim \\
\sim\texttt{method}\sim + \\
\texttt{method} \\
\texttt{method} \leftarrow'\ \texttt{def}'\ \texttt{id}\ \texttt{methodBody} \\
... \\
...
\end{array}\right\}$$

Figure B.6: Parsing expressions of the grammar from Listing 5.5.

$\sim\texttt{method}\sim$. We do this in the following steps:

1. First, $\mathcal{N}_S(PE,\ \sim\texttt{method}\sim)$ for all the parsing expressions PE is called.

2. This results in the *not a parent* case as in Figure B.7, because the `start` nonterminal in `start` $\leftarrow$ `class`$*$ is not a parent of `∼method∼`. This case invokes $\mathcal{N}_S$ on all the remaining expressions except for the first one.

3. The *not a parent* case is repeated until `∼method∼` $+$ is selected from the set of parsing expressions as in Figure B.8. This is the repetition case, which adds `∼method∼` to the set and merges it with $\mathcal{N}_S$ of `∼method∼` on the remaining expressions and with $\mathcal{N}_S$ of `∼method∼` $+$ $\mathcal{N}_S(PE,$ `∼method∼` $+$).

4. The rest of the $\mathcal{N}_S$ computation of `∼method∼` ends up in the *not a parent* case, because there are no more parent expressions of `∼method∼`.

5. The $\mathcal{N}_S(PE,$ `∼method∼` $+$) consumes expressions from PE until it finds `∼method∼` $+ / \epsilon$, the only parent of `∼method∼` $+$.

6. Based on the choice (option 1) a $\mathcal{N}_S(PE,$ `∼method∼` $+ / \epsilon$) is invoked.

7. Based on the nonterminal case a $\mathcal{N}_S(PE,$ `classBody`) is invoked.

8. Eventually, $\mathcal{N}_S$ for `classBody` ends in the sequence case as in Figure B.9. This case adds `'end'` into the set and finishes.

9. In the end, the complete $\mathcal{N}_S($ `∼method∼` $, PE) =$ { `∼method∼` `'end'` }

Not a parent

1. $$S = \left\{ \begin{array}{c} \texttt{start} \leftarrow \texttt{class}* \\ \texttt{class} \leftarrow' \texttt{class}' \; \texttt{id} \; \texttt{classBody} \; '\texttt{end}' \\ '\texttt{class}' \; \texttt{id} \; \texttt{classBody} \; '\texttt{end}' \\ \texttt{classBody} \leftarrow \; \sim\!\texttt{method}\!\sim \; + \, / \, \sim\!\epsilon\!\sim \\ \sim\!\texttt{method}\!\sim \; + \, / \, \sim\!\epsilon\!\sim \\ \sim\!\texttt{method}\!\sim \; + \\ \texttt{method} \\ \texttt{method} \leftarrow' \texttt{def}' \; \texttt{id} \; \texttt{methodBody} \\ \ldots \end{array} \right\}$$

2. $$S' = \left\{ \begin{array}{c} \texttt{class} \leftarrow' \texttt{class}' \; \texttt{id} \; \texttt{classBody} \; '\texttt{end}' \\ '\texttt{class}' \; \texttt{id} \; \texttt{classBody} \; '\texttt{end}' \\ \texttt{classBody} \leftarrow \; \sim\!\texttt{method}\!\sim \; + \, / \, \sim\!\epsilon\!\sim \\ \sim\!\texttt{method}\!\sim \; + \, / \, \sim\!\epsilon\!\sim \\ \sim\!\texttt{method}\!\sim \; + \\ \texttt{method} \\ \texttt{method} \leftarrow' \texttt{def}' \; \texttt{id} \; \texttt{methodBody} \\ \ldots \end{array} \right\}$$

3.

$$\mathcal{N}_S(\sim\!\texttt{method}\!\sim , S) = \mathcal{N}_S(\sim\!\texttt{method}\!\sim , S')$$

Figure B.7: The *not a parent* inference rule when computing static *NEXT*.

Repetition (case 1):

1. $$\text{S=} \left\{ \begin{array}{c} \sim\!\texttt{method}\!\sim \; + \\ \texttt{method} \\ \texttt{method} \leftarrow' \texttt{def}' \; \texttt{id} \; \texttt{methodBody} \\ \ldots \end{array} \right\}$$

2. $$\text{S'=} \left\{ \begin{array}{c} \texttt{method} \\ \texttt{method} \leftarrow' \texttt{def}' \; \texttt{id} \; \texttt{methodBody} \\ \ldots \end{array} \right\}$$

$$\mathcal{N}_S(\sim\!\texttt{method}\!\sim , S) =$$
$$\{ \sim\!\texttt{method}\!\sim \} \cup \mathcal{N}_S(\sim\!\texttt{method}\!\sim , S') \cup \mathcal{N}_S(\sim\!\texttt{method}\!\sim \; + , PE)$$

Figure B.8: The repetition (case 1) inference rule when computing static *NEXT*.

Sequence (case 1):

1. $$\text{S=}\left\{\begin{array}{c} '\texttt{class}'\ \texttt{id}\ \texttt{classBody}\ '\texttt{end}' \\ \texttt{classBody} \leftarrow\ \sim\!\texttt{method}\!\sim\ + /\ \sim\!\epsilon\!\sim \\ \texttt{method} \leftarrow'\ \texttt{def}'\ \texttt{id}\ \texttt{methodBody} \\ \texttt{methodBody} \leftarrow \texttt{blockBody}\ '\texttt{end}' \\ \ldots \end{array}\right\}$$

2. $$\text{S'=}\left\{\begin{array}{c} \texttt{classBody} \leftarrow\ \sim\!\texttt{method}\!\sim\ + /\ \sim\!\epsilon\!\sim \\ \texttt{method} \leftarrow'\ \texttt{def}'\ \texttt{id}\ \texttt{methodBody} \\ \texttt{methodBody} \leftarrow \texttt{blockBody}\ '\texttt{end}' \\ \ldots \end{array}\right\}$$

$$\mathcal{N}_S(\ \texttt{classBody}\ , S) = \{\ '\texttt{end}'\ \} \cup \mathcal{N}_S(\ \texttt{classBody}\ , S')$$

Figure B.9: The sequence (case 1) inference rule when computing static *NEXT*.

B.3 Overlapping Seas Example

Let us go through the overlapping seas problem for the following grammar:

```
S ← ~a~ ~b~
```

using `'..a..b..'` as input. The nonterminal expansion of `S` is in Figure B.10. In the example we use the following notation: input to the left, output to the right, the invocation stack S_I in the middle, the parsing expression being interpreted is on the top of S_I. For clarity reasons the rest of a parsing context is omitted. The invocation stack S_I is initialized with $(S, 0)$.

The whole result is `'..a..b..'`, because it is the result of the nonterminal expansion `S ← ~a~ ~b~`. The sequence on the top is straightforward, `~a~` consumes `'..a..'` and `~b~` consumes `'b..'`. The result is then `'..a..b..'` (see Figure B.11).

Nonterminal:

1. $S \leftarrow$ `~a~ ~b~` $\in R$

2. `..a..b..` | 1 (`~a~ ~b~` ,0) ; 0 (`S` ,0) | ⇒ $o =$ `..a..b..`

`..a..b..` | 0 (`S` ,0) | ⇒ $o =$ `..a..b..`

Figure B.10: The inference rule for nonterminal.

Sequence (success):

1. `..a..b..` | 2 (`~a~` ,0) ; 1 (`~a~ ~b~` ,0) ; 0 (`S` ,0) | ⇒ $x_1 =$ `..a..`

2. `b..` | 2 (`~b~` ,5) ; 1 (`~a~ ~b~` ,0) ; 0 (`S` ,0) | ⇒ $o_2 =$ `b..`

`..a..b..` | 1 (`~a~ ~b~` ,0) ; 0 (`S` ,0) | ⇒ $o_1 o_2 =$ `..a..b..`

Figure B.11: The inference rule for sequence.

In order to get a result of `~a~` invoked in the position 0, we first follow Definition 5.2 (see Figure B.12). It is a sequence of three parsers (generalization from the sequence of two to the sequence of three is straightforward). In Figure B.13 we see that before-water consumes `'..'`, the island itself consumes the desired `'a'` and another `'..'` is consumed by after-water.

Rewrite according to the Definition 5.2

1. `..a..b..`

2 | (`∼a∼` ,0)
1 | (`∼a∼ ∼b∼` ,0)
0 | (`S` ,0)

⇒ $o =$ `..a..`

`..a..b..`

2 | (`≈ (a/NEXT( ∼a∼ ))a ≈ (NEXT( ∼a∼ ))` ,0)
1 | (`∼a∼ ∼b∼` ,0)
0 | (`S` ,0)

⇒ $o =$ `..a..`

Figure B.12: Rewrite rule according to the Definition 5.2.

Sequence (success case):

1. `..a..b..`

3 | (`≈ (a/NEXT( ∼a∼ ))` ,0)
2 | (`≈ (a/NEXT( ∼a∼ ))a ≈ (NEXT( ∼a∼ ))` ,0)
1 | (`∼a∼ ∼b∼` ,0)
0 | (`S` ,0)

⇒ $o_1 =$ `..`

2. `a..b..`

3 | (`a` ,2)
2 | (`≈ (a/NEXT( ∼a∼ ))a ≈ (NEXT( ∼a∼ ))` ,0)
1 | (`∼a∼ ∼b∼` ,0)
0 | (`S` ,0)

⇒ $o_2 =$ `a`

3. `..b..`

3 | (`≈ (NEXT( ∼a∼ ))` ,3)
2 | (`≈ (a/NEXT( ∼a∼ ))a ≈ (NEXT( ∼a∼ ))` ,0)
1 | (`∼a∼ ∼b∼` ,0)
0 | (`S` ,0)

⇒ $o_3 =$ `..`

`..a..b..`

2 | (`≈ (a/NEXT( ∼a∼ ))a ≈ (NEXT( ∼a∼ ))` ,0)
1 | (`∼a∼ ∼b∼` ,0)
0 | (`S` ,0)

⇒ $o_1o_2o_3 =$ `..a..`

Figure B.13: The inference rule for sequence.

Let us investigate what happens in before-water of `∼a∼`. First of all, we need to determine *NEXT* (`∼a∼`). In this case it is { `∼b∼` }. Once `∼a∼` knows its boundary, before-water tries to find the island `a` or its boundary `∼b∼` in positions 0 and 1 until it finds the island in the position 2 (see Figure B.14). Before-water returns a substring from all the positions a boundary or an island failed, *i.e.*, `'..'`

Water (boundary):

1.	`..a..b..`	4	(`a/ ∼b∼` ,0)	⇒	f
		3	(`≈ (a/NEXT( ∼a∼ ))` ,0)		
		2	(`≈ (a/NEXT( ∼a∼ ))a ≈ (NEXT( ∼a∼ ))` ,0)		
		1	(`∼a∼ ∼b∼` ,0)		
		0	(`S` ,0)		
2.	`.a..b..`	5	(`a/ ∼b∼` ,1)	⇒	f
		4	(`≈ (a/NEXT( ∼a∼ ))` ,1)		
		3	(`≈ (a/NEXT( ∼a∼ ))` ,0)		
		2	(`≈ (a/NEXT( ∼a∼ ))a ≈ (NEXT( ∼a∼ ))` ,0)		
		1	(`∼a∼ ∼b∼` ,0)		
		0	(`S` ,0)		
3.	`a..b..`	6	(`a / ∼b∼` ,2)	⇒	`a` $\neq f$
		5	(`≈ (a/NEXT( ∼a∼ ))` ,2)		
		4	(`≈ (a/NEXT( ∼a∼ ))` ,1)		
		3	(`≈ (a/NEXT( ∼a∼ ))` ,0)		
		2	(`≈ (a/NEXT( ∼a∼ ))a ≈ (NEXT( ∼a∼ ))` ,0)		
		1	(`∼a∼ ∼b∼` ,0)		
		0	(`S` ,0)		

`..a..b..`	3	(`≈ (a/NEXT( ∼a∼ ))` ,0)	⇒	$x =$ `..`
	2	(`≈ (a/NEXT( ∼a∼ ))a ≈ (NEXT( ∼a∼ ))` ,0)		
	1	(`∼a∼ ∼b∼` ,0)		
	0	(`S` ,0)		

Figure B.14: The inference rule for water.

Overlapping Seas The interesting question is, why does `∼b∼` fail in the position 0 when invoked from the water of `∼a∼` as can be seen in the first case of Figure B.14? We have already explained the problem with overlapping seas in subsection 5.2.2 and now we show the computation formally. First of all we expand the choice and rewrite the sea on the top of the stack according to Definition 5.2. The new sequence on the top of the stack fails because before-water returns ϵ and there is no `b` in the position 0 (see Figure B.15).

Sequence (failure case):

$$1.\quad \texttt{..a..b..}\quad \begin{array}{r|c|} 6 & (\ \approx(\texttt{b/NEXT(}\ \sim\texttt{b}\sim\ \texttt{))}\ ,0) \\ 5 & (\ \approx(\texttt{b/NEXT(}\ \sim\texttt{b}\sim\ \texttt{))b}\approx(\texttt{NEXT(}\ \sim\texttt{b}\sim\ \texttt{))}\ ,0) \\ 4 & (\ \texttt{a/}\ \sim\texttt{b}\sim\ ,0) \\ 3 & (\ \approx(\texttt{a/NEXT(}\ \sim\texttt{a}\sim\ \texttt{))}\ ,0) \\ 2 & (\ \approx(\texttt{a/NEXT(}\ \sim\texttt{a}\sim\ \texttt{))a}\approx(\texttt{NEXT(}\ \sim\texttt{a}\sim\ \texttt{))}\ ,0) \\ 1 & (\ \sim\texttt{a}\sim\ \sim\texttt{b}\sim\ ,0) \\ 0 & (\ \texttt{S}\ ,0) \end{array} \quad\Rightarrow\quad \epsilon$$

$$2.\quad \texttt{..a..b..}\quad \begin{array}{r|c|} 6 & (\ \texttt{b}\ ,0) \\ 5 & (\ \approx(\texttt{b/NEXT(}\ \sim\texttt{b}\sim\ \texttt{))b}\approx(\texttt{NEXT(}\ \sim\texttt{b}\sim\ \texttt{))}\ ,0) \\ 4 & (\ \texttt{a/}\ \sim\texttt{b}\sim\ ,0) \\ 3 & (\ \approx(\texttt{a/NEXT(}\ \sim\texttt{a}\sim\ \texttt{))}\ ,0) \\ 2 & (\ \approx(\texttt{a/NEXT(}\ \sim\texttt{a}\sim\ \texttt{))a}\approx(\texttt{NEXT(}\ \sim\texttt{a}\sim\ \texttt{))}\ ,0) \\ 1 & (\ \sim\texttt{a}\sim\ \sim\texttt{b}\sim\ ,0) \\ 0 & (\ \texttt{S}\ ,0) \end{array} \quad\Rightarrow\quad f$$

$$\texttt{..a..b..}\quad \begin{array}{r|c|} 5 & (\ \approx(\texttt{b/NEXT(}\ \sim\texttt{b}\sim\ \texttt{))b}\approx(\texttt{NEXT(}\ \sim\texttt{b}\sim\ \texttt{))}\ ,0) \\ 4 & (\ \texttt{a/}\ \sim\texttt{b}\sim\ ,0) \\ 3 & (\ \approx(\texttt{a/NEXT(}\ \sim\texttt{a}\sim\ \texttt{))}\ ,0) \\ 2 & (\ \approx(\texttt{a/NEXT(}\ \sim\texttt{a}\sim\ \texttt{))a}\approx(\texttt{NEXT(}\ \sim\texttt{a}\sim\ \texttt{))}\ ,0) \\ 1 & (\ \sim\texttt{a}\sim\ \sim\texttt{b}\sim\ ,0) \\ 0 & (\ \texttt{S}\ ,0) \end{array} \quad\Rightarrow\quad f$$

Figure B.15: The inference rule for sequence (failure case).

The before-water of $\sim$b$\sim$ returns ϵ because of the overlapping seas case. It analyzes the invocation stack noticing the before-water of $\sim$a$\sim$ invoked in the position 0 (using the $\mathcal{SO}$ function from Definition 5.5) and returns ϵ (see Figure B.16).

If there is no overlapping seas case in the semantics of bounded seas, the before-water of $\sim$b$\sim$ consumes `'..a..'` contrary to the correct parse ϵ (see Figure B.16). This means that the before-water of $\sim$a$\sim$ (see Figure B.14) would be $x' = \epsilon$. This would then fail the whole $\sim$a$\sim$ and consequently the whole $\sim$a$\sim$ $\sim$b$\sim$.

Water (overlapping):

1. $$S_I = \left|\begin{array}{rc} {\scriptstyle 6} & (\ \approx(\texttt{b}/\texttt{NEXT}(\ \sim\texttt{b}\sim\ ))\ ,0) \\ {\scriptstyle 5} & (\ \approx(\texttt{b}/\texttt{NEXT}(\ \sim\texttt{b}\sim\ ))\texttt{a} \approx (\texttt{NEXT}(\ \sim\texttt{b}\sim\ ))\ ,0) \\ {\scriptstyle 4} & (\ \texttt{a}/\ \sim\texttt{b}\sim\ \ ,0) \\ {\scriptstyle 3} & (\ \approx(\texttt{a}/\texttt{NEXT}(\ \sim\texttt{a}\sim\ ))\ ,0) \\ {\scriptstyle 2} & (\ \approx(\texttt{a}/\texttt{NEXT}(\ \sim\texttt{a}\sim\ ))\texttt{a} \approx (\texttt{NEXT}(\ \sim\texttt{a}\sim\ ))\ ,0) \\ {\scriptstyle 1} & (\ \ \sim\texttt{a}\sim\ \ \sim\texttt{b}\sim\ \ ,0) \\ {\scriptstyle 0} & (\ \texttt{S}\ ,0) \end{array}\right|$$

2. $$\mathcal{SO}(S_I) = \texttt{true}$$

$$\texttt{..a..b..} \quad \left|\begin{array}{rc} {\scriptstyle 6} & (\ \approx(\texttt{b}/\texttt{NEXT}(\ \sim\texttt{b}\sim\ ))\ ,0) \\ {\scriptstyle 5} & (\ \approx(\texttt{b}/\texttt{NEXT}(\ \sim\texttt{b}\sim\ ))\texttt{a} \approx (\texttt{NEXT}(\ \sim\texttt{b}\sim\ ))\ ,0) \\ {\scriptstyle 4} & (\ \texttt{a}/\ \sim\texttt{b}\sim\ \ ,0) \\ {\scriptstyle 3} & (\ \approx(\texttt{a}/\texttt{NEXT}(\ \sim\texttt{a}\sim\ ))\ ,0) \\ {\scriptstyle 2} & (\ \approx(\texttt{a}/\texttt{NEXT}(\ \sim\texttt{a}\sim\ ))\texttt{a} \approx (\texttt{NEXT}(\ \sim\texttt{a}\sim\ ))\ ,0) \\ {\scriptstyle 1} & (\ \ \sim\texttt{a}\sim\ \ \sim\texttt{b}\sim\ \ ,0) \\ {\scriptstyle 0} & (\ \texttt{S}\ ,0) \end{array}\right| \quad \Rightarrow \quad \epsilon$$

Figure B.16: The inference rule for overlapping seas.

