Science of Computer Programming 161 (2018) 57–88

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Efficient parsing with parser combinators

Jan Kurš [a,*], Jan Vraný [b], Mohammad Ghafari [a], Mircea Lungu [c], Oscar Nierstrasz [a]

[a] *Software Composition Group, University of Bern, Switzerland*
[b] *Software Engineering Group, Czech Technical University, Czech Republic*
[c] *Software Engineering and Architecture Group, University of Groningen, Netherlands*

A R T I C L E I N F O

Article history:
Received 3 January 2017
Received in revised form 12 November 2017
Accepted 1 December 2017
Available online 8 December 2017

Keywords:
Optimizations
Parsing expression grammars
Parser combinators

A B S T R A C T

Parser combinators offer a universal and flexible approach to parsing. They follow the structure of an underlying grammar, are modular, well-structured, easy to maintain, and can recognize a large variety of languages including context-sensitive ones. However, these advantages introduce a noticeable performance overhead mainly because the same powerful parsing algorithm is used to recognize even simple languages. Time-wise, parser combinators cannot compete with parsers generated by well-performing parser generators or optimized hand-written code.

Techniques exist to achieve a linear asymptotic performance of parser combinators, yet there is a significant constant multiplier. The multiplier can be lowered to some degree, but this requires advanced meta-programming techniques, such as staging or macros, that depend heavily on the underlying language technology.

In this work we present a language-agnostic solution. We optimize the performance of parsing combinators with specializations of parsing strategies. For each combinator, we analyze the language parsed by the combinator and choose the most efficient parsing strategy. By adapting a parsing strategy for different parser combinators we achieve performance comparable to that of hand-written or optimized parsers while preserving the advantages of parsers combinators.

1. Introduction

A parser combinator is a higher-order function that takes one or more parsers as input and produces a new parser as its output. Parser combinators [53,39] represent a popular approach to parsing. They are straightforward to construct, readable, modular, well-structured and easy to maintain. Parser combinators are also highly expressive as they can parse not only context-free languages but also some context-sensitive ones (*e.g.*, layout-sensitive languages [24,2]).

Nevertheless, parser combinators at the moment are considered more a technology for prototyping than for actual deployment, since the expressive power of parser combinators comes at the price of less efficiency. A parser combinator uses the full power of a Turing-equivalent formalism to recognize even simple languages that could be recognized by finite state

* Corresponding author.
E-mail address: kurs@inf.unibe.ch (J. Kurš).
URLs: http://www.scg.unibe.ch (J. Kurš), http://www.swing.fit.cvut.cz (J. Vraný), http://www.cs.rug.nl/search (M. Lungu).

https://doi.org/10.1016/j.scico.2017.12.001
0167-6423/© 2017 Elsevier B.V. All rights reserved.

machines or pushdown automata. Consequently, parser combinators cannot reach the peak performance of parser generators [37], hand-written parsers, or optimized code [6] (see section 5).

Meta-programming approaches such as macros [9] and staging [47] have been applied to Scala parser combinators [39] with significant performance improvements [6,28]. In general, these approaches remove composition overhead and intermediate representations. Other approaches attack performance problems using more efficient structures, macros *etc.* (see *Parboiled 2*,[1] *attoparsec*[2] or *FParsec*[3]).

While Scala optimizations rely on the power of the Scala compiler, and other solutions exploit knowledge about the internal implementation, our solution provides optimizations based on the domain knowledge of the parsing formalism, is language-agnostic, and does not rely on specifics of the internal implementation.

In this work we build on our idea of a *parser compiler* [34], which optimizes the parser for a language by using specialized parsing strategies for different parser combinators. A strategy is selected based on the language the given parser combinator parses. Different subsets of a language are matched to different parsing strategies. Each of these strategies fits the best for the given subset. Our approach preserves all the advantages of parser combinators and does not impose any restrictions on their expressiveness.

We choose as a case study the performance of PetitParser [46,31] – a parser combinator framework using the parsing expression grammar (PEG) formalism [16]. As a validation of the ideas presented in this work, we implement a parser combinator compiler (for short, a *parser compiler*): an ahead of time source-to-source translator. This compiler (i) analyzes parser combinators of PetitParser, (ii) chooses the most appropriate parsing strategy for each of them, and (iii) integrates these strategies into a single top-down parser, which is equivalent to the original parser. Based on our measurements covering six different grammars for PetitParser,[4] our parser compiler offers a performance improvement of a factor ranging from two to ten, depending on the grammar. Based on our Smalltalk case study, our approach is only 10% slower than a highly-optimized, hand-written parser.

To summarize, this paper makes the following contributions: i) a discussion of performance bottlenecks of parser combinators, ii) a description of optimization techniques addressing these bottlenecks, and iii) a detailed analysis of their effectiveness.

The paper is organized as follows: We explain the parsing overhead of parser combinators using a concrete example in section 2. In section 3 we introduce a parser compiler, which reduces the existing overheads of parser combinators. In section 4 we describe in detail how we identify and apply different parsing strategies. In section 5 we analyze the performance impact of different parsing strategies. In section 6 we briefly discuss related work and finally, section 7 concludes this paper.

2. Motivating example

In this section, we present, as an example, the parsing overhead of PetitParser. PetitParser [46,31] is a parser combinator framework [24] that uses packrat parsing [15], scannerless parsing [52], and parsing expression grammars (PEGs) [16].

We identify the most critical performance bottlenecks of PetitParser and explain them using an example with a grammar describing a simple programming language as shown in Listing 1 (we use a simplified version of the actual PetitParser DSL as described in detail in Table B.2).

A program conforming to this grammar consists of a non-empty sequence of classes. A class starts with `classToken`, followed by an `idToken` and `body`. The `classToken` rule is a `'class'` keyword that must be followed by a space that is not consumed. This is achieved by using the *and* predicate `&` followed by `#space`, which expects a space or a tabulator. Identifiers start with a letter followed by any number of letters or digits. The class keyword and identifiers are transformed into instances of `Token`, which maintain 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.

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.*, column strictly smaller than the previous line. The `indent` and `dedent` rules are represented by specialized action parsers that manipulate an indentation stack by pushing and popping the current indentation level, similarly to the scanner of Python.[5] The class body contains a sequence of classes and methods.

[1] http://www.webcitation.org/6k6195CiS.
[2] http://www.webcitation.org/6k61DC2EA.
[3] http://www.webcitation.org/6k61HcnHU.
[4] Expressions, Smalltalk, Java, Ruby, and Python.
[5] http://www.webcitation.org/6k637RJ7V.

```
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
               ]
methodToken ← ('method' &#space) token
method      ← methodToken  idToken  body
         map: [:methodToken :idToken :body |
                 MethodNode new
                   name: idToken value;
                   body: body
               ]
body           ← #indent
                     (class / method)*
                 #dedent
program      ← class+
```

Listing 1: Example grammar defined in a simplified version of the PetitParser DSL. More details about the syntax are in Appendix B.

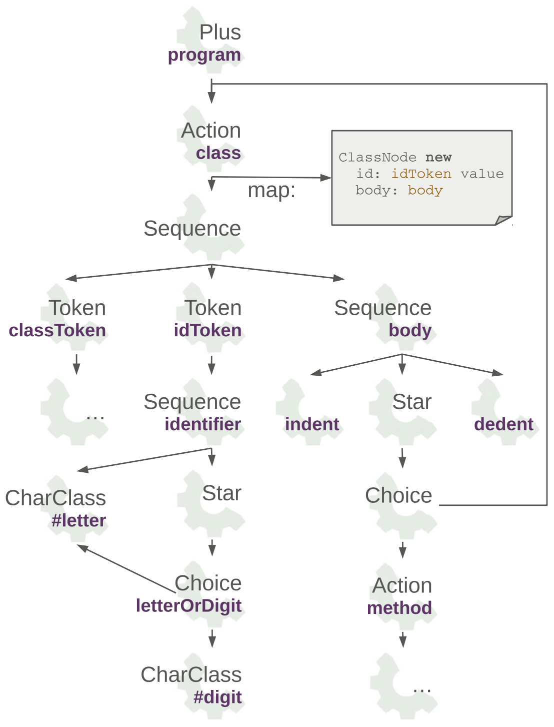

Fig. 1. The structure of parser combinators created after evaluating the code of Listing 1.