C Implementation

In this section we provide implementation of the combinators mentioned in this work to help the reader better understand the PetitParser internals and its functionality. Furthermore, we provide implementation of methods from `Context` that are called from the parser combinators.

Implementation of `Action` is in Listing C.1, implementation of `And` predicate is in Listing C.2, implementation of `CharClass` is in Listing C.3, implementation of `Choice` is in Listing C.4, implementation of `Literal` is in Listing C.5, implementation of `Sequence` of expressions is in Listing C.6, implementation of `Star`, *i.e.*, zero-or-more repetitions, is in Listing C.7, implementation of `Token` is in Listing C.8, and implementation of `Wrapping`, *i.e.*, an adapter from block closure to a combinator, is in Listing C.9.

```
Action>>parseOn: context
  | result |
  "evaluate the underlying combinator"
  result ← child parseOn: context.
  "return if failure"
  result isFailure ifTrue: [ ↑ result ]

  "evaluate block with result as an argument"
  ↑ block withArguments: result
```

Listing C.1: Implementation of `Action`.

C.1 Bounded seas

Even though a bounded sea consists of before-water, island and after-water, it has only one instance variable (see Listing C.10). The `parseOn:` method of a bounded sea

```
AndPredicate>>parseOn: context
  | memento |
  memento ← context remember.
  result ← parser parseOn: context.
  context restore: memento.
  ↑ result isPetitFailure ifTrue: [
    result
  ] ifFalse: [
    nil
  ]
```

Listing C.2: Implementation of `AndPredicate`.

```
CharClass>>parseOn: context
  (context atEnd not and:
   [predicate value: context peek]) ifTrue: [
    ↑ context next
  ] ifFalse: [
    ↑ PPFailure message: 'Predicate expected'
  ]
```

Listing C.3: Implementation of `CharClass`.

```
Choice>>parseOn: context
  | result |
  self children do: [:child |
    result ← child parseOn: context.
    result isPetitFailure ifFalse: [
      ↑ result
    ]
  ].
  ↑ result
```

Listing C.4: Implementation of `Choice`.

```
Literal>>parseOn: context
  | lookahead |
  lookahead ← (context next: literal size).
  (lookahead == literal) ifTrue: [
    ↑ lookeahead
  ] ifFalse: [
    context back: literal size
    ↑ Failure message: 'literal expected'
  ]
```

Listing C.5: Implementation of `Literal`.

```
Sequence>>parseOn: context
  | memento retval result |
  retval ← OrderedCollection new.
  "memoize"
  memento ← context remember.
  children do: [:child |
    "evaluate an underlying child"
    result ← child parseOn: context.
    "restore and return if failure"
    result isFailure ifTrue: [
      context restore: memento
      ↑ result
    ].
    retval add: result
  ].
  ↑ retval
```

Listing C.6: Implementation of `Sequence`.

```
Star>>parseOn: context
  | retval result |
  retval ← OrderedCollection new.
  [
    result ← child parseOn: context.
    result isPetitFailure
  ] whileFalse: [
    retval add: result
  ]
  ↑ retval
```

Listing C.7: Implementation of `Star`.

is in Listing C.11. The before and after water are computed automatically, as demonstrated in Listing C.12.

The *NEXT* set can be computed dynamically from the expression stack as demonstrated in Listing C.13 or statically, before a parse attempt. The static approach hooks into the interface method `parse:` (see section 3.2) and computes the *NEXT* set for all the sea expressions as demonstrated in Listing C.14.

The `Context` methods `invoked:`, `return:` and `fail:` are used to manage the expression stack. The overlapping seas are detected by comparison of positions of two top waters (see Listing C.16).

```
Token>>parseOn: context
  | memento result |
  memento ← context remember.
  whitespace parseOn: context.
  result ← parser parseOn: context.

  result isPetitFailure ifTrue: [
    context restore: memento
    result
  ].
  whitespace parseOn: context.

  ↑ Token new
    value: result flatten;
    start: memento position;
    end: context position
```

Listing C.8: Implementation of `Token`.

```
Wrapping>>parseOn: context
  ↑ blockClosure value: context
```

Listing C.9: Implementation of wrapping parser to provide `parseOn:` interface to block closures.

```
Parser subclass: #BoundedSea
    instanceVariables: ’island’.
```

Listing C.10: `BoundedSea` has only one instance variable `island`; before and after-water are created dynamically.

```
BoundedSea>>parseOn: context
    | result1 result2 result3 memento |
    context invoked: self.
    memento ← context remember.

    "Phase One"
    result1 ← self parseBeforeWater: context.
    result1 ifFailure: [
        ↑ context fail: 'boundary or island not found'
    ].

    "Phase Two"
    result2 ← island parse: context
    result2 ifFailure: [
        context restore: memento.
        ↑ context fail: 'island not found'
    ]

    "Phase Three"
    result3 ← self parseAfterWater: context.
    result3 ifFailure: [
        context restore: memento.
        ↑ context fail: 'boundary not found'
    ].

    ↑ context return: { resutl1 . result2 . result3 }
```

Listing C.11: Implementation of a `parse:` method in `BoundedSea`. The three phases correspond to the phases in Definition 5.1.

```
BoundedSea>>parseBeforeWater: context
    | next |
    "Catch the overlapping seas"
    context seasOverlap ifTrue: [
        ↑ nil
    ].

    ↑ self goUpTo: island / self next
```

Listing C.12: Implementation of a `beforeWater:` method in `BoundedSea`.

```
Context>>next
    | stack |
    stack ← self expressionStackFrom: thisContext.
    ↑ self next: stack into: Set new

Context>>next:stack into: set
    ...
    "Sequence (case 1):"
    (stack secondTop isSequence and: [
     stack secondTop firstChild == stack top ] and: [
     stack secondTop secondChild acceptsEpsilon not ]) ifTrue: [
        set add: stack secondTop secondChild.
        ↑ set
    ]
    ...
```

Listing C.13: Fragment of a `next` method in `Context`, which can be computed only in runtime.

```
Parser>>parse:input
    ...
    self allChildren select: [ :e | e isSea ]  thenDo: [ :sea |
        "compute next for each sea"
        sea nextIn: self allChildren into: Set new
    ]
    ...

Parser>>nextIn: parsers into: aSet
    | first withoutFirst |
    first ← parsers first.
    withoutFirst ← parsers removeFirst.

    ...
    "Sequence (case 1):"
    (first firstChild == self and: [
     first secondChild acceptsEpsilon not]) ifTrue: [
        aSet add: parser secondChild.
        ↑ self nextIn: withoutFirst into: aSet.
    ]
    ...
```

Listing C.14: A fragment from *NEXT* computation, which can be computed at any time.

```
Context>>invoked: parser
  self invokedPositions push: self position

Context>>return: result
  self invokedPositions pop.
  ↑ result

Context>>fail: message
  self invokedPositions pop.
  ↑ Failure message: message on: self
```

Listing C.15: Implementation of `invoked:`, `return:` and `fail:` methods in `Context`.

```
Context>>seasOverlap
  ↑ self invokedPositions top ==
    self invokedPositions secondTop
```

Listing C.16: Implementation of a `seasOverlap` method in `Context`.

D
Layout Sensitivity in the Wild

The first layout-sensitive language, ISWIM (If you See What I Mean), was proposed by Landin in 1966 [Lan66]. ISWIM introduced an offside rule in programming, inspired by the offside rule from soccer. Once the offside line is set, the code (just like a player) is not allowed to cross the offside line.[1] As with soccer, the trick is to know where the line is and not get ahead of it. Languages adjust the offside rule in different ways and we report on these approaches in the following text.

D.1 Haskell

The Haskell grammar[2] is defined in a context-free form. Haskell uses indentation to (i) delimit expressions; and (ii) determine borders of a group of expressions.[3] Haskell uses a special stage in the scanner-parser pipeline to fill in missing tokens that correspond to indentation.

The offside line is set by the first non-whitespace token after the keyword `let`, `where`, `do`, or `if` if the open brace is omitted. An expression aligned with the offside line is an expression that starts at the column set by the offside line. An alignment with the offside line can be used as an expression delimiter (instead of semicolon) in a group of expressions. A line starting further (a column greater than a column set by the offside line) is the continuation of a previous line. An empty line has no effect. A group of expressions ends when an expression appears *in the offside position* — on a column smaller than a column set by the offside line. Examples are shown in Listing D.1.

[1] We assume that only spaces are used. If tabulators are used, they are replaced by a predefined number of spaces.

[2] `http://www.webcitation.org/6k7g6DgOR`

[3] `http://www.webcitation.org/6k7g7EC0P`

```
do putStr "Hi"
   putStr
     "There"            -- Continuation

   if (x > 2) then do   -- blocks can start anywhere
putStr "> 2"            -- part of the then block
putStr "Hi There"       -- still in the then block
```

Listing D.1: Indentation rules of Haskell.

D.2 Python

The Python grammar[4] is also defined in a context-free form. It uses two special tokens to describe indentation: `indent` and `dedent`.[5] These two special tokens are inserted by a specialized lexer that uses a stack of indentation levels to emit `indent` and `dedent`.

Python uses indentation to (i) delimit expressions; (ii) determine block borders; and (iii) resolve ambiguity in `if`-`then`-`else` expressions. In Python, the offside line is set by the first non-whitespace character on a line. `indent` is emitted when the offside line of the current line is further right (on a greater column) than the offside line of the previous line. `dedent` is emitted when the offside line of the current line is closer to the left (on a smaller column) than the offside line of the previous line. After `dedent` the offside line must match the previous offside line (the offside lines must be on the same column).

Expressions can be delimited by placing them on separate lines without interleaving `indent` or `dedent`. The beginning of a block is signaled by `indent` and the end of a block is signaled by `dedent`. Blocks may contain empty lines. An `else` is matched to the closest `if` at the same indentation level. Listing D.2 shows examples.

```
if x > 2:
    print "> 2"    # indented
else:              # dedented
    print "< 2"    # indented


if x > 2:
    print "> 2"    # indented
 else:             # fail - else not aligned!
    print "< 2"

if x > 2: print "> 2"
          print x  # unexpected indent
```

Listing D.2: Indentation rules of Python.

[4] http://www.webcitation.org/6k633YacT
[5] http://www.webcitation.org/6k637RJ7V

D.3 F#

The F# grammar[6] is also defined in a context-free form. F# preprocesses the token stream to insert special tokens such as `$in`, `$done`, `$begin` and `$end`. F# uses indentation to (i) delimit expressions; (ii) determine block borders; and (iii) resolve ambiguity in `if`-`then` expressions.

An expression aligned with the offside line is an expression that starts at the column set by the offside line. An expression is in the *offside position* if it starts at a column less than the column set by the offside line. An expression is in the *onside position* if it starts at a column column equal to or greater than the column set by the offside line.

The offside line is set by the first non-whitespace token after the keywords `let`, `then`, `else`, `try`, `finally`, `do` and `=` in the `let` expression. Alignment with the offside line can be used as a delimiter between expressions (instead of semicolon). An empty line has no effect. A block is finished by the first expression in the offside position. An `else` token is related to the closest `if` with the `else` in onside position. Relevant examples are in Listing D.3.

```
let f =
    let x = 1   // this line sets
                // the offside line to 4
    let y = 2   // aligned, $in inserted
    x + y       // aligned
printfn "hi"    // expression in offside,
                // not part of let


if x > 2 then
    if x < 10 then
        printfn "2 < x < 10"
  else          // connected to
                // the first if
    printfn "x <= 2"
```

Listing D.3: Indentation rules of F#.

F# treats infix operators in a special way. Infix operators (such as `+` or `>>`) can be in the offside position by their length plus one space. They can also be in an arbitrary onside position, *i.e.*, they don't have to be aligned. Listing D.4 illustrates this feature.

```
let x =  1    // offside line set by 1
       + 2    // + a bit in offside position
              // x = 1 + 2

let x =  1    // offside line set by 1
   + 2        // fails, too much in offside
```

Listing D.4: Infix operators in F# can be in an offside position.

[6] http://www.webcitation.org/6k7gNo8HA

D.4 YAML

YAML is a data serialization language that aims to replace JSON and be more human-friendly. The YAML grammar[7] is defined using a parameterized BNF grammar, where one of the parameters specifies the indentation level. A parser of YAML uses the approach proposed by Brunauer [BM06] based on an extension of context-free grammars with counters.

YAML uses indentation (i) to determine structure of block styles;[8] (ii) to distinguish comments from block contents in trailing lines; and (iii) to insert flow styles into block styles. In YAML, indentation is defined as zero or more space characters at the start of a line. The `-`, `?` and `:` characters, which are used to denote block collection entries, are perceived by people to be part of the indentation and must be treated as indentation as well. In the input, the indentation parameter may be specified explicitly (Listing D.5, line 13), otherwise it is determined automatically using *auto-detect* function (Listing D.5, line 18). Indentation is irrelevant in flow styles, relevant in block styles.

```
1  # nested list ("a" ("b" "c"))
2   - a
3   - - b
4     - c
5
6  # flow style in block
7  # list with one element ("a - b - c")
8   - a
9     - b  # continuation of the previous line
10    - c
11
12 # string " a\n b"
13 |1      # explicit indentation
14   a     # " a", the second space is not part of indentation
15   b     # " b"
16
17 # string "a\nb"
18 |       # implicit indentation
19   a     # "a", auto-detect determines indentation to two
20   b     # "b"
```

Listing D.5: Indentation rules of YAML.

D.5 OCaml

OCaml[9] has a line-oriented preprocessor called “The Whitespace Thing” (TWT),[10] which uses indentation (i) to delimit expressions in a group; (ii) to determine borders

[7] http://www.webcitation.org/6k7gY6Qiz

[8] Contrary to the explicit indicators of flow styles.

[9] http://www.webcitation.org/6k7geqiDX

[10] http://www.webcitation.org/6k7gfm7lm

of a group; and (iii) to resolve ambiguity in nested `let`, `if`-`then`-else and `try`-`catch` expressions.

Alignment with a former expression on a previous line can be used as a delimiter (instead of a semicolon). A group of expressions ends when an expression is not aligned with the previous expressions. See examples in Listing D.6.

```
for  i = 1 to 10 do
    print-int i      (* semicolon ommited *)
    print-newline () (* still in the block *)
                     (* "done" not needed *)
print-string "done"
```

Listing D.6: Indentation rules of OCaml.

D.6 CoffeeScript

The CoffeeScript grammar[11] is defined in a context-free form. CoffeeScript uses indentation to (i) separate expressions; (ii) to determine block borders; (iii) to resolve ambiguity in `if`-`then`-`else` expressions; and (iv) to create objects without explicit curly braces.

The lexer of CoffeeScript[12] inserts the special tokens `indent` and `outdent` using a stack of indentation levels. Indentation in multi-line strings is ignored.

```
if (x > 2)
    if (x < 10)
        alert(2 < x < 10)
 else        // conected to
             // the first if
        alert (x <= 2)

kids =       // Curly braces ommited
  son:
    name: "Max"
    age:  11
  daughter:
    name: "Ida"
    age:  9
```

Listing D.7: Indentation rules of CoffeeScript.

D.7 Grace

The goal of the Grace programming language[13] is to integrate proven newer ideas in programming languages into a simple language for teaching. Grace does not use in-

[11] http://www.webcitation.org/6k7gnbzWr
[12] http://www.webcitation.org/6k7goQcOo
[13] http://www.webcitation.org/6k7gsNYsl

dentation to signal block boundaries. In Grace a line break followed by an increase in the indent level implies a line continuation, whereas line break followed by the next line at the same or lesser indentation implies a semicolon [Gra].

```
while {stream.hasNext} do {
    print (stream.read) // semicolon ommited
}                       // auto-inserted on dedent
```

Listing D.8: Indentation rules of Grace.

D.8 SRFI 49 — Indentation-Sensitive Scheme

This SRFI 49[14] is a *Scheme Request For Implementation* requesting for I-expressions (indentation-sensitive S-expressions) in Scheme [KRS08]. The syntax uses indentation to group expressions, and has no special cases for semantic constructs of the language. I-expressions can be used both for program and data input.

SRFI 49 uses `indent` and `dedent` tokens to replace the begin and end of S-expressions. `indent` is emitted when text on a new line starts at a column greater than that of the previous line. `dedent` is emitted when text on a new line starts at a column less than that of the previous line.

```
  if
    = x 0       ; condition
    1           ; then branch
    * x         ; else branch
      y
```

Listing D.9: SRFI 49 — Scheme With Indentation.

D.9 Elastic Tabstops

Elastic tabstops[15] offer an alternative way to handle tabstops. Rather than saying that a tab character places the text that follows it at the next N-th column (traditional fixed tabstops), a tab character is a delimiter between table "cells". The columns in the lines above and below the "cell" that is being changed are always kept aligned.[16]

As an example see Listing D.10 with a matrix definition, the alignment servers as a column delimiter. Space and tab characters are denoted respectively by `␣` and `→`. Note that the tab character at line 3 is smart enough to align itself to the previous line, no matter how far a number on the previous line is. Note also that the `123` and `456` on line 2 have an extra space between them in order to allow for `12345` from line 4 to fit in. In elastic tabstops there are more offside lines to which code is aligned to

[14] http://www.webcitation.org/6k7gxCzNE

[15] http://www.webcitation.org/6k7gxzfkg

[16] To easily understand this feature, we recommend to follow the previous link to a self-explanatory graphical example.

(*e.g.*, in our example there are three separate offside lines, one for each column of the matrix).

```
matrix ←
␣␣123→   456→789
␣␣1→     4→   7
␣␣12345→ 4→   7890
␣␣123→    4→  7890 "Invalid row, 4 not aligned"
```

Listing D.10: Example of Elastic Tabstops, the numbers must be aligned.

E
Scanner

In this chapter we describe how to integrate a scanner into a PEG-based parser. Furthermore we inspect (i) regular parsing expressions; (ii) regular parsing languages, which can be described by regular parsing expressions; and (iii) finite state automata, which can recognize regular parsing languages.

E.1 Scanners in PEG-based parsers

The path from a PEG parser to a scanning PEG parser includes several steps: (i) the expressions representing tokens and the expressions recognizable by a scanning parser have to be identified; (ii) these expressions are then used to identify choices that can be optimized with the help of a finite state automaton; and finally (iii) the automaton is embedded into a scanner, which is consequently integrated into the parser.

E.1.1 Tokens and Scannable Parsing Expressions

Parsing expressions that describe regular parsing languages (RPELs) are regular parsing expressions (RPEs) described in Appendix E.2. RPEs are recognizable by deterministic finite state automata (DFSAs). Ordered choices of expressions in RPEs are expressed as unions and complements of RPELs. We provide the detailed description in Appendix E.3. The transformation of a RPE into a $DFSA$ is described in section E.4. A token is a regular parsing expression ($RPE \in \mathbb{RPE}$) preceded and succeeded by whitespace, *i.e.*, a regular parsing expression wrapped `token` operator:

$$\mathbb{RTPE} = \{ \texttt{RPE token} \mid \forall RPE \in \mathbb{RPE}\}$$

Parsing expressions that can be recognized by a scanning PEG-based parser are *scannable parsing expressions* (SPEs). We define a parsing expression e to be *scannable* if all its terminal subexpressions (empty string, literal and character class) are parsing subexpressions of some regular token parsing expression $RTPE \in \mathbb{RTPE}$. In other words e is scannable if it is made of tokens, *e.g.*, in the concrete case of PetitParser using the `token` operator.