2.1. *Parser combinators in PetitParser*

Fig. 1 shows a composition of parser combinators that are created after evaluating the code in Listing 1 in PetitParser. This composition is created *"ahead-of-time"* before a parse attempt takes place and can be reused for multiple parse attempts.

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 discard 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 Fig. 1). The `identifier` rule is a sequence of `CharClass` and `Star` parsers. The `Choice` parser `letterOrDigit` shares the `CharClass` parser with its `identifier` grand-parent.

The `body` rule is a sequence of `Indent`, `Star` and `Dedent` combinators. The `class` combinator is shared by `program` and `body`. The structure of `method` has been omitted.

All the parser combinators share the same interface, `parseOn: context`. The `context` parameter provides access to the input stream and to other information (*e.g.*, indentation stack). The result of `parseOn: context` is either `Failure` or any other output. In case of failure, it is the responsibility of the combinator to restore the context. PetitParser combinators can be easily implemented by following this contract.

The `identifier` rule would be implemented in PetitParser by the following method[6]:

```
identifier
  ↑ letter letterOrDigit⋆
```

This code snippet illustrates three aspects of defining combinators in PetitParser: (i) there is one method per grammar rule, (ii) combinators are objects from a set of classes of which many useful ones are predefined (*e.g.*, `⋆`), and (iii) combinators are composed via a DSL that is inspired from standard PEG syntax.[7]

Appendix B presents more detail about how PetitParser handles context-sensitive grammars, how are the parsers invoked, how backtracking works, and how PetitParser handles lexical elements as these are important for the subsequent performance analysis. In the remainder of this section we discuss four kinds of overhead caused by the parser combinators of PetitParser.

2.2. *Composition overhead*

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

For example, consider the grammar shown in Listing 1 and `letterOrDigit∗` in `identifier`. This can be implemented as a simple loop, but with parser combinators many methods are called. For each character, the following parsers are invoked: A `Star` parser (see Listing B.9), a `Choice` parser (see Listing B.5) and two `CharClass` parsers (see Listing B.4). Each of these parsers contains at least five lines of code, averaging twenty lines of code per character.

The same situation can be observed when parsing `&#space`. `AndPredicate` (see Listing B.3) and `CharClass` (see Listing B.4) 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. All this happens even though the result could be determined with a single comparison to the next character in the input stream.

2.3. *Superfluous intermediate objects*

Superfluous intermediate objects are allocated when an intermediate object is created but not used.

For example, consider an input `'SCP'` parsed by `idToken`. The return value of `idToken` is a `Token` object containing `'SCP'` as a value and `1` and `4` as start and end positions. Yet before a `Token` is created, a `Star` parser (see Listing B.9) and a `Sequence` parser (see Listing B.8) create intermediate collections resulting in `(S,(C,P))` nested arrays that are later flattened into `'SCP'` in `TokenParser` (see Listing B.10) again. Furthermore `Sequence` creates a memento that is never used because the second part of the `identifier` sequence (*i.e.*, `letterOrDigit∗`) never fails.[8]

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 B.2).

2.4. *Backtracking overhead*

Backtracking overhead arises when a parser enters a choice option that is predestined (based on the next k tokens) to fail. Before the failure, intermediate structures, mementos and failures are created and the effort spent is wasted.

Consider an input `'123'` and the `idToken` rule, which starts only with a letter. Before a failure, the following parsers are invoked: `TokenParser` (see Listing B.10), `Sequence` (see Listing B.8), `CharClass` (see Listing B.4). Furthermore, as `TokenParser` tries to consume a whitespace (*i.e.*, by using `#space∗`), another `Star` (see Listing B.9) and

6 Please note, Smalltalk code is shown with a white background.

7 See Appendix B for details.

8 Zero repetitions are allowed, which means that empty strings are also valid, see A.2.2.

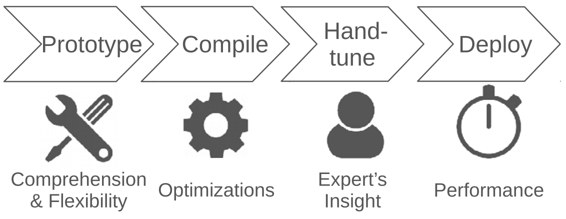

Fig. 2. In the prototype phase a language engineer uses the advantages of parser combinators that are later compiled to a high performance parser, possibly tuned by the engineer and deployed.

`CharClass` (see Listing B.4) 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 `class/method`, which for clarity reasons can be expanded to:

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

The choice always invokes the first option and underlying combinators before the second option. In some cases, *e.g.*, for input `'method bark'` based on the first character of the input, it is valid to invoke the second option 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 options creating superfluous intermediate collections, mementos and failures before it actually fails.

2.5. Context-sensitivity overhead

In the case of PetitParser, most of the context-sensitivity overhead is produced when a parsing context contains complex objects (*e.g.*, an indentation stack). When remembering or restoring contexts, a parser combinator deep copies the whole parsing context [33] (see implementation in Listing B.11). A deep copy is, however, needed only in some cases, *e.g.*, in the case of a `body` sequence where `indent` modifies the stack (see Listing B.6): if any subsequent rule in `body` fails, the original stack has to be restored. In some other cases, *e.g.*, in the case of the `identifier` sequence where none of the underlying combinators modify the indentation stack, the deep copy of the context is superfluous.

3. A parser combinator compiler

The goal of a parser compiler is to generate a high-performance parser from a parser combinator while preserving all the advantages of parser combinators. To achieve this 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.

Fig. 2 shows the work-flow of parser development with the parser compiler we present. First, the flexibility and ease of expression of combinators is used by a language engineer to define a grammar (*prototype phase*). Then, the parser compiler analyzes the grammar to choose the most suitable parsing strategies, builds a top-down parser (*compile phase*), and allows an expert to further modify the compiled parser at their will to further improve the performance (*hand-tune phase*). In the end, the resulting parser can be deployed as an ordinary class to be used for parsing with peak performance (*deployment phase*).

3.1. Adaptable strategies

As mentioned in section 2 there are four different kinds of overhead: (i) composition overhead, (ii) superfluous intermediate objects, (iii) backtracking overhead, and (iv) context-sensitivity overhead. We introduce several strategies that can significantly reduce these four kinds of overhead.

The parser compiler makes use of two different, interleaved modes: a scanning mode, in which tokens are recognized by a dedicated tokenizer, and a scannerless mode, in which tokens are recognized by the parser itself. A parser compiler in a scanning mode uses an adaptable scanner (see Section 3.2) to parse regular parsing expressions (see Appendix A.1) and to guide choices to choose the correct alternative based on the next token. The scanning mode provides better performance but cannot be used for all expressions, because not all of them are scannable. In case a scanner cannot be used, a parser compiler falls back to a scannerless mode with character-based alternatives. Furthermore, context-free expressions are optimized to reduce the overhead of remembering and restoring contexts. In the following we describe each of the strategies. An overview of the strategies for scannerless and scanning modes can be found in Figs. 3 and 4, respectively.

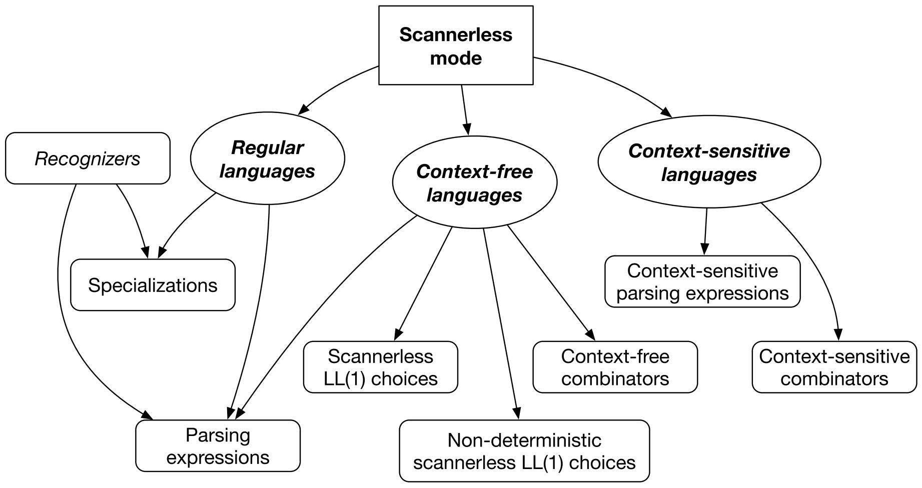
Scannerless mode
Recognizers
Regular languages
Context-free languages
Context-sensitive languages
Specializations
Context-sensitive parsing expressions
Scannerless LL(1) choices
Context-free combinators
Context-sensitive combinators
Parsing expressions
Non-deterministic scannerless LL(1) choices

Fig. 3. Classification of parsing strategies applied in a scannerless mode.