E.1.2 Scannable Choices

Based on a static analysis of a grammar, we can identify choices that can be optimized with the help of a scanner. To describe this analysis, we use the following choice-related terminology:

For a choice expression $e = e_1/e_2$, e_1 and e_2 are *alternatives* of a choice e. For any scannable parsing expression ($SPE \in \mathbb{SPE}$) we can compute a first set, $first : SPEs \to \mathbb{P}(\mathbb{RTPE})$ (where $\mathbb{P}(\mathbb{RTPE})$ is the power set of all trimmed RPEs, *i.e.*, the set of all subsets) using the standard first set algorithm as described in Definition A.5.[1] Two expressions are *distinct* if there is no string s that is recognized by both of them. A *distinct expression set* is a set of expressions where every two expressions are distinct. And finally a *distinct scannable parsing expression ($DSPE$)* is a scannable parsing expression that has a distinct first set.

Each choice is treated separately and has its own customized scan method. This way the same string may represent different tokens depending on the actual rule being parsed.

Deterministic Choice In some cases, the first set of the whole choice is distinct. In such a case, the alternatives can be uniquely distinguished based on the next token. Such a choice can be implemented without any memoization or backtracking, using a PEG that is equivalent to LL(1) choice of LL(k) grammars [GJ08a]. A parsing expression $e = e_1/e_2$ is a *deterministic choice* if $first(e)$ is a distinct expression set.

Nondeterministic Choice In case the first set of a choice is not distinct, an alternative of a choice can still be guarded: an alternative is selected only if the next token is in the first set of that alternative. A guard is applied only if the first set of the alternative is distinct. The guard invokes a scanner, which uniquely determines the next token. The guard allows parser to enter the alternative only if a token from the first set of the alternative is found. If the first set of the option is not distinct, the guard cannot be applied and the alternative has to handled as traditional (*i.e.*, non-scanning) parsing expression.

Even if a guard returns `true` and an alternative is entered, it is not guaranteed that the alternative will succeed. PEGs are nondeterministic and it might happen that that a parser backtracks after an arbitrarily long prefix of an input is consumed. Therefore, even the scanner has to support backtracking to recover from an incorrectly chosen alternative.

E.1.3 Scanner

The scanner as used in the previous section has to provide the following functionality. It has to (i) provide guards for scannable parsing expressions; (ii) recognize a token from a distinct set of RPEs; (iii) memoize and backtrack; and (iv) consume input. In the following text, we describe this functionality in detail.

[1] The token expressions, *i.e.*, expressions in a form `etoken`, serves as the first set terminals.

Scanner Guards A guard for a scannable parsing expression SPE checks if the scanning function has been invoked, *i.e.*, if there is a value assigned to the current token. If so, it returns `true` when the current token is in $\mathit{first}(SPE)$. If no value is assigned to the current token,[2] the scanner invokes a scanning method to recognize which token from $\mathit{first}(SPE)$ is in the input and returns its result. An implementation of the guard `guard_privateOrClass` used to parse the grammar from Listing 6.1 is shown in Listing E.1.

```
Scanner>>guard_privateOrClass
    currentToken notNil ifTrue: [
        ↑ #(#'private' #'class') contains: currentToken
    ].
    ↑ #(#'private' #'class') contains: self scan_privateOrClass
```

Listing E.1: Implementation of `privateOrClass` guard.

Recognizing a token In case a token is not recognized, the scanning method itself has to recognize the next token. For a scannable parsing expression SPE the scanning method is defined as follows:

From $\mathit{first}(SPE) = \{t_1, t_2, ..., t_i\}, t_i \in \mathbb{RTPE}$ a deterministic automaton for a regular expression of unordered choice of first tokens $t_1|t_2|...|t_i$ is created and compiled into a method.[3]

A scan method determines the token based on the current first set. The result of a scan method is `true` if a token from the first set was found and `false` otherwise. If a token is found, as a side-effect, the scan method stores its value (*i.e.*, its representation in input, *e.g.*, `'var'`) and its unique identifier (*e.g.*, `#VAR` or `#ID`). If a token is not found, a distinguished value `#failure` is set as its identifier. An example is in Listing E.2.

Memoization and Backtracking The memoization method must store the state of a scanner. In contrast to a PEG parser, the state of a scanner is not only a position in a stream, but also the current token. The state of a scanner is a 4-tuple (s, p, t, v), where s is an input string, $p \in \mathbb{N}$ is the current position in input, t is the current token identifier, which is either `nil` (token undefined), *failure* (token not found), or t (token identified by t found) and v, which is the string value of the current token, if t is not `nil`.

As shown in subsection 6.3.2, the state is remembered before a recognition attempt of a non-deterministic choice and in case of failure, it is restored to its original value. Implementation of the remember and restore methods is shown in Listing E.3. In the PetitParser implementation, an instance of a scanner is created per input, therefore a scanner remembers only the triple (p, t, v)

Consuming a token Once a parser reaches a token expression, its scanner is used to consume input. As an example consider the `classToken` from the grammar in Listing 6.1 which is implemented as demonstrated in Listing E.4.

[2] This might happen at the very beginning, after the previous token has been consumed or after backtracking.

[3] Note that we use unordered choice, because our first algorithm returns an unordered set. Using ordered sets is a research path we would like to explore in future.

```
Scanner>>scan_privateOrClass
    ((char ← self next) == $p) ifTrue: [
        ((char ← self next) == $r) ifTrue: [
            "consumes the rest of the 'private' token"
            ...
            currentValue ← 'private'.
            currentToken ← #'private'
        ] ifFalse: [
            currentToken ← #'failure'
        ]
    ] ifFalse: [ (char == $c) ifTrue: [
        "consumes the rest of the 'class' token"
        ...
        currentValue ← 'class'.
        currentToken ← #'CLASS'
    ] ifFalse: [
        currentToken ← #'failure'
    ]].
    ↑ currentToken
```

Listing E.2: Implementation of the `scan_privateOrClass` scanning method, used in Listing E.1.

```
Scanner>>remember
    ↑ Array
        with: stream position.
        with: currentToken
        with: currentValue

Scanner>>restore: memento
    stream position: memento first.
    currentToken ← memento second.
    currentValue ← memento third.
```

Listing E.3: Implementation of the `remember` and `restore` methods.

```
ScanningParser>>classToken
    ↑ scanner consume_CLASS
```

Listing E.4: Implementation of `classToken`, used to parse the grammar from Listing 6.1.

When consuming a token, the current token identifier is set to `nil` (*i.e.*, the current token is unknown), whitespace is consumed and the token value is returned. Eventually, if the follow set is also distinct, a scanning method for the next set might be invoked.

```
Scanner>>consume_CLASS
    currentToken ← nil.
    self consumeWhitespace.
    ↑ currentValue
```

Listing E.5: Implementation of the `consume_CLASS` method.

E.2 Regular Parsing Expressions

The traditional parsing technology is built on top of context-free grammars (CFGs) and their subset, regular expressions (REs), which are equivalent to FSAs. In this work, we focus on a PEG parsing technology and consequently we introduce a subset of parsing expressions, regular parsing expressions, which are equivalent to FSAs as well.

However, there is a semantic gap between these two formalisms. PEGs use ordered choice contrary to unordered choice of CFGs. Furthermore, PELs are defined as a set of strings for which an expression *succeeds* (see Definition A.3), *i.e.*, all the strings which prefix is recognized by a given parsing expression. This is in contrast to CFLs, in which partially consumed strings are excluded from the language. Last but not least, there is no notion of failure in CFGs.

While ordered choice can be expressed using unordered choice and the difference of REs (as we describe in section E.3), to deal with partially consumed strings and failures would require to extend the FSA formalism.

In this work we do not extend FSAs and we adhere to standard automata because they provide better performance than extensions with which we experimented so far.[4] We overcome the problem with partially consumed strings by introducing a simplification: once an expression e accepts a string x, it succeeds on any other string xy, *i.e.*, it succeeds on any string with x as a prefix. This is reasonable simplification, which does not pose any problems for practical applications, *e.g.*, `'PetitParser'/'Petit'` where :

$$\mathcal{L}(\,\texttt{'PetitParser'/'Petit'}\,) = \left\{ \begin{array}{c} \texttt{'Petit'} \\ \texttt{'PetitP'} \\ ... \\ \texttt{'PetitParser'} \\ ... \end{array} \right\}$$

On the other hand, in some cases the simplification would change the expected results, *e.g.*, `('PetitParser'/'Petit')'P'` where:

$$\mathcal{L}(\,\texttt{('PetitParser'/'Petit')'P'}\,) = \left\{ \begin{array}{c} \texttt{'PetitP'} \\ \texttt{'PetitPa'} \\ ... \\ \texttt{'PetitParse'} \\ \texttt{'PetitParserP'} \\ ... \end{array} \right\}$$

but the string `'PetitParser'` is not in the language.

[4] An efficient implementation of FSAs that can deal with partially consumed strings remains a subject of our further research.

To detect the cases when this simplification breaks the semantics, we define a *concatenation prefix set* that is a set of all expression tuples (e_1, e_2) whose concatenation leads to the broken semantics. First of all, we introduce a prefix set:

Definition E.1 (Prefix Set)**.** A non-empty string p is a *prefix* of a string u if the string $u = pv$. A *prefix set* P_u is the set of all possible prefixes p of a string u:

$$P_u = \{p \mid pv = u\}$$

□

Definition E.2 (Concatenation Prefix Set ($\mathbb{C}_\mathbb{P}$))**.** A concatenation of strings u, v is a *concatenation prefix* of a string w if $uv \in P_w$.

A *concatenation prefix set* $\mathbb{C}_\mathbb{P}$ is a set of language pairs (L_1, L_2) for which there exist strings $u \in L_1, v \in L_2$ such that uv is a concatenation prefix of some $w \in L_2$:

$$\mathbb{C}_\mathbb{P} = \{(L_1, L_2) \mid \exists u \exists w \in L_1, \exists v \in L_2 : uv \in P_w\}$$

□

For example, for expressions $e_1 =$ `'PetitParser'/'Petit'` and $e_2 =$ `'P'`, concatenation of `'Petit'` and `'P'` is a prefix of `'PetitParser'`. Therefore

$(\mathcal{L}($ `'PetitParser'/'Petit'` $, \mathcal{L}($ `'P'` $)) \in \mathbb{C}_\mathbb{P}$

Definition E.3 (Regular Parsing Expressions (RPEs))**.** Regular parsing expressions (RPEs) are defined inductively, if e_1 and e_2 are regular parsing expressions, then so is:

1. **Empty string** ϵ is a regular parsing expression.

2. **Literal** is a regular parsing expression.

3. **Ordered choice** e_1/e_2 is a regular parsing expression if e_1, e_2 are regular parsing expressions.

4. **Sequence** $e_1 e_2$ is a regular parsing expression if $e_1 e_2$ are regular parsing expressions, and their languages $(\mathcal{L}(e_1), \mathcal{L}(e_2)) \notin \mathbb{C}_\mathbb{P}$

5. **Repetition** $e*$ is a regular parsing expression if e is a regular parsing expression, and the language $(\mathcal{L}(e), \mathcal{L}(e)) \notin \mathbb{C}_\mathbb{P}$, and $\epsilon \notin \mathcal{L}(e)$

□

Theorem 1. *Regular parsing expressions are a subset of parsing expressions,* $RPEs \subset PEs$*.*

Proof. From definition of regular parsing expressions: they are defined from the same terminals using a subset of parsing expression operators excluding nonterminals. □

E.3 Regular Parsing Expression Languages

Now we focus on languages described by RPEs. We show that RPELs are a subset of RLs and thus they can be recognized by FSAs. In standard REs, choice and sequence are equivalent to union and concatenation of their languages. For RPEs we define priority union and priority concatenations of their languages that are equivalent to priority choice and priority sequence.

We start with priority union, follow with priority concatenation and finally we define regular parsing expression languages. First of all, we introduce a concept of *union* and *concatenation terminators*. Intuitively a terminator is a prefix of another string, but the terminator has higher priority and terminates the recognition attempt of the other lower priority string.

Definition E.4 (Union Terminator)**.** A string t is a *union terminator* $t_\cup$ of a string u if $t \in P_u$, *i.e.*, t is a prefix of u:

$$t_\cup(t, u) = \begin{cases} \texttt{true} & t \in P_u \\ \texttt{false} & \text{otherwise} \end{cases}$$

The *union-terminated set* $T_\cup$ of two languages L_1 and L_2 is the set of strings $u \in L_2$ that are terminated by some string $t \in L_1$:

$$T_\cup(L_1, L_2) = \{u \mid \exists t \in L_1 : t \in P_u, u \in L_2\}$$

□

Definition E.5 (Priority Union)**.** The *priority union* of languages L_1, L_2 is defined as the union of L_1 and L_2 excluding all the strings in L_2 that have a string from L_1 as a prefix:

$$L_1 \cup_p L_2 = L_1 \cup L_2 \setminus (T_\cup(L_1, L_2))$$

□

Consider union of two languages { `'Petit'` } and { `'Bounded Seas'` `'PetitParser'` }. The priority union results in { `'Petit'` `'Bounded Seas'` } because `'Petit'` is a prefix of `'PetitParser'` , has higher priority and is a union terminator of `'PetitParser'` .

Theorem 2. *Priority union $\cup_p$ of regular parsing expression languages is equivalent to priority choice of regular parsing expressions:*

$$\mathcal{L}(e_1) \cup_p \mathcal{L}(e_2) \equiv \mathcal{L}(e_1/e_2)\ \forall e_1, e_2 \in \mathbb{RPE}$$

Proof. The definition of priority union (Definition E.5) follows the semantics of priority choice (Definition A.2):

- The *option 1* is covered, because all the strings that are accepted by e_1 are in the union.
- The *option 2:* is covered, because all the strings that are accepted by e_2 and for which e_1 fails are in the union.

If a string xy is accepted by e_2 for which e_1 does not fail, it means that $\exists x : x \in \mathcal{L}(e_1)$, x is a terminator of xy and all such strings are by definition removed from union. □

Definition E.6 (Concatenation Terminator)**.** A string t is a *concatenation terminator* $t_\circ$ of two strings u, v, if t is a prefix of a sequence uv, but not prefix of u:

$$t_\circ(t, u, v) = \begin{cases} \texttt{true} & t \in P_{uv} \text{ and } t \notin P_u \\ \texttt{false} & \text{otherwise} \end{cases}$$

The *terminated concatenation set* $T_\circ$ of two languages L_1 and L_2 is a set of strings $uv : u \in L_1, v \in L_2$ that are terminated by some string $t \in L_1$:

$$T_\circ(L_1, L_2) = \{uv \mid \exists t \in L_1 : t_\circ(t, u, v), u \in L_1, v \in L_2\}$$

□

Consider three strings `'PetitP'`, `'Petit'` and `'Parser'`. `'PetitP'` is a concatenation terminator of `'Petit'` and `'Parser'` because it is a prefix of their concatenation `'PetitParser'`.

Definition E.7 (Priority Concatenation)**.** The *priority concatenation* of a language L_1 and L_2 is defined as follows:

$$L_1 \circ_p L_2 = L_1 \circ L_2 \setminus T_\circ(L_1, L_2)$$

□

Consider a sequence of two expressions `'PetitP'/'Petit'` and `'Parser'`. They are not in $\mathbb{C}_\mathbb{P}$. Neither `'PetitPParser'` (the string recognized by concatenation of `'PetitP'` and `'Petit'`) nor `'PetitParser'` (the string recognized by concatenation of `'Petit'` and `'Parser'`) is a prefix of `'PetitP'` or `'Petit'`. But the language of this sequence contains only `'PetitPParser'`:

$$\mathcal{L}(\ \texttt{('PetitP'/'Petit') 'Parser'}\ ) = \{\ \texttt{'PetitPParser'}\ \}$$

The input `'PetitParser'` is not in the language, because the `'PetitP'/'Petit'` rule consumes `'PetitP'` and the `'Parser'` rule fails on the remaining input `'arser'` resulting in failure as follows from Definition A.2, *Sequence (failure 2).*

Theorem 3. *Priority concatenation $\circ_p$ is equivalent to a sequence of parsing expressions:*

$$\mathcal{L}(e_1) \circ_p \mathcal{L}(e_2) \equiv \mathcal{L}(e_1 e_2)\ \forall e_1, e_2 \in \mathbb{RPE}$$

Proof. The definition of priority concatenation (Definition E.7) follows the semantics of sequence (Definition A.2):

- The success case of a sequence is in $\mathcal{L}(e_1) \circ_p \mathcal{L}(e_2)$ and is not in the terminated concatenation set. If e_1 accepts uv and e_2 accepts wx and there is a string $uvwx$ in $\mathcal{L}(e_1) \circ_p \mathcal{L}(e_2)$ removed by the terminated concatenation set, it means that there must be a string uvw in $\mathcal{L}(e_1)$. Because of a greedy nature of PEs, e_1 has to consume the whole uvw and therefore uv and wx cannot be concatenated.

- The failure (case 1) of a sequence means, that there cannot be a string in $L = \mathcal{L}(e_1) \circ_p \mathcal{L}(e_2)$ for which e_1 fails. This must be true, because all the strings in L start with a string accepted by e_1.

- The (failure case 2) of a sequence means, that if e_1 accepts uv and e_2 rejects wx, no string $uvwx$ or its prefix is in $\mathcal{L}(e_1) \circ_p \mathcal{L}(e_2)$. Because $\circ_p$ is not defined for the expressions which concatenation is a prefix of uv and because the concatenation terminator removes any uvw for such that e_1 accepts u and e_2 accepts vw, such a string cannot be in the sequence $\mathcal{L}(e_1) \circ_p \mathcal{L}(e_2)$.

□

Definition E.8 (Regular Parsing Expression Languages (RPELs)). Finally we define a *regular parsing expression language*. If $L_1, L_2 \in \mathbb{RPEL}$, and Σ is an alphabet, we define regular parsing expression languages inductively as follows:

1. **Empty string** $\{\epsilon\}$ is a regular parsing expression language.
2. **Terminal** $\{t \mid t \in \Sigma\}$ is a regular parsing expression language.
3. **Priority Union** $L_1 \cup_p L_2$ is a regular parsing expression language.
4. **Priority Concatenation** $L_1 \circ_p L_2$ is a regular parsing expression language, if $(L_1, L_2) \notin \mathbb{C}_\mathbb{P}$.
5. **Kleene Star** $L_1{}^*$ is a regular parsing language if $(L_1, L_1) \notin \mathbb{C}_\mathbb{P}$

□

Definition E.9 (Regular Languages (RLs)). If $L_1, L_2 \in \mathbb{RL}$, and Σ is an alphabet, parsing languages are inductively defined as follows:

1. **Empty string** ϵ is a regular language.
2. **Terminal** $\{t \mid t \in \Sigma\}$ is a language.
3. **Union** $L_1 \cup L_2$ is a regular language.
4. **Concatenation** $L_1 \circ L_2$ is a regular language.
5. **Kleene Star** L_1^* is a regular language.

□

Theorem 4. *Regular parsing expression languages are a subset of regular languages,* $\mathbb{RPEL} \subset \mathbb{RL}$.

Proof. The definition of $RPELs$ mimics the definition of RLs with the following differences:

- Instead of union, it uses priority union. Priority union results in a subset of union.
- Instead of sequence, it uses priority sequence. Priority sequence results in a subset of sequence and is defined only for a subset of RPELs (a pairs of RPELs that are not in $\mathbb{C}_\mathbb{P}$).

Therefore, $\mathbb{RPEL}$ is a subset of $\mathbb{RL}$. □

Theorem 5. *Regular parsing expressions are equivalent to regular parsing expression languages.*

Proof. Because the definition of RPEs mimics the structure of definition of RPELs and because the operations on RPEs and RPELs are equivalent:

- The priority union is equivalent to a priority choice of regular parsing expressions.
- The priority concatenation is equivalent to a sequence of regular parsing expressions.

□

Theorem 6. *Contrary to PELs, priority sequence and priority choice are distributive for RPELs.*

Proof. For expression $(e_1/e_2)e_3$ the distributivity $(e_1/e_2)e_3 \equiv (e_1e_3/e_2e_3)$ is violated for a string $uvwxy$ if:

1. $uvw \in \mathcal{L}(e_1), x \notin \mathcal{L}(e_3)$,
   $u \in \mathcal{L}(e_2), v \in \mathcal{L}(e_3)$
2. $uvw \in \mathcal{L}(e_1), x \notin \mathcal{L}(e_3)$,
   $u \in \mathcal{L}(e_2), vw \in \mathcal{L}(e_3)$
3. $uvw \in \mathcal{L}(e_1), x \notin \mathcal{L}(e_3)$,
   $u \in \mathcal{L}(e_2), vwx \in \mathcal{L}(e_3)$

If e_1 and $e_2, e_3 \in \mathbb{RPE}$, cases 1) and 2) are covered by concatenation prefixes $\mathbb{C}_{\mathbb{P}}$, by definition the priority sequence is not defined for these. Case 3) is covered by priority sequence definition, which excludes $uvwx$ from the language because uvw is a concatenation terminator of $uvwx$. □

Summary In the previous section (section E.2) we have shown that RPEs are a subset of PEs. In this section we have shown that RPEs describe RPELs and that RPELs are a subset of RLs. It is already known, that RLs can be recognized by FSAs and because RPELs are a subset of RLs, they can be recognized by FSAs as well.

E.4 Finite State Automata

Finite state automata (FSAs) are equivalent to RLs [Yu97]. And because RPELs are subset of RLs (see Theorem 4), we can use FSAs to recognize RPEs as well.

First we provide standard definitions of finite state automata. Because priority union and priority concatenation require union and concatenation terminators, we extend the automata with priorities and terminating states that are used for this purpose. We describe how to transform automata with priorities to deterministic automata.

Definition E.10 (Deterministic Finite State Automata ($\mathbb{DFSA}$)). A *deterministic finite state automaton* is represented by a 5-tuple $(Q, \Sigma, \delta, q_s, F)$ where

- Q is a finite set of states
- Σ is a finite set of input symbols
- δ is a transition function $\delta : Q \times \Sigma \to Q$

- q_s is a start state $q_s \in Q$
- F is a set of final states

□

Definition E.11 (Nondeterministic Finite State Automata ($\mathbb{NFSA}$)). A *nondeterministic finite state automaton* is represented by a 5-tuple $(Q, \Sigma, \Delta, q_s, F)$ where

- Q is a finite set of states
- Σ is a finite set of input symbols
- Δ is a transition function $\delta : Q \times \Sigma \to \mathbb{P}(Q)$
- q_s is a start state $q_s \in Q$
- F is a set of final states

where $\mathbb{P}(Q)$ is the power set of Q. □

Definition E.12 (Nondeterministic Finite State Automata with ϵ-transitions (ϵ-$\mathbb{FSA}$)). A *nondeterministic finite state automaton with ϵ-transitions* is represented by a 5-tuple $(Q, \Sigma, \Delta, q_s, F)$ where

- Q is a finite set of states
- Σ is a finite set of input symbols
- Δ is a transition function $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathbb{P}(Q)$
- q_s is a start state $q_s \in Q$
- F is a set of final states

□

To detect union and concatenation terminators, we extend FSAs with terminating states and priorities. The terminating states represent the union and concatenation terminators that terminate outgoing transitions of the low-priority states. In the context of FSAs, the terminator state terminates the paths representing the low-priority alternative of a choice.

Definition E.13 (Deterministic Finite State Automata with priorities ($\mathbb{PDFSA}$)). A *deterministic finite state automaton with priorities* is represented by a 7-tuple $(Q, \Sigma, \delta, q_s, F, T, \pi)$ where

- Q is a finite set of states
- Σ is a finite set of input symbols
- δ is a transition function $\delta : Q \times \Sigma \to Q$
- q_s is a start state $q_s \in Q$
- F is a set of final states
- T is a set of terminating states

- π is a priority function $\pi : Q \to \{0, -1\}$

□

Definition E.14 (Nondeterministic Finite State Automata with priorities ($\mathbb{PFSA}$))**.** A *nondeterministic finite state automaton with priorities* is represented by a 7-tuple $(Q, \Sigma, \Delta, q_s, F, T, \pi)$ where

- Q is a finite set of states
- Σ is a finite set of input symbols
- Δ is a transition function $\Delta : Q \times \Sigma \to \mathbb{P}(Q)$
- q_s is a start state $q_s \in Q$
- F is a set of final states
- T is a set of terminating states
- π is a priority function $\pi : Q \to \{0, -1\}$

□

Definition E.15 (Nondeterministic Finite State Automata with ϵ-transitions and priorities (ϵ-$\mathbb{PFSA}$))**.** A *nondeterministic finite state automaton with ϵ-transitions and priorities* is represented by a 7-tuple $(Q, \Sigma, \Delta, q_s, F, T, \pi)$ where

- Q is a finite set of states
- Σ is a finite set of input symbols
- Δ is a transition function $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathbb{P}(Q)$
- q_s is a start state $q_s \in Q$
- F is a set of final states
- T is a set of terminating states
- π is a priority function $\pi : Q \to \{0, -1\}$

□

The terminating states are tagged with superscript T, *i.e.*, s^T is a terminating state s. The low-priority states are tagged with superscript –, *i.e.*, s^- is a low-priority state s.

Our method to construct a $DFSA \in \mathbb{DFSA}$ from a regular parsing expression RPE is inspired by Thompson's algorithm [Tho68]. We define a function $\mathcal{DFSA} : \mathbb{RPE} \to \mathbb{DFSA}$ that converts a regular parsing expression to a deterministic finite state automaton. The $\mathcal{DFSA}$ function is defined for different kinds of regular parsing expressions as follows:

$$\mathcal{DFSA}(e) = \begin{cases} \mathcal{D}(\mathcal{FSA}(e)) & \text{if } e = \epsilon \\ \mathcal{D}(\mathcal{FSA}(e)) & \text{if } e = x \\ \mathcal{D}(\mathcal{FSA}(\mathcal{DFSA}(e_1)/\mathcal{DFSA}(e_2)) & \text{if } e = e_1/e_2 \\ \mathcal{D}(\mathcal{FSA}(\mathcal{DFSA}(e_1) \circ_p \mathcal{DFSA}(e_2)) & \text{if } e = e_1 e_2 \\ \mathcal{D}(\mathcal{FSA}(\mathcal{DFSA}(e)*)) & \text{if } e = e' \end{cases}$$

where 1. $\mathcal{FSA}$ is a function that transforms RPE to ϵ-$PFSA$; and 2. $\mathcal{D}$ is a determination function that transforms ϵ-$PFSA$ to $DFSA$. In the remainder of this section we provide a description of $\mathcal{FSA}$ and $\mathcal{D}$.

E.4.1 Construction of finite state automata from regular parsing expressions ($\mathcal{FSA}$)

The $\mathcal{FSA}$ function is a function $\mathcal{FSA} : \mathbb{RPE} \rightarrow \epsilon\text{-}\mathbb{PFSA}$ that is defined for different kinds of regular parsing expressions as described in the following text.

Empty string ϵ is represented by ϵ-$PFSA$ with two states with an ϵ-transition between them. A schematic example is in Figure E.1.

$$\mathcal{FSA}(\epsilon) = \epsilon\text{-}PFSA(Q, \Sigma, \Delta, q_s, F, T, \pi))$$

where

- $Q = \{q_s, q_f\}$
- $\Sigma = \emptyset$
- $\Delta = (q_s, \epsilon) \rightarrow q_f$
- $F = \{q_f\}$
- $T = \emptyset$
- $\pi = \{q \rightarrow 0 \mid \forall q \in Q\}$

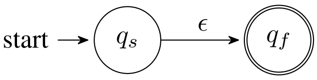


Figure E.1: ϵ-$PFSA$ for an empty string ϵ.

Literal string x is represented by a start state and a transition to the following state for each character in the string. A schematic example is in Figure E.2.

$$\mathcal{FSA}(x) = \epsilon\text{-}PFSA(Q, \Sigma, \Delta, q_1, F, T, \pi)$$

where

- $x = c_1c_2..c_n$
- $Q = \{q_1, ..., q_{n+1}\}$
- $\Sigma = \{c_1, ..., c_n\}$
- $\Delta = (q_i, c_i) \rightarrow q_{i+1}, \forall q_i \in Q \setminus \{q_{n+1}\}$
- $F = \{q_{n+1}\}$
- $T = \emptyset$
- $\pi = \{q \rightarrow 0 \mid \forall q \in Q\}$

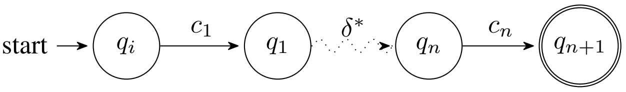


Figure E.2: ϵ-$PFSA$ for a string $x = c_1c_2..c_n$.

Choice e_1/e_2 is represented by a start state q_s with an epsilon transition to a start state q_s of $\mathcal{FSA}(e_1)$ and q_s of $\mathcal{FSA}(e_2)$. The priorities of states in $\mathcal{DFSA}(e_2)$ are decreased. A schematic example is in Figure E.3.

$$\mathcal{FSA}(e_1/e_2) = \epsilon\text{-}PFSA(Q, \Sigma, \Delta, q_s, F, \pi)$$

where

- $\mathcal{DFSA}(e_1) = DFSA(Q_{e_1}, \Sigma_{e_1}, \Delta_{e_1}, q_{s_{e_1}}, F_{e_1})$
- $\mathcal{DFSA}(e_2) = DFSA(Q_{e_2}, \Sigma_{e_2}, \Delta_{e_2}, q_{s_{e_2}}, F_{e_2})$
- $Q = Q_{e_1} \cup Q_{e_2} \cup \{q_s\}$
- $\Sigma = \Sigma_{e_1} \cup \Sigma_{e_2}$
- $\Delta = \Delta_{e_1} \cup \Delta_{e_2} \cup \{(q_s, \epsilon) \rightarrow \{s_{e_1}, s_{e_2}\}\}$
- $F = F_{e_1} \cup F_{e_2}$
- $T = F_{e_1}$
- $\pi = \{q_{e_1} \rightarrow 0\} \cup \{q_{e_2} \rightarrow -1\}, \forall q_{e_1} \in Q_{e_1}, \forall q_{e_2} \in Q_{e_2}$

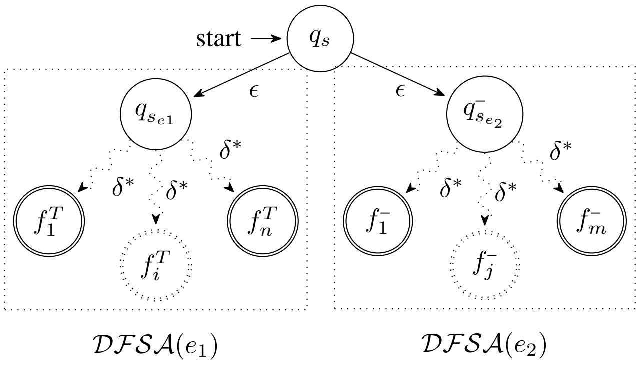


Figure E.3: ϵ-$PFSA$ for a choice $e = e_1/e_2$.

Sequence e_1e_2 is represented by ϵ-$PFSA$ where all the finite states $\{f_1...f_n\} \in F$ of $\mathcal{DFSA}(e_1)$ are connected by an epsilon transition to the start state q_s of $\mathcal{DFSA}(e_2)$. The priorities of states in $\mathcal{DFSA}(e_2)$ are decreased. A schematic drawing is in Figure E.4.

$$\mathcal{FSA}(e_1e_2) = \epsilon\text{-}PFSA(Q, \Sigma, \Delta, q_s, F, \pi)$$

where

- $\mathcal{DFSA}(e_1) = DFSA(Q_{e_1}, \Sigma_{e_1}, \delta_{e_1}, q_{s_{e_1}}, F_{e_1})$
- $\mathcal{DFSA}(e_2) = DFSA(Q_{e_2}, \Sigma_{e_2}, \delta_{e_2}, q_{s_{e_2}}, F_{e_2})$
- $Q = Q_{e_1} \cup Q_{e_2}$
- $\Sigma = \Sigma_{e_1} \cup \Sigma_{e_2}$
- $\Delta = \Delta_{e_1} \cup \Delta_{e_2} \cup \{(q_f, \epsilon) \to \{s_{e_1}\} \mid \forall q_f \in F_{e_1}\}$
- $q_s = q_{s_{e_2}}$
- $F = F_{e_2}$
- $T = F_{e_1}$
- $\pi = \{q_{e_1} \to 0\} \cup \{q_{e_2} \to -1\}, \forall q_{e_1} \in Q_{e_1}, \forall q_{e_2} \in Q_{e_2}$

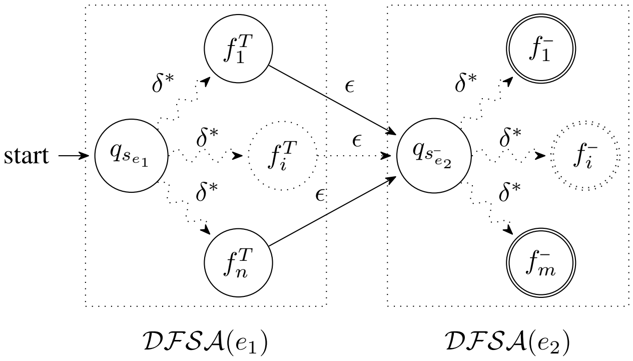


Figure E.4: ϵ-$PFSA$ for a sequence e_1e_2.

Repetition $e*$ is represented by FSA where all the finite states are connected by an ϵ-transition to the start state. A schematic example is in Figure E.5.

$$\mathcal{FSA}(e*) = \epsilon\text{-}PFSA(Q, \Sigma, \Delta, q_s, F, T, \pi)$$

where

- $\mathcal{DFSA}(e) = DFSA(Q_e, \Sigma_e, \delta_e, q_{s_e}, F_e)$
- $Q = Q_e$
- $\Sigma = \Sigma_e$
- $\Delta = \Delta_e \cup \{(q_f, \epsilon) \to \{s_e\} \mid \forall q_f \in F_e\}$
- $q_s = q_{s_e}$
- $F = F_e$
- $T = F_e$
- $\pi = \{q \to 0 \mid \forall q \in Q\}$

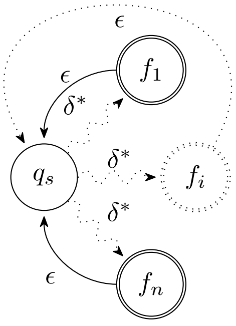


Figure E.5: ϵ-$PFSA$ for a repetition $e*$.

E.4.2 Determinization of the automata with epsilons and priorities ($\mathcal{D}$)

We call *determinization* a process that transforms ϵ-$PFSA$ to $DFSA$. To transform ϵ-$PFSA$ to $DFSA$ we perform three transformations:

$$\mathcal{D}(\epsilon\text{-}PFSA) = \mathcal{R}_\pi(\mathcal{R}_\Delta(\mathcal{R}_\epsilon(\epsilon\text{-}PFSA)))$$

1. ϵ-transitions removal with an ϵ removal function $\mathcal{R}_\epsilon : \epsilon\text{-}\mathbb{PFSA} \to \mathbb{PFSA}$
2. disambiguation with a disambiguation function ($\mathcal{R}_\Delta$) that removes all the ambiguous states $\mathcal{R}_\Delta : \mathbb{PFSA} \to \mathbb{PDFSA}$; and
3. priority removal with a priority-removal function: $\mathcal{R}_\pi : \mathbb{PDFSA} \to \mathbb{DFSA}$

Epsilon transitions removal

ϵ-transitions are removed using a standard ϵ-transition removal method [ASU86]. $\mathcal{R}_\epsilon$ is a function $\mathcal{R}_\epsilon : \epsilon\text{-}\mathbb{PFSA} \to \mathbb{PFSA}$:

$$\mathcal{R}_\epsilon(\epsilon\text{-}PFSA) = PFSA'$$

where

- ϵ-$PFSA$ = $(Q, \Sigma, \Delta, q_s, F, T, \pi)$
- $PFSA$' = $(Q, \Sigma, \Delta', q_s, F, \pi)$
- ϵ-$closure(q, \Delta)$ returns a set of states reachable through ϵ-transitions from state $q \in Q$
- $move(R, \Delta, s)$ returns all the the states reachable from a set of states $R \subseteq Q$ on symbol $s \in \Sigma$
- $\Delta'(q, s) = \Delta(q, s) \cup move(\epsilon\text{-}closure(q, \Delta), s), \forall s \in \Sigma$
- $R \subseteq Q, q \in Q, s \in \Sigma$

Disambiguation

To determine states we create a unique mapping $m : \mathbb{P}(Q) \rightarrow Q'$. Each state $q' \in Q'$ represents a set $P \in \mathbb{P}(Q)$.

$$\mathcal{R}_\Delta(PFSA) = PDFSA$$

where

- $PFSA = (Q, \Sigma, \Delta, q_s, F, T, \pi)$
- $PDFSA = (Q', \Sigma', \delta', q'_s, F', T', \pi')$
- $\Sigma' = \Sigma$
- $q'_s = m(q_s)$
- δ' is a transition function $\delta' : Q' \times \Sigma' \rightarrow Q'$
- $F' \in \mathbb{P}(Q')$
- $T' = \emptyset$
- π' is a priority function $\pi' : Q' \rightarrow \{0, 1\}$

δ' pays special attention to priorities. In case $q' \in Q'$ represents a state with a terminating state, *i.e.*, if $m(P) = q'$, where $T \cap P \neq \emptyset$, only transitions from high priority states are included:

$$\delta'(q', s) = \begin{cases} m(\bigcup(\Delta(p, s) \mid p \in P)) & P \cap T = \emptyset \\ m(\bigcup(\Delta(p, s) \mid \pi(p) = 0)) & T \cap F \neq \emptyset \end{cases}$$

The priority case mimics the logic of priority choice of PEGs — once a successful option is found, no more options are tried. F' contains all the states $q' \in Q$ that represents a $P \in \mathbb{P}(Q)$ that contains at least one final state:

$$F' = \{q_f \mid P \cap F \neq \emptyset, m(P) = q_f\}$$

and π' is the maximum priority of states that are represented by q':

$$\pi'(q') = max\{\pi(p) \mid p \in P, m(P) = q'\}$$

Priority removal

Removing the priorities is a trivial step. The $\mathcal{R}_\pi$ is a function $\mathcal{R}_\pi : \mathbb{PDFSA} \rightarrow \mathbb{DFSA}$:

$$\mathcal{R}_\pi(Q, \Sigma, \delta, q_s, F, T, \pi) = (Q, \Sigma, \delta, q_s, F)$$

F
Measurements

In this chapter we show all the graphs and measurements that did not fit into chapter 6. We provide a zoomed-in version of graphs, when necessary. We also report on time per character and measurements of time spend in garbage collector. In section F.2 we add graphs with speedup and time per character for each of the configurations for all the measured parsers.