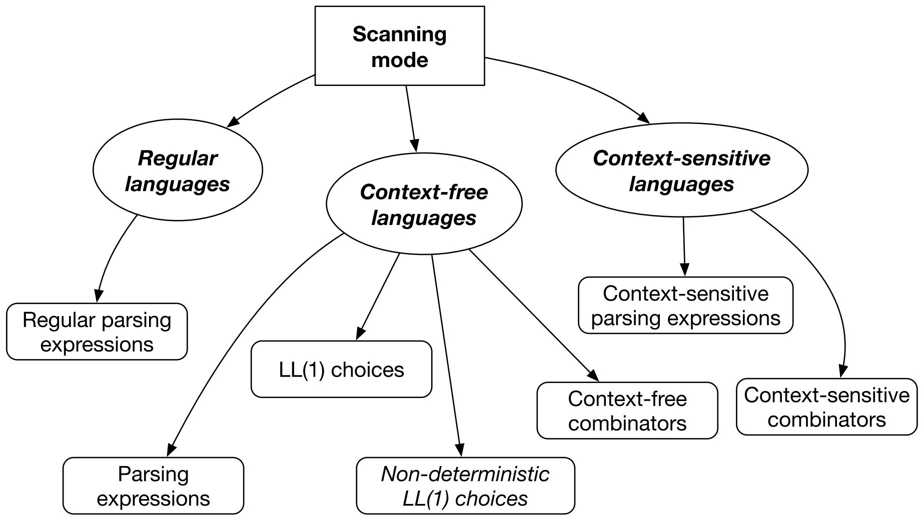
Scanning mode
Regular languages
Context-free languages
Context-sensitive languages
Context-sensitive parsing expressions
Regular parsing expressions
LL(1) choices
Context-free combinators
Context-sensitive combinators
Parsing expressions
Non-deterministic LL(1) choices

Fig. 4. Classification of parsing strategies applied in a scanning mode.

Combinators serve as a fallback option. Whenever a parser compiler does not identify a suitable strategy, the original combinator is used. This ensures that the parser compiler does not impose any restrictions and works even for unknown combinators.

Context-Sensitive Parsing Expressions (CS-PEs) reduce the overhead of composition, because they perform loop unrolling [4] of choices (see the implementation in Listing B.5) and sequences (see the implementation in Listing B.8).

Parsing Expressions (PEs) reduce composition overhead of context-sensitive combinators and context-sensitivity overhead by (i) performing loop unrolling, and (ii) using only context-free mementos instead of more complex context-sensitive ones.

Nondeterministic choices reduce backtracking overhead by rejecting an alternative based on the next character (or token) of input. They are not deterministic because even if an alternative is rejected, it is not clear which of the remaining alternatives should be selected.

Deterministic choices reduce backtracking overhead by choosing 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 is more efficient than interactions of combinators.

Recognizers reduce superfluous intermediate allocations by avoiding intermediate representations. Recognizers return `true` or `false` instead of a parse tree. This improves performance since new objects do not need to be allocated, object initialization methods do not need to be run, and the garbage collector has less work to do [54].

Different strategies target different kinds of languages. In the scannerless mode, combinators describing regular languages are optimized with the help of specializations, parsing expressions and recognizers. In the scanning mode they are handled by regular parsing expressions. The recognizers strategy of a scannerless mode can be combined with either specializations or 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. We are not aware of many opportunities to optimize combinators describing context-sensitive restrictions, although some combinators can be optimized using context-sensitive parsing expressions.

3.2. Adaptable scanner

An *adaptable scanner* is a component of a parser that is responsible for consuming input. Contrary to the traditional pipeline in which scanning and parsing are performed in separate phases (*e.g.*, Lex and YaCC [36,26]), an adaptable scanner is directly invoked by the parser. An adaptable scanner can be used to parse scannerless grammars [52] such as SDF [51] or PEGs [16]. Scannerless grammars are especially useful for describing languages with embedded sublanguages (*e.g.*, embedded SQL or regular expressions). An adaptable scanner has to adapt to the rule from which it is invoked, because in scannerless grammars the same string can represent different tokens for different sublanguages. To achieve such adaptability, the scanner must have multiple scan methods, one for each grammar nonterminal. The method is based on first set analysis [44,19] of the given nonterminal (see subsections 4.2.1 and 4.2.2).

Memoization In order to integrate with the unlimited lookahead of PEGs, the scanner has backtracking capabilities. In a PEG parser, the state is a position in a stream, and in PetitParser, the state is a deep copy of a context (including the position). The adaptable scanner adds information about the current token and its value to the context. When parsing in scanning mode, the context (including the current token and its value) is remembered and is restored to its original value in case of failure.

4. Parser optimizations

Parser combinators form a graph with cycles (see Fig. 1 and Appendix B for more detail). The 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 generator that produces code in the host language, *i.e.*, for the Pharo version of PetitParser, the code generator produces Pharo Smalltalk code. There also exist Java[9] and Dart[10] implementations of PetitParser. To the best of our knowledge, there is no obstacle to implementing parser compiler for these implementations of PetitParser.

Code generation results in a class where each method represents a combinator in the modified graph of parser combinators. In the case of PetitParser, the class contains two instance variables: `context` and (if applicable) `scanner`. The `context` variable keeps the value of a context argument: the input parameter to the parser combinator (see the contract of a `parseOn:` method in Appendix B). The `scanner` is an adaptable scanner as described in subsection 3.2.

The intermediate representation of a parser compiler is high-level, directly representing the target domain, and thus allowing for domain-specific optimizations. For example, a parser compiler can directly check if an expression is nullable or if a choice is deterministic. This would be difficult with lower-level representations used for performance optimizations (*e.g.*, AST [1], Bytecode, SSA [11], LLVM IR [35]).

The following section describes the optimizations in detail. We use the following syntax for rewriting rules: The class of a combinator is given in angle brackets `<>`, *e.g.*, all the character class combinators are marked as `<CharClass>`. Any parser combinator is `<Any>`. We denote a single-step derivation of a parent `P` to a child `C` as `P→C`. A parser combinator that is a (recursive) descendant `D` of a parent `P` is marked `P→*D`. 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 1.

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

Delegating parsers embed the parser that 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

9 http://www.webcitation.org/6k62uAxHz.
10 http://www.webcitation.org/6k62vRsWA.

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

Listing 2: The code produced from the `letterOrDigitStar` repetition after applying the specialization optimization.

to re-use the PEG syntax, *e.g.*, `<Any> <Any>` is also a sequence of two arbitrary combinators. The rewrite operation is ⇒. As an example, merging a choice of two character classes into a single one is written as:

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

4.1. Regular optimizations

Regular optimizations are performed at the level of tokens, *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 (see Appendix A.1).

4.1.1. Regular parsing expressions

Regular parsing expressions (RPEs) (see Appendix A.1) 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, and those recognized as regular expressions are marked `:regular`. All regular expressions are wrapped with a `<Scanner>` combinator.

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

A `<Scanner>` combinator represents a scan method in an adaptable scanner that uses a 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
```

4.1.2. 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 2.2 (let us suppose that a scanner is not used), 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* = <CharClass[a-zA-Z]>/<CharClass[0-9]>*
    ⇒ <CharClass[a-zA-Z0-9]>*
    ⇒ <CharClassStar[a-zA-Z0-9]>
```

In the final phase, the code generator produces the code as shown in Listing 2, which contains only three lines of code per consumed character.

Returning to the problem with `&#space` in subsection 2.2, 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 code as shown in Listing 3, which does not create any mementos and does not invoke any extra methods.

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

Listing 3: The code produced from the `&#space` character class after applying the specialization optimization.

```
letterOrDigitStar
  //no need to create an intermediate collection
  [context peek isLetter or:
  [context peek isDigit]] whileTrue: [
    context next.
  ].
  ↑ true
```

Listing 4: The code produced from the `letterOrDigitStar` rule after applying the recognizer optimization.