F.1 Summary

The overview of speedup of measurements is in Figure F.1, the zoomed-in version is in Figure F.2, time per character and its zoomed-in version is in Figure F.3 and in Figure F.4 respectively.

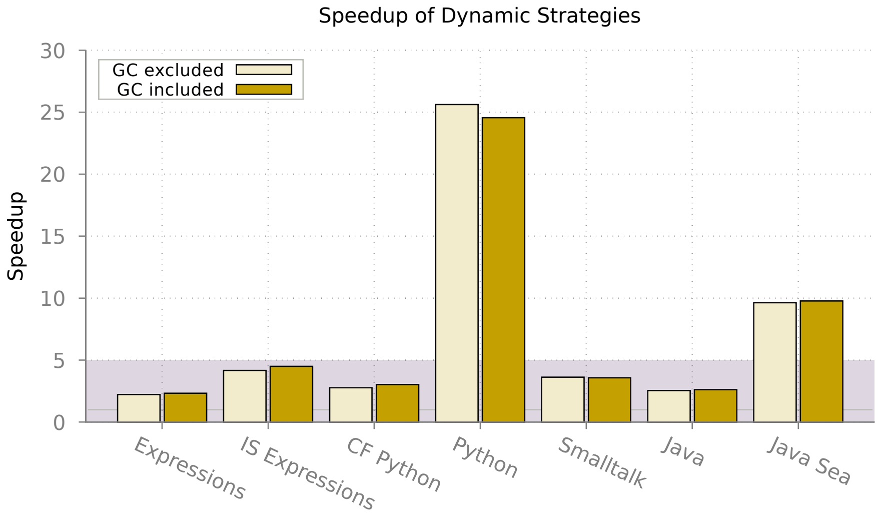


Figure F.1: The speedup of compilation for different grammars.

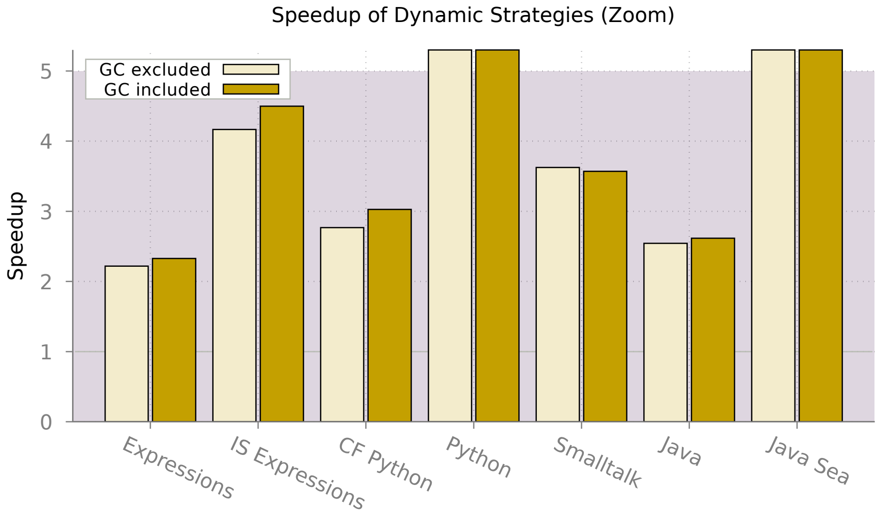


Figure F.2: The speedup of compilation for different grammars, the zoomed-in version.

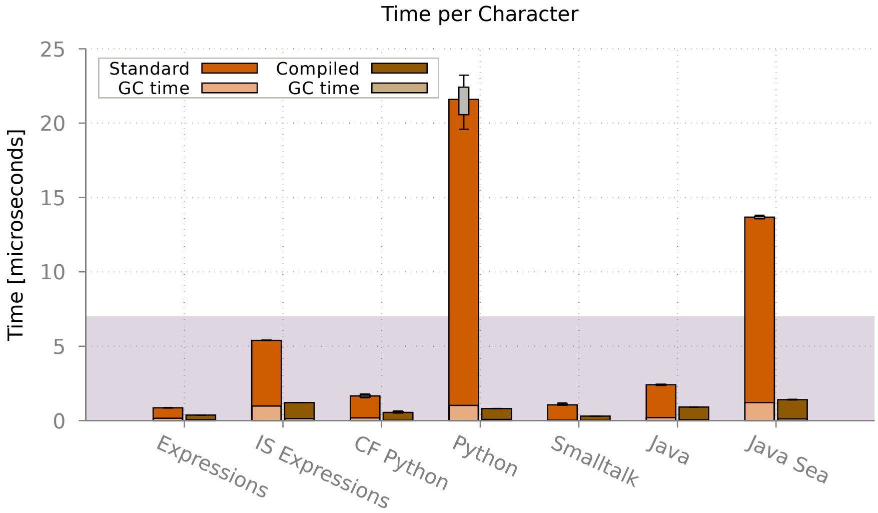


Figure F.3: Time per character of plain PetitParser and its compiled version for different grammars.

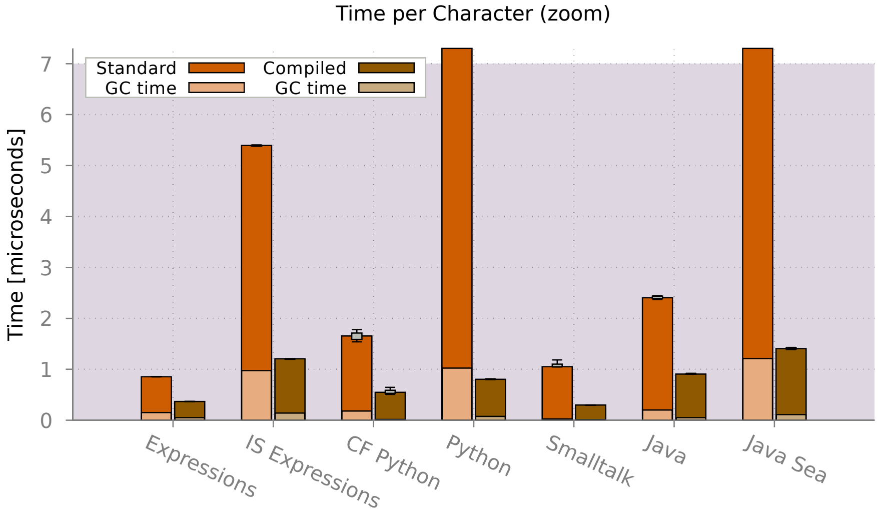


Figure F.4: Time per character of plain PetitParser and its compiled version for different grammars, the zoomed-in version.

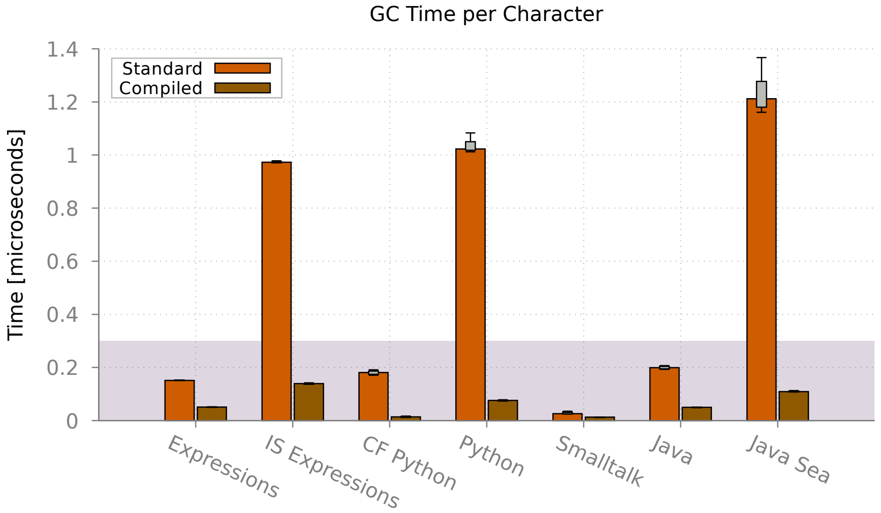


Figure F.5: Time spend in garbage collector for plain PetitParser and its compiled version for different grammars.

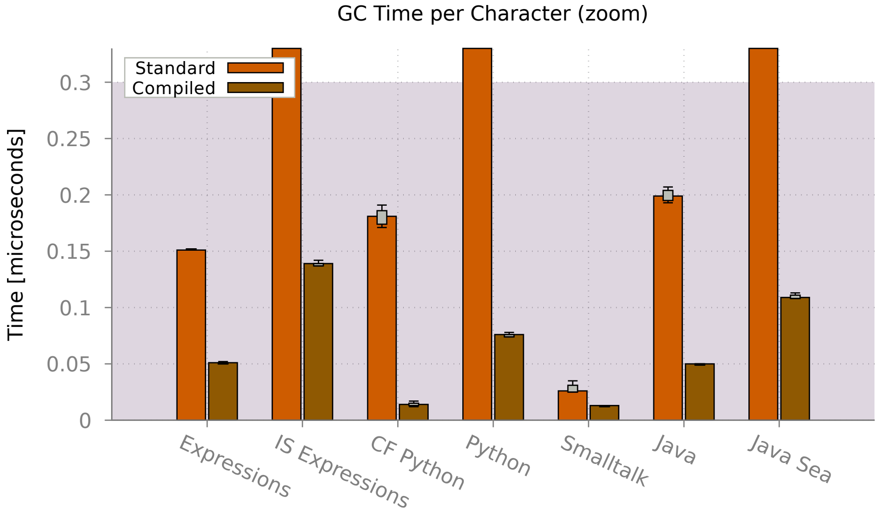


Figure F.6: Time spend in garbage collector for plain PetitParser and its compiled version for different grammars, the zoomed-in version.

F.2 Strategies Details

In this section we provide detailed graphs with impact of strategies from subsection 6.4.3 on all the measured parser. The grouped speedup is in Figure F.7, the zoomed-in version is in Figure F.8, time per character and its zoomed-in version is in Figure F.9 and in Figure F.10 respectively.

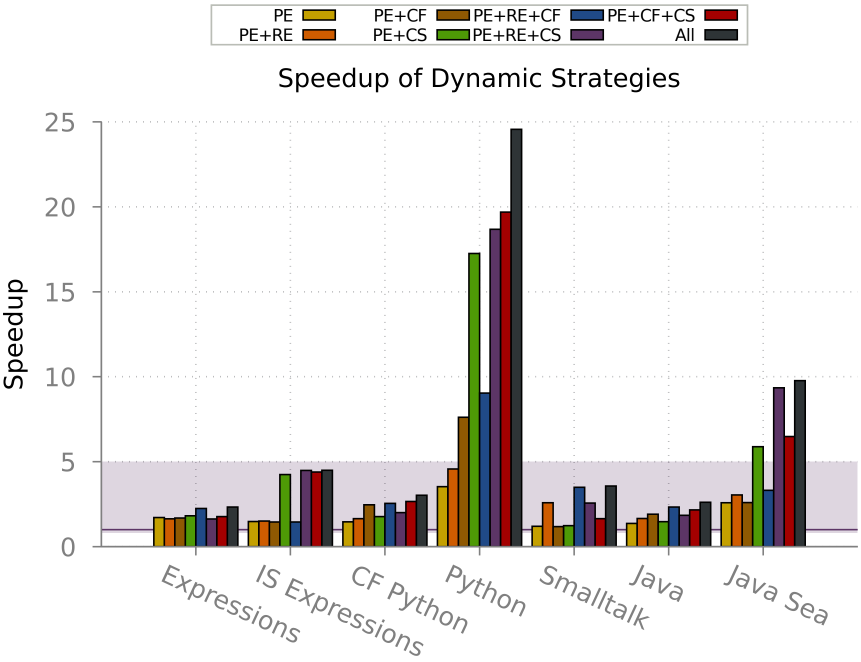


Figure F.7: Speedup against plain PetitParser for various configurations.

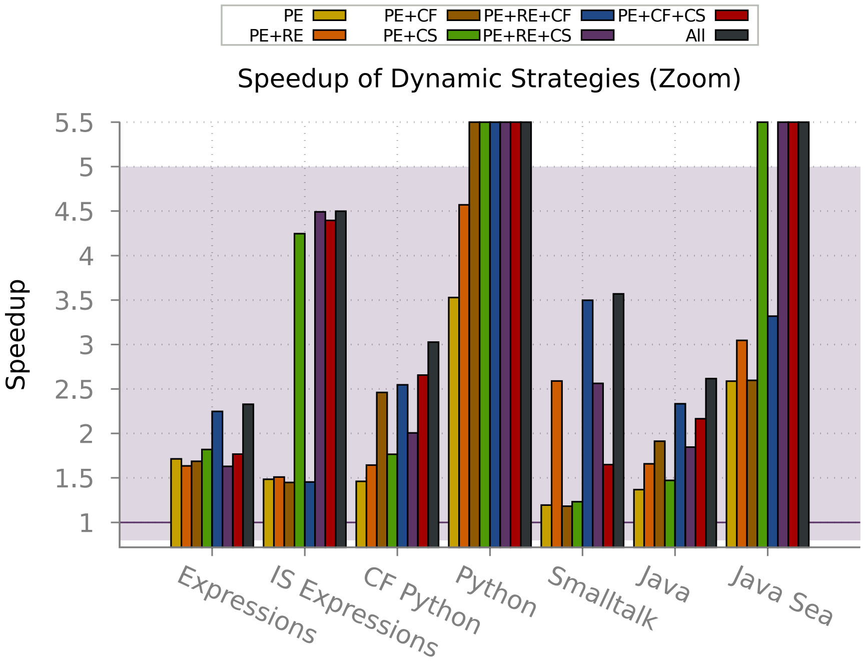


Figure F.8: Speedup against plain PetitParser for various configurations, the zoomed-in version.

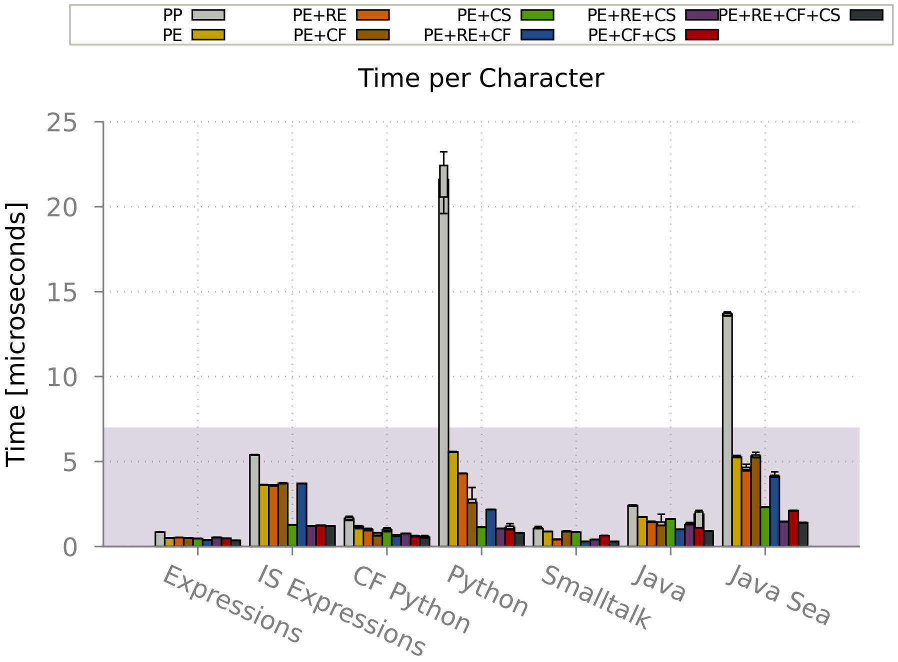


Figure F.9: Time per character of plain PetitParser and its compiled version for various configurations.

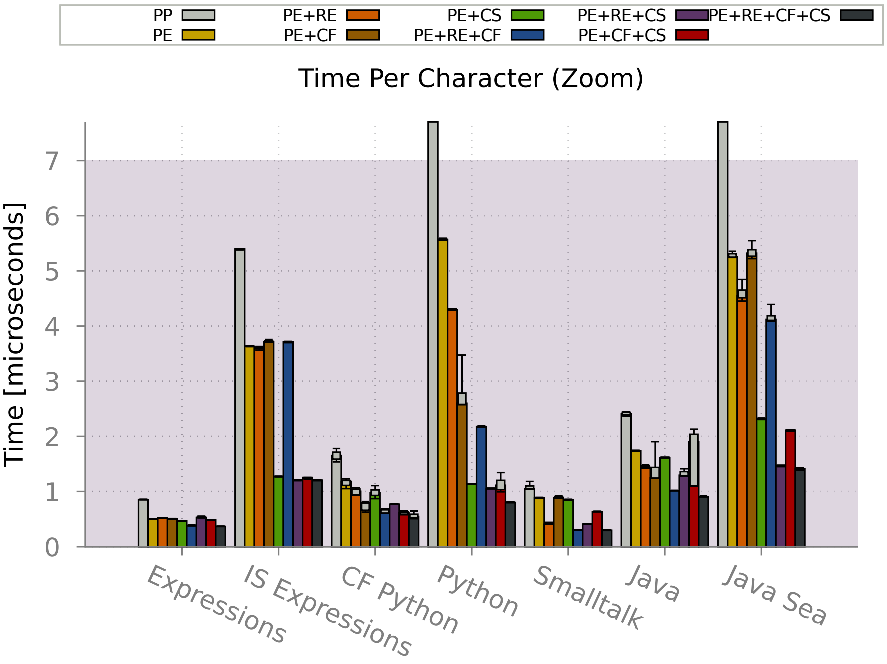


Figure F.10: Time per character of plain PetitParser and its compiled version for various configurations, the zoomed-in version.

F.2.1 Expressions

Speedup of different configurations compared to the plain PetitParser version of Expressions is in Figure F.11 and time per character is in Figure F.12.

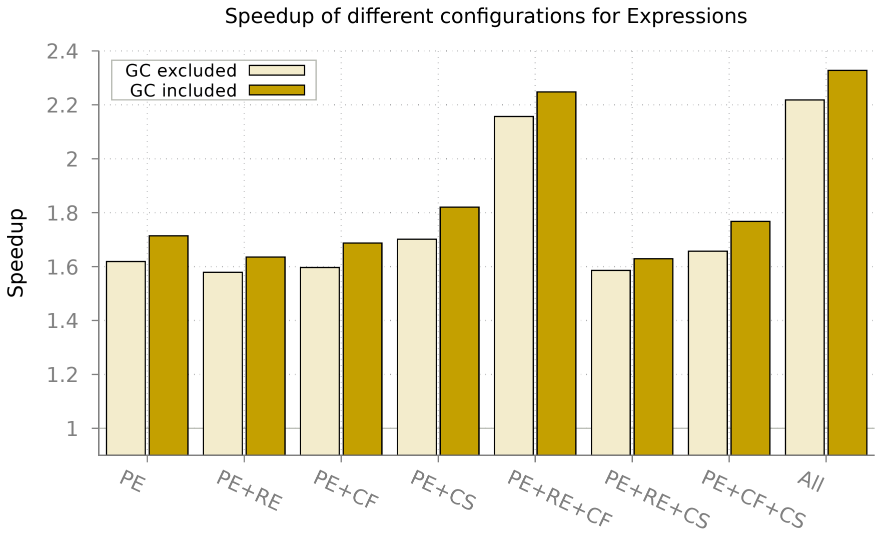


Figure F.11: Speedup of Expressions against PetitParser for various configurations.

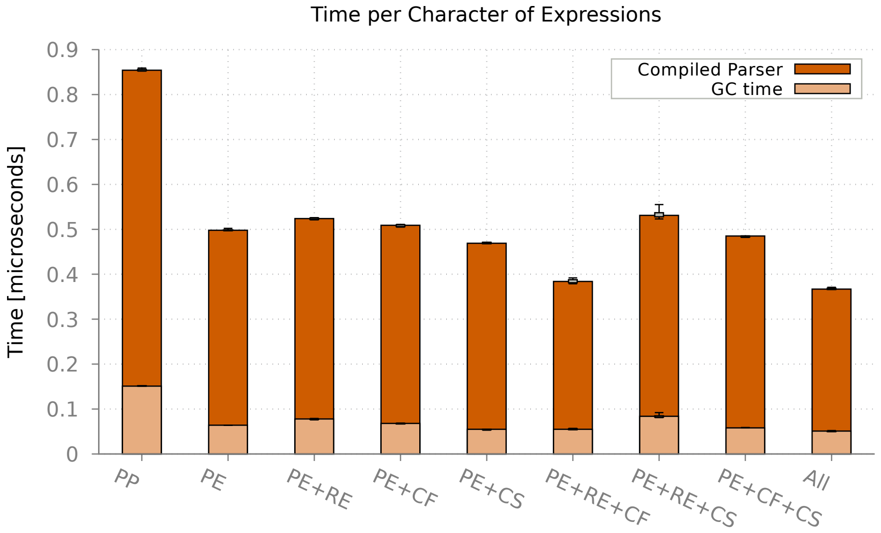


Figure F.12: Time per character of a plain and compiled Expressions parser for various configurations.

F.2.2 IS Expressions

Speedup of different configurations compared to the plain PetitParser version of IS Expressions is in Figure F.13 and time per character is in Figure F.14.

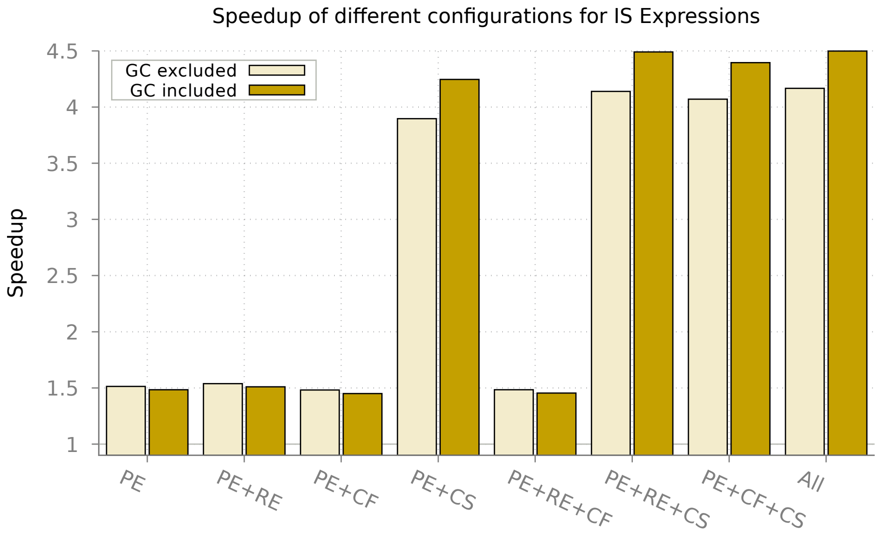


Figure F.13: Speedup of IS Expressions against PetitParser for various configurations.

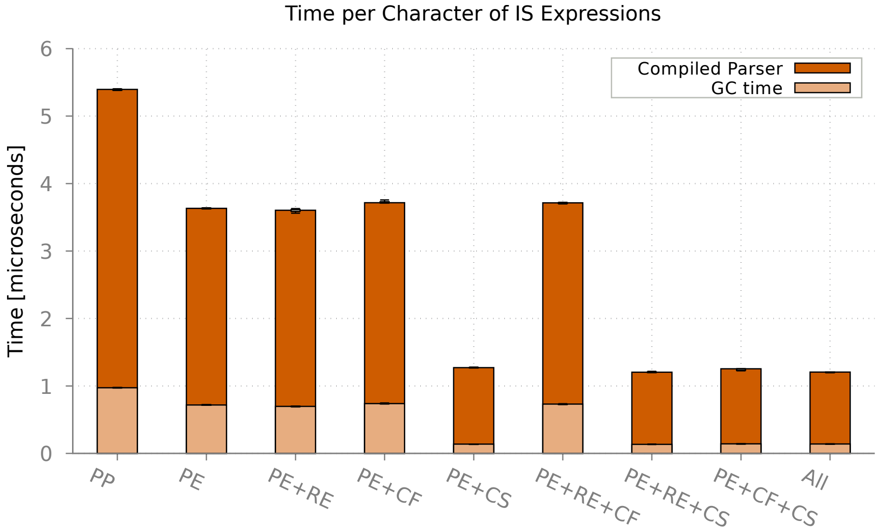


Figure F.14: Time per character of a plain and compiled IS Expressions parser for various configurations.

F.2.3 CF Python

Speedup of different configurations compared to the plain PetitParser version of CF Python is in Figure F.15 and time per character is in Figure F.16.

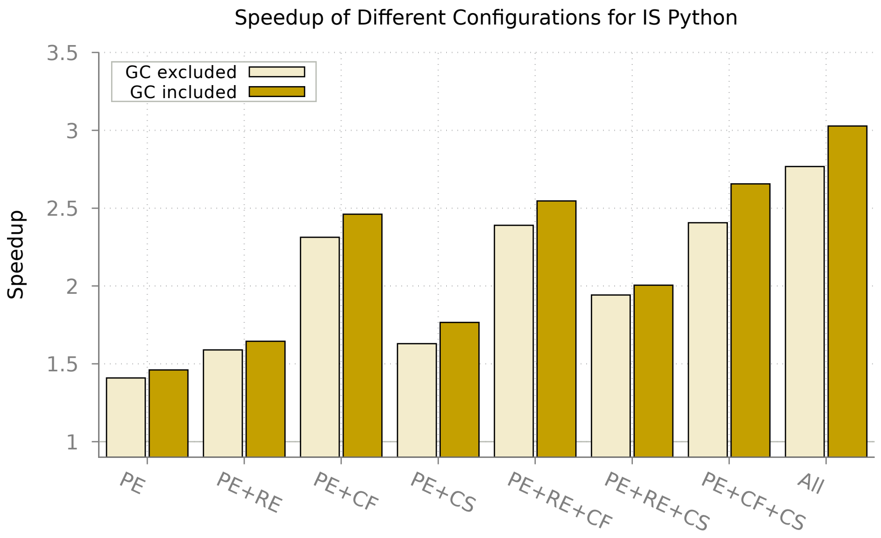


Figure F.15: Speedup of CF Python against PetitParser for various configurations.

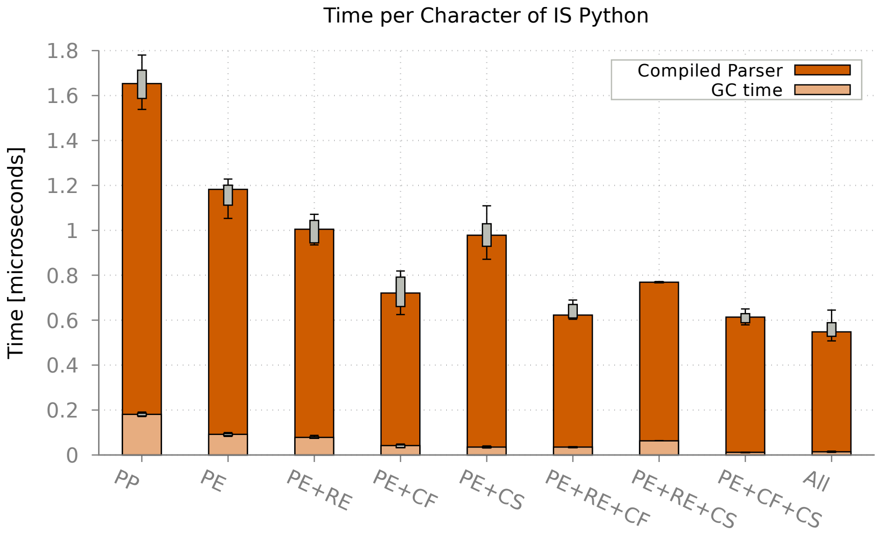


Figure F.16: Time per character of a plain and compiled CF Python parser for various configurations.

F.2.4 Python

Speedup of different configurations compared to the plain PetitParser version of Python is in Figure F.17 and time per character is in Figure F.18.

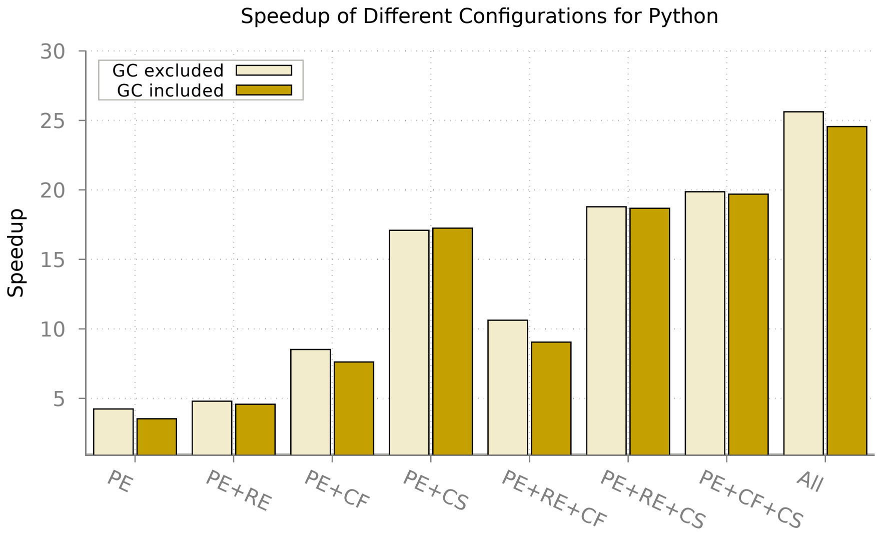


Figure F.17: Speedup of Python against PetitParser for various configurations.

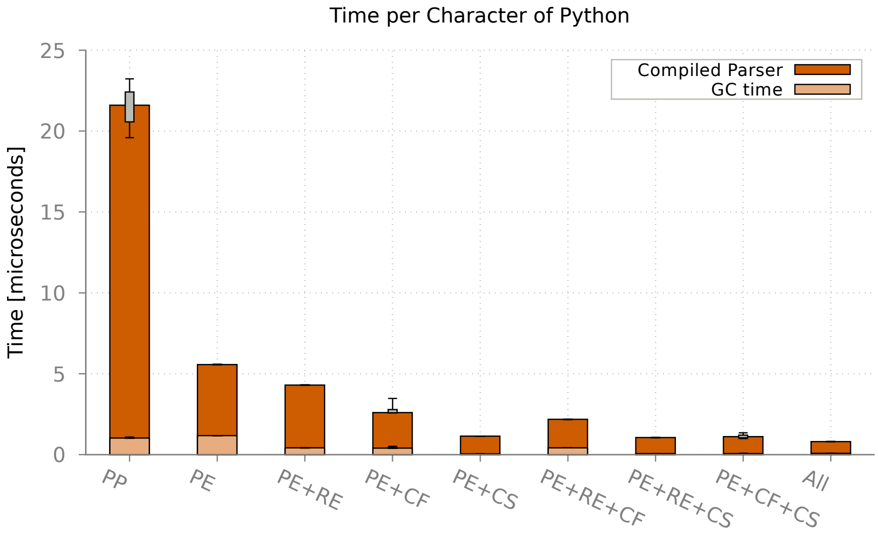


Figure F.18: Time per character of a plain and compiled Python parser for various configurations.

F.2.5 Smalltalk

Speedup of different configurations compared to the plain PetitParser version of Smalltalk is in Figure F.19 and time per character is in Figure F.20.

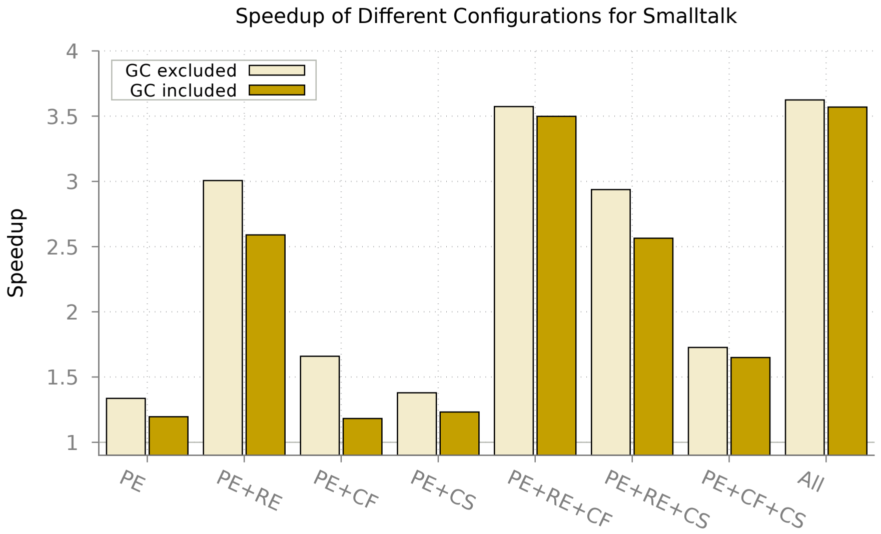


Figure F.19: Speedup of Smalltalk against PetitParser for various configurations.

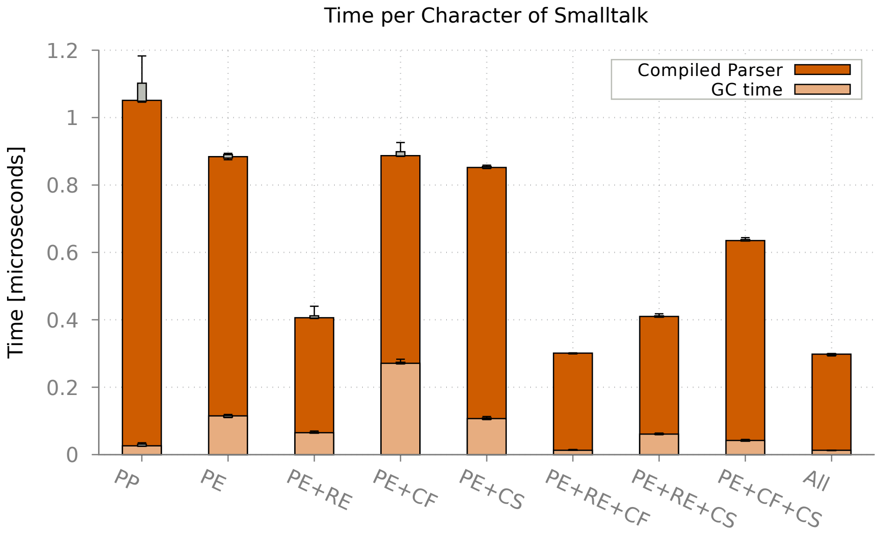


Figure F.20: Time per character of a plain and compiled Smalltalk parser for various configurations.

F.2.6 Java

Speedup of different configurations compared to the plain PetitParser version of Java is in Figure F.21 and time per character is in Figure F.22.

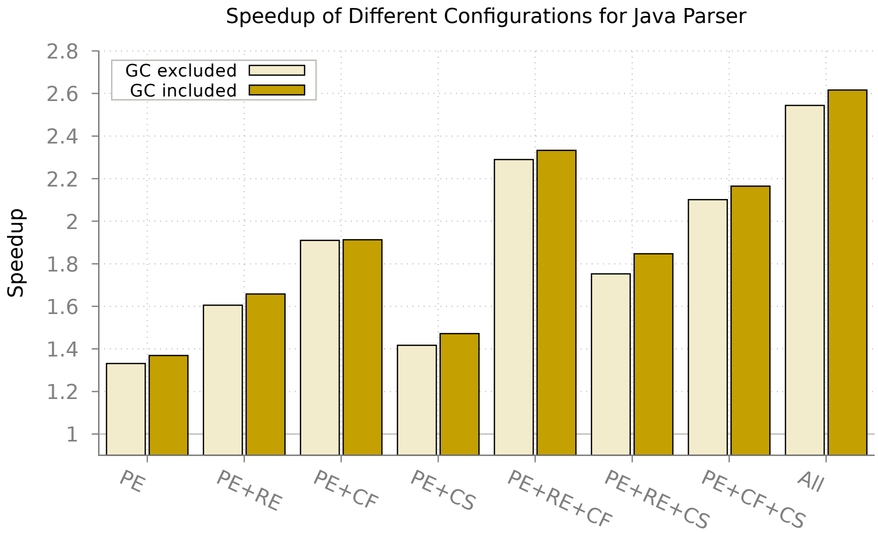


Figure F.21: Speedup of Java against PetitParser for various configurations.

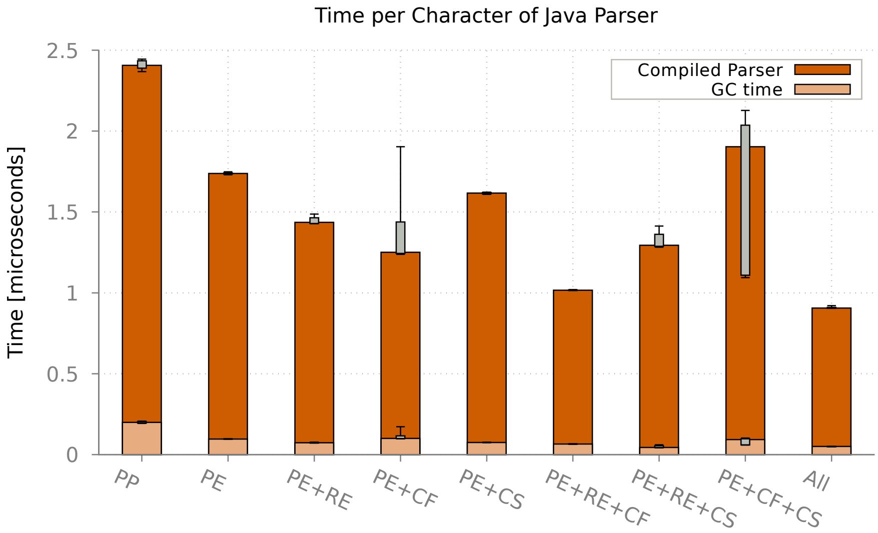


Figure F.22: Time per character of a plain and compiled Java parser for various configurations.

F.2.7 Java Sea

Speedup of different configurations compared to the plain PetitParser version of Java Sea is in Figure F.23 and time per character is in Figure F.24.

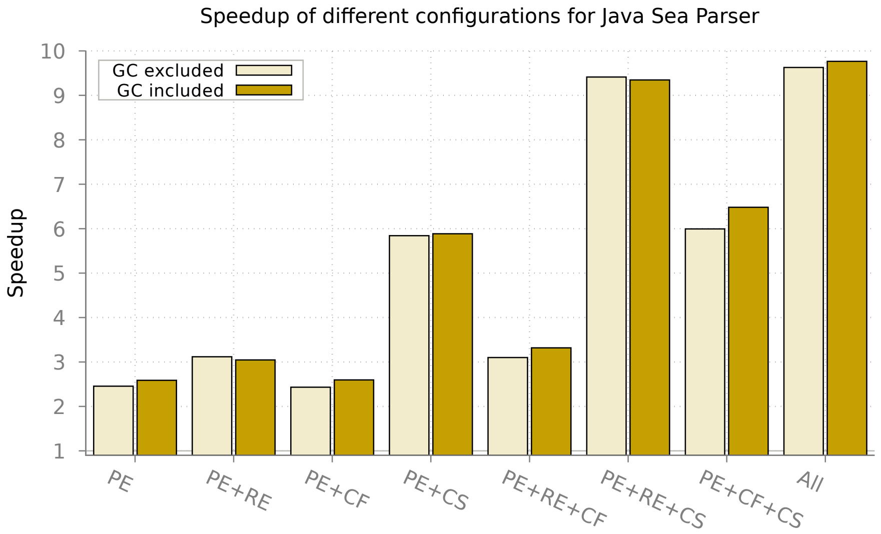


Figure F.23: Speedup of Java Seas against PetitParser for various configurations.

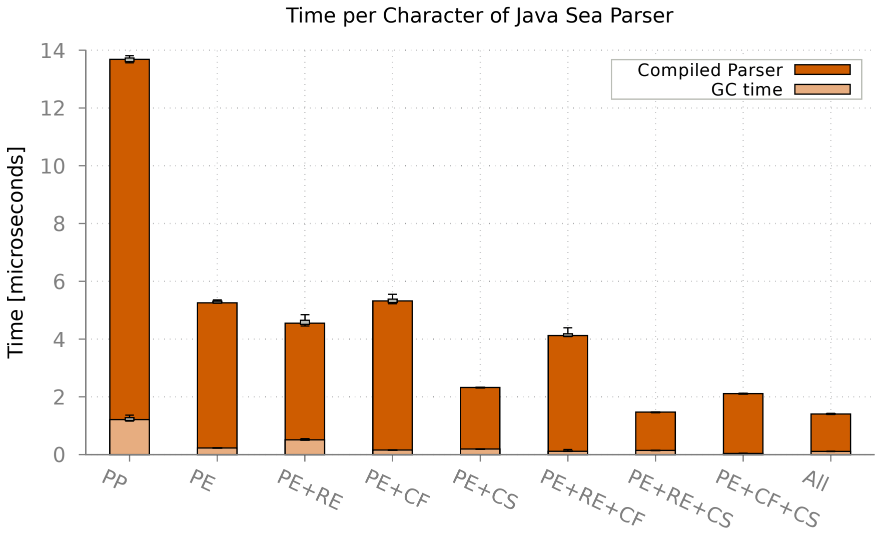


Figure F.24: Time per character of a plain and compiled Java Seas parser for various configurations.

F.3 Scanner Impact

Impact of a scanner on performance of Expressions and Smalltalk parsers is visualized in Figure F.25, time per character is in Figure F.26 and gc time in Figure F.5.

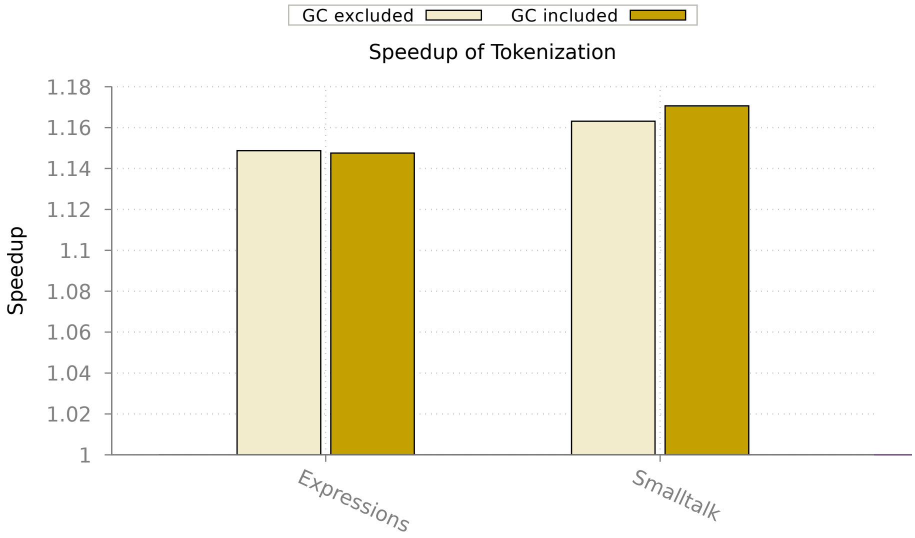


Figure F.25: Performance impact of a scanner.

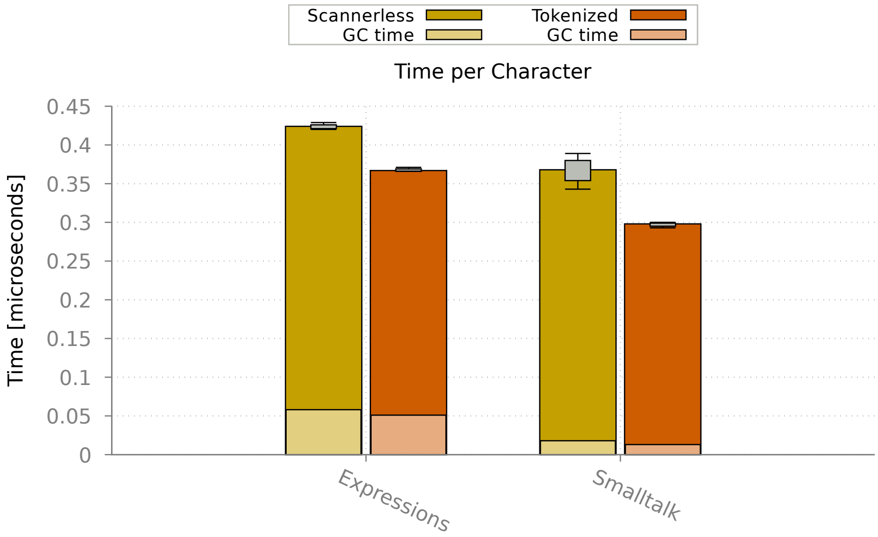


Figure F.26: Time per character of scanning and non-scanning variants of Expressions and Smalltalk parsers.

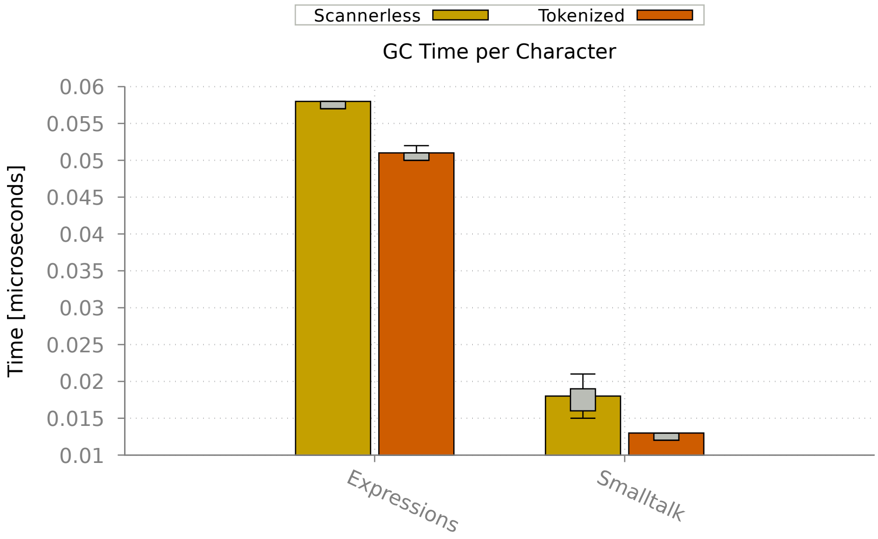


Figure F.27: Time spend in garbage collector of scanning and non-scanning variants of Expressions and Smalltalk parsers.

F.3.1 Expressions

Impact of tokenizing (PE+TRE and PE+TRE+CF) and scannerless (PE+SRE and PE+SRE+CF) configurations on Expressions is in Figure F.28 and time per character in Figure F.29.

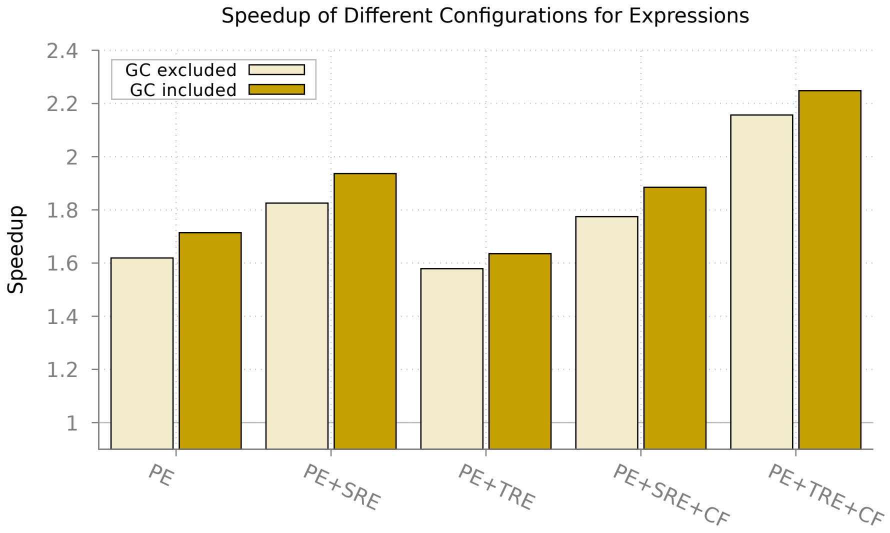


Figure F.28: Speedup of Expressions against PetitParser for various tokenizing and scannerless configurations.

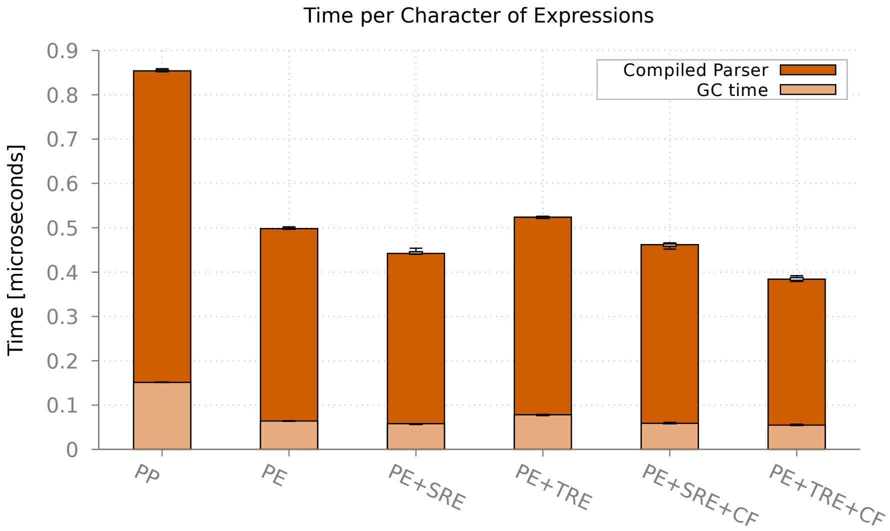


Figure F.29: Time per character of a plain and compiled variant of Expressions for various tokenizing and scannerless configurations.

F.3.2 Smalltalk

Impact of various tokenizing (PE+TRE and PE+TRE+CF) and scannerless (PE+SRE and PE+SRE+CF) configurations on a Smalltalk parser is in Figure F.30 and time per character in Figure F.31.

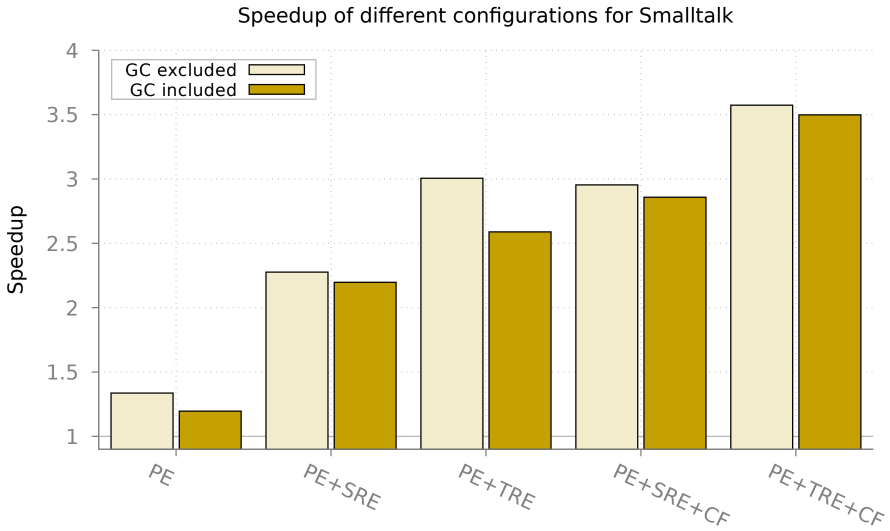


Figure F.30: Speedup of Smalltalk against PetitParser for various tokenizing and scannerless configurations.

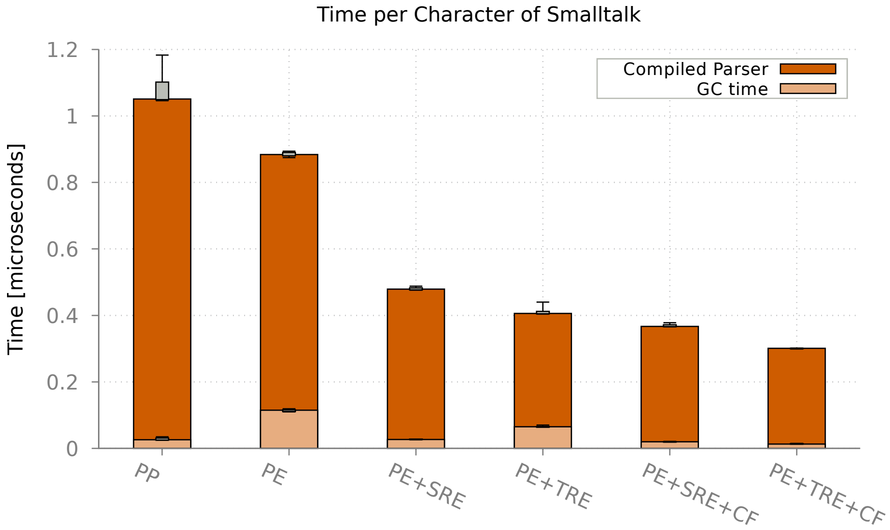


Figure F.31: Time per character of a plain and compiled variant of Smalltalk for various tokenizing and scannerless configurations.

F.4 Memoization Details

Speedup of the *context-sensitive* analysis and the *push–pop* analysis on the IS Expressions, Python and Java Seas parsers is in Figure 6.11 and time per character in Figure F.33.

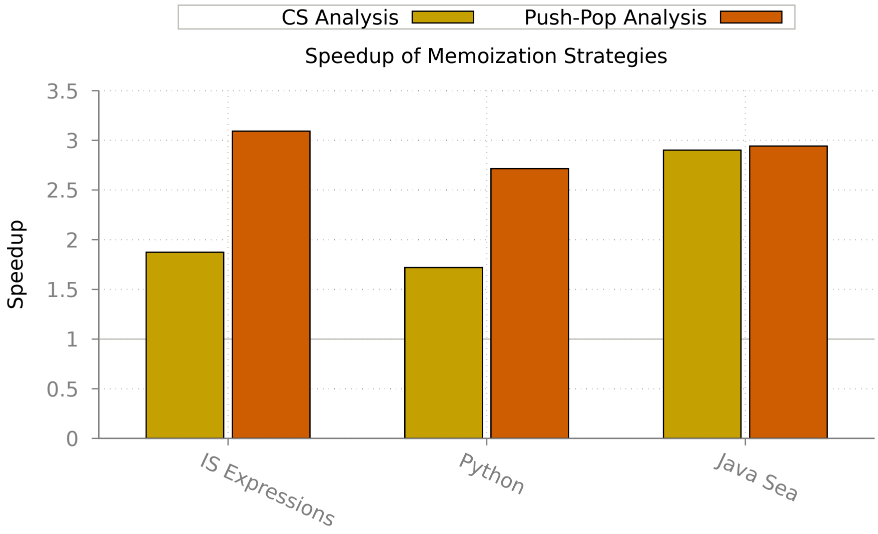


Figure F.32: Impact of the *context-sensitive* analysis and the *push–pop* analysis on the performance of context-sensitive parsers.

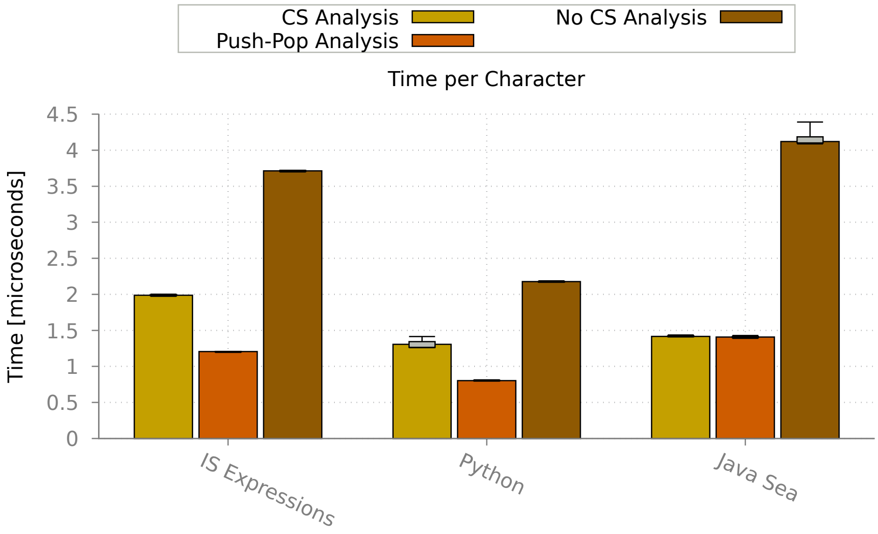


Figure F.33: Time per character of context-sensitive parsers with different context analyses.

F.5 Smalltalk Parsers

Parse time of different Smalltalk parsers compared to the parser compiled by PetitParser compiler is in Figure F.34 and time per character of each of the parsers in Figure F.35.

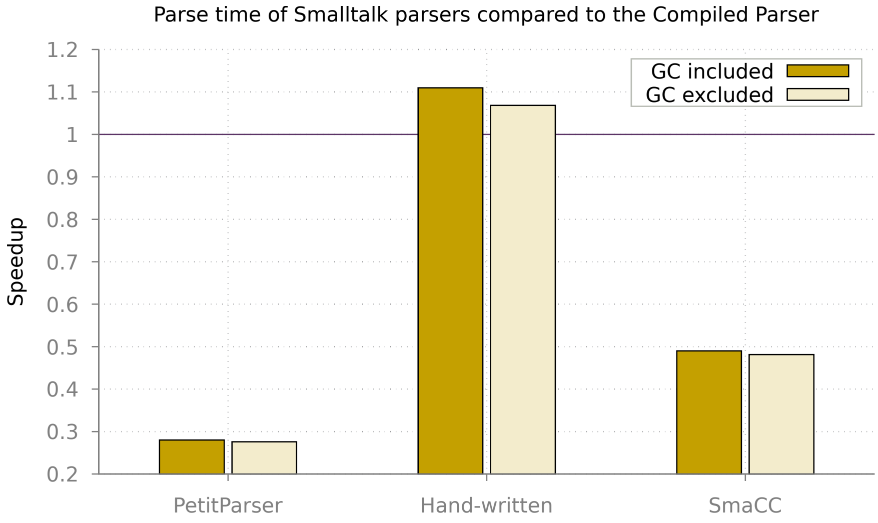


Figure F.34: Performance speedup of Smalltalk parsers.

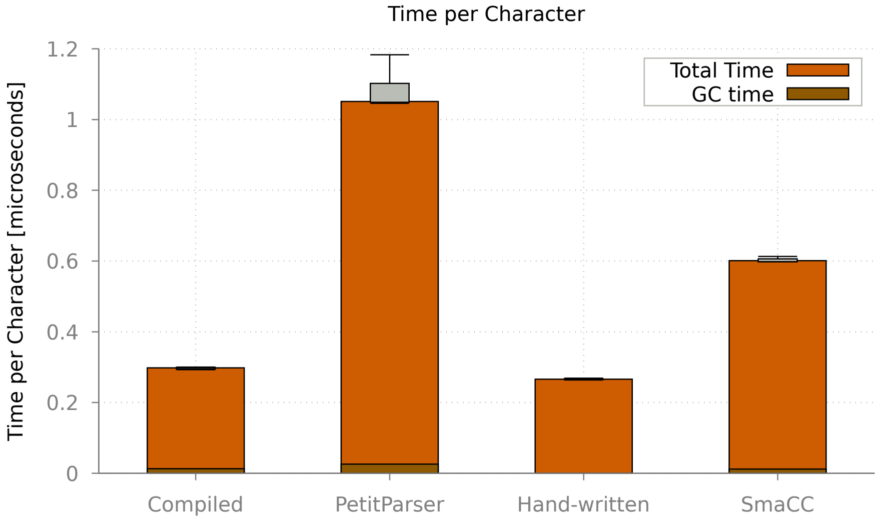


Figure F.35: Time per character of Smalltalk parsers.

F.6 Java Parsers

Parse time of different Java semi-parsers compared to the full Java parser is in Figure F.34 and time per character of each of the parsers in Figure F.35.

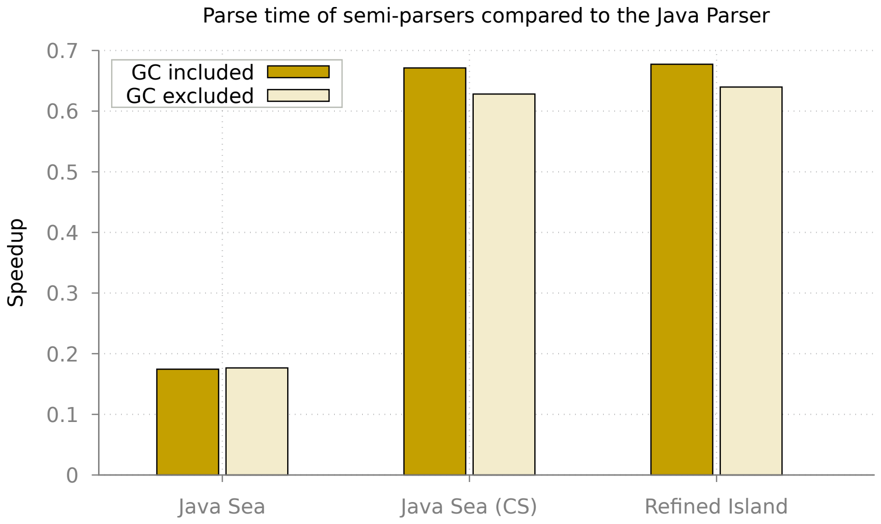


Figure F.36: Performance of Java semi-parsers compared to the Java parser.

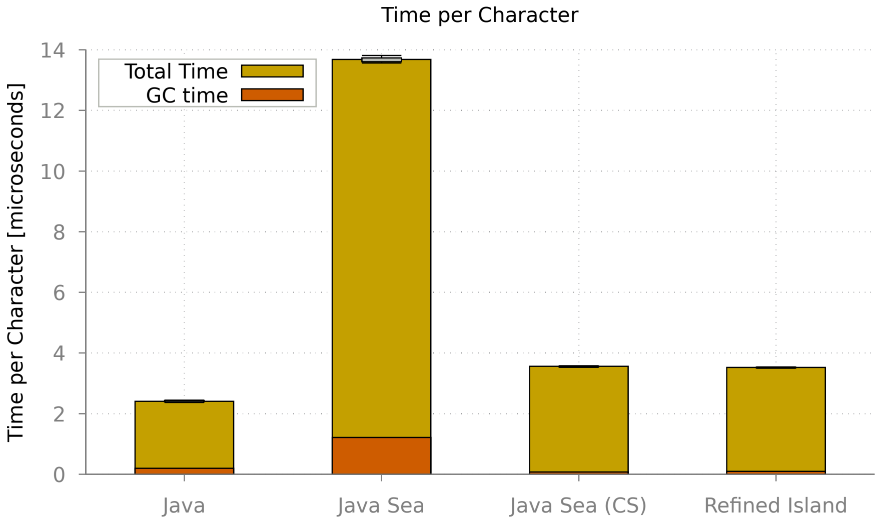


Figure F.37: Time per character of Java semi-parsers and the Java parser.

G

CommonMark Grammar Definition

In the following text we describe a slightly simplified grammar of a CommonMark 0.19 structure.[1] The complete grammar with tests can be found in the prepared Smalltalk image.[2] The parsing of a CommonMark is separated into two phases; the first phase recognizes structure elements and the second phase recognizes inline elements. We present a context- and layout-sensitive grammar of structure elements.

The document structure of CommanMark code is in Listing G.1. The `prefix` uses the `prefix` parser combinator as defined in Algorithm 4.5. There are several types of content elements, such as rules, quoted blocks, code, lists, *etc.* The content of these elements is valid as long as a line starts with the given prefix that is stored in the S_P stack.

The definition of a horizontal rule element is in Listing G.2. The rule definition is straightforward; it expects three or more stars, minuses or underlines at the beginning of a line. The horizontal rule (as any other content element) consumes the final line end so that a new content element can at a new line.

The quoted block of CommonMark changes prefix stack. If a line starts with `'>'` it is pushed to the stack S_P. In the end, the `'>'` is popped from S_P.

There are two types of code block in CommonMark; indented and fenced (see Listing G.4). The indented code starts with four spaces. The fenced start with some indent and is fenced by a sequence of three or more ``'`'`` or `'˜'`. An example of indented code looks like this:

```
␣␣␣␣I am a code.
␣␣␣␣So am I.
```

The indented code definition (`indentedCode`) is straightforward and consists of a lines that are prefixed with four spaces.

The fenced code is more complicated. The problem is that a parser needs to remember (i) the initial indentation (see the rule `codeFenceFirstIndent`, which is

[1] http://spec.commonmark.org/0.19/

[2] http://scg.unibe.ch/research/parsingForAgileModeling.

```
start           ← document
document        ← (prefix contentElem)*
prefix          ← pref_P
contentElem     ← horizontalRule /
                  quoteBlock /
                  code /
                  list /
                  htmlBlock /
                  header /
                  linkRefDef /
                  paragraph /
                  emptyLine

content         ← contentElement
                  (prefix contentElement)*
```

Listing G.1: A PEG definition of a CommonMark document.

```
horizontalRule  ← (stars / minuses / unders) lineEnd
stars           ← '***' '*'*
minuses         ← '---' '-'*
minuses         ← '___' '_'*
```

Listing G.2: A PEG definition of a CommonMark horizontal rule.

```
quoteBlock      ← quoteIndent
                   content
                  quoteDedent

quoteIndent     ← ∇_P(quote)
quoteDedent     ← !prefix △_P(quote)
quote           ← '>'
```

Listing G.3: A PEG definition of CommonMark quote blocks.

removed from the content (consumed by `codeFenceIndent`); and (ii) the fence to properly detect end of fenced code, (see `codeFenceStart` and `codeFenceStop`). As an example, consider fenced code with `'    I am a code'`:

~~~~
```
␣␣␣␣I am a code!
```
~~~~

And this is fenced code with `'I am a code'`. The leading whitespace is omitted because of the initial indentation:

~~~~~
␣␣␣````
␣␣␣I am a code!
␣␣␣````
~~~~~

Therefore two different stacks are used: S_F for fence (see rules `codeFenceStart` and `codeFenceStop`) and S_{FI} for fenced code indentation (see rules `codeFenceFirstIndent`) and `codeFenceIndent`).

````
code                ←  indentedCode / fencedCode

indentedCode        ←  codeIndent codeLine lineEnd
                       (prefix codeIndent codeLine lineEnd)*
codeIndent          ←  #space #space #space #space
codeLine            ←  (!lineEnd •)*

fencedCode          ←  codeFenceStart infoString? lineEnd
                       (
                         !(prefix codeFenceStop)
                          (prefix codeFenceIndent codeLine lineEnd
                       )*
                       (prefix codeFenceStop lineEnd) / !prefix

codeFenceStart      ←  codeFenceFirstIndent
                       ∇F('~~~' '~'*) / ('```' '`'*)
codeFenceStop       ←  prefF &lineEnd ΔF
codeFenceFirstIndent←  ∇FI(#space? #space? #space?)
codeFenceIndent     ←  prefFI

infoString          ←  (!(lineEnd / fenceStop) •)+
````

Listing G.4: A PEG definition of CommonMark code elements.

Lists in CommonMark can be ordered or unordered (see Listing G.5). The items of unordered lists start with `'-'`, `'*'` or `'+'` bullets. The items of ordered ones start with a bullet consisting of a number followed by either `'.'` or `')'`. After an item bullet there can be up to four spaces which should be part of the line prefix (represented by the `prefix` rule). To verify that all the list items have the same bullet formatting, we use an additional stack S_L, which remembers what kind of item bullets are used (see `listBegin` and `listEnd`).

However a numbered list item can start with an arbitrary number; there is possibly infinite number of strings that can start a numbered list item:

```
1.   first list item
3.   second list item
124. third list item
```

In order to ensure that a list item starts with an expected expression, we store the expression to S_L (see `listOrderedMark`). Because we don't store a string, but an expression, we create a special combinator `parse` that parses the expression on the top of the referenced stack and returns its result (see `listMark`).

Note that the continuation line has to be aligned to the text on the first line as in the following example:

```
1. first list item
   continuation of the item
```

```
3.  second list item
    continuation of the item
```

The `'1.  '` or `'3.  '` is consumed by the `listBullet` rule, which pushes to the prefix stack S_P as many spaces as is the length of parsed input (using the `spaces:` method). These spaces become a part of a line prefix. The spaces are popped from S_P in `listItemEnd`.

```
list              ←  listBegin listItem
                         (prefix listItem / listEmptyItem)*
                      listEnd
listBegin         ← ∇L(&(listOrderedMark / listBulletMark))
listItem          ← listBullet content listItemEnd
listEnd           ← △L

listBulletMark    ← '-' map: [:res | '-' ] /
                    '*' map: [:res | '*' ] /
                    '+' map: [:res | '+' ]
listOrderedMark   ← listMarkDot     map: [:res | listMarkDot ] /
                    listMarkBracket map: [:res | listMarkBracket ]
listMarkDot       ← [0-9]+ '.'
listMarkBracket   ← [0-9]+ ')'

                    // push as many spaces as the size of result
listBullet        ← ∇P(listMark maxFourSpaces
                         map: [:r | self spaces: r size ])
listItemEnd       ← △P
listMark          ←  parseL
```

Listing G.5: A PEG definition of CommonMark lists.

HTML blocks consist of properly formated HTML code (see Listing G.6). However the line prefix has to be removed from, therefore `htmlBlock` parses its content line by line while removing the prefix using the `prefix` rule.

```
htmlBlock         ← &htmlTag htmlBlockLine lineEnd
                    (prefix htmlBlockLine lineEnd)*
htmlBlockLine     ← (!lineEnd •)*
htmlTag           ← '<' htmlTagContent '>'
htmlTagContent    ← ...
```

Listing G.6: A PEG definition of a CommonMark HTML block

There are two kinds of headers, `ATXHeader` or `setextHeader` (see Listing G.7). `ATXHeader` has six different levels, `setextHeader` has two different levels.

Link references (see Listing G.8) consists of a label, a destination and an optional title.

A paragraph is defined as in Listing G.9. The content of a paragraph is guarded by a `notAParagraph` rule which ensures that the paragraph does not consume unwanted

```
header           ← ATXHeader / setextHeader
ATXHeader        ← ATXHeaderL1 /
                   ATXHeaderL2 /
                   ATXHeaderL3 /
                   ATXHeaderL4 /
                   ATXHeaderL5 /
                   ATXHeaderL6

ATXHeaderL1      ← ’#’ ATXTitle ATXEnd
ATXHeaderL2      ← ’##’ ATXTitle ATXEnd
ATXHeaderL3      ← ’###’ ATXTitle ATXEnd
ATXHeaderL4      ← ’####’ ATXTitle ATXEnd
ATXHeaderL5      ← ’#####’ ATXTitle ATXEnd
ATXHeaderL6      ← ’######’ ATXTitle ATXEnd

ATXTitle         ← (!ATXEnd •)*
ATXEnd           ← (space ’#’+) lineEnd

setextHeader     ← setextHeaderL1 /
                   setextHeaderL2
setextHeaderL1   ← !emptyLine setexLine lineEnd setexHeaderU1
setextHeaderL2   ← !emptyLine setexLine lineEnd setexHeaderU2
setextLine       ← (!lineEnd •)*
setextHeaderU1   ← ’=’+
setextHeaderU1   ← ’-’+
```

Listing G.7: A PEG definition of a CommonMark header.

```
linkRefDef       ← linkLabel ’:’ linkDestination
                      linkTitle? lineEnd
linkLabel        ← ’[’ (!’]’ (’\]’ / •))* ’]’
linkDestination  ← ’<’ (!’>’ •)* ’>’ /
                   ’(’ (!’)’ •)* ’)’
linkTitle        ← ’"’ (!’"’ •)* ’"’ /
                   ’(’ (!’)’ •)* ’)’
```

Listing G.8: A PEG definition of a CommonMark link reference definition.

content. Furthermore, because paragraphs have relaxed rules for the line prefix, the content of a paragraph can start either with `prefix` or with `lazyPrefix`.

Finally there are the helper rules for empty lines, whitespace *etc.* in Listing G.10.

```
paragraph       ← !emptyLine
                       paragraphLine lineEnd
                  (prefix / lazyPrefix) notAParagraph
                       paragraphLine lineEnd
paragraphLine   ← (!lineEnd •)*
notAParagraph   ← !(emptyLine /
                    ATXHeader /
                    horozontalRule /
                    fencedCode /
                    htmlBlock /
                    list /
                    quote)
lazyPrefix      ← !(prefix quoteIndent) (quote / space)*
```

Listing G.9: A PEG definition of a CommonMark paragraph.

```
emptyLine       ← space* &lineEnd
lineEnd         ← #newline / #eof
space           ← ' '
maxFourSpaces   ← #space? #space? #space? #space? !#space
```

Listing G.10: A PEG definition of helper rules.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname:	Kurs/Jan
Matrikelnummer:	11-118-908
Studiengang:	Informatik
	Bachelor □ Master □ Dissertation ☑
Titel der Arbeit:	Parsing for Agile Modeling
LeiterIn der Arbeit:	Prof. Dr. Oscar Nierstrasz

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist. Ich gewähre hiermit Einsicht in diese Arbeit.

Ort/Datum

Unterschrift

Curriculum Vitæ

Personal Information

Name	Jan Kurš
Email	kurs.jan@gmail.com

Education

2012–2016	PhD in Computer Science, University of Bern, Switzerland Thesis: *Parsing For Agile Modeling*
2007–2010	MSc in Informatics, Czech Technical University, Czech Republic Thesis: *Introducing XML Schema Data Types into XQuery Interpreter*
2004–2007	BSc in Informatics, Czech Technical University, Czech Republic Thesis: *Test pattern compression done with a help of evolutionary algorithms*

Work Experience

2012–2016	Research Assistant *University of Bern*, Switzerland
2015	Software Engineering Intern *Google, Inc.*, California
2008–2012	Software Engineer *Certicon, a.s.*, Czech Republic
2006–2008	Startup Developer *Hledejceny.cz*, Czech Republic