We implement several similar specializations, including the following:

```
<CharClass> / <CharClass> ⇒ <CharClass>
<CharClass> negate ⇒ <CharClass>
<CharClass>*   ⇒ <CharClassStar>
<CharClass>+   ⇒ <CharClassPlus>
&<CharClass>   ⇒ <AndCharClass>
!<CharClass>   ⇒ <NotCharClass>
&<Literal>     ⇒ <AndLiteral>
!<Literal>     ⇒ <NotLiteral>
<CharClass> token   ⇒ <TokenCharClass>
<CharClass>?   ⇒ <OptionalCharClass>
```

Note that the `OptionalCharClass` combinator can return directly `nil` without intermediate failure.

4.1.3. *Recognizers*

A recognizer simply recognizes a sublanguage, returning `true` or `false` rather than a parse tree. In cases where parse trees are not needed, we can replace parsers by recognizers and thus avoid superfluous object allocations (*i.e.*, unneeded parse trees). Combinators forming a `Token` parser are marked to avoid generating intermediate representations (`:recognizer`) because tokens do not have an internal structure (as described in Appendix B). The token itself can be created as a substring of the input stream, where start and end positions are positions before and after parsing the given token.

The same strategy of avoiding creation of intermediate objects is used also for and `&` and not `!` predicates, which return only `true` or `false` and discard their results.

The following reduction rules are used to illustrate formally the rules for applying this strategy in the three aforementioned cases.

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

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

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

In the final phase, the code generator produces the code as shown in Listing 4.

4.2. *Context-free optimizations*

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

4.2.1. *Deterministic choices*

Deterministic choices limit invocations and allocations caused by backtracking. During a dedicated optimization phase, character-based or token-based *first sets* [44,19] are computed. If all *n* choice alternatives `a`$_1$`/a`$_2$`/.../a`$_n$ have distinct first sets (*i.e.*, their first sets do not overlap) the choice is marked as being deterministic (`:dch`) and choices with the `dch` property are replaced with a deterministic choice combinator.

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

Listing 5: The code produced from the `classOrMethod` rule after analyzing deterministic choices.

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

Listing 6: The code produced from the `classOrMethod` rule after analyzing deterministic choices and using a scanner.

```
<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 code as shown in Listing 5 from this choice (since a class definition starts with `class` and a method with `def`). If a scanner is used, it produces code as in Listing 6.

4.2.2. 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 may still 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 code that fails immediately, without entering the underlying combinator. As an example, let us slightly modify `class` and `method` from Listing 1 to allow for private definitions:

```
classToken   ← 'class' &#space token
defToken     ← 'def' &#space token
method       ← privateToken? defToken idToken ...
class        ← privateToken? classToken  idToken ...
privateToken ← 'private' token
```

In such a case, the alternatives of a `class/method` can be wrapped with `Guard`.

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

In the final phase, the code generator produces the code as in Listing 7 from the choice of `class` and `method`. Alternatively, if a scanner is used, the code generated is as shown in Listing 8.

4.3. Context-sensitive optimizations

Optimizations in this category address (i) performance problems of parser combinators that are too generic, and (ii) superfluous context-sensitive memoizations.

```
classOrMethod
    | result |
    (context peek == $c or: [ context peek == $p]) ifTrue: [
        (result ← self class) isSuccess ifTrue: [
            ↑ result
        ]
    ].
    (context peek == $d or: [ context peek == $p]) ifTrue: [
        (result ← self method) isSuccess ifTrue: [
            ↑ result
        ]
    ].
  ↑ Failure message: 'neither class nor method found'
```

Listing 7: The code produced from the `classOrMethod` choice after applying guard optimizations.

```
classOrMethod
    | result memento |
    memento ← scanner remember.
    (scanner guard_privateOrClass) ifTrue: [
        (result ← self class) isSuccess ifTrue: [
            ↑ result
        ]
        scanner restore: memento.
    ].
    (scanner guard_privateOrMethod) ifTrue: [
        (result ← self method) isSuccess ifTrue: [
            ↑ result
        ]
        scanner restore: memento.
    ].
  ↑ Failure message: 'neither class nor method found'
```

Listing 8: The code produced from the `classOrMethod` choice after applying guard optimizations with a scanner.

```
classOrMethod
  | retval |
  (retval ← self class) isSuccess ifTrue: [
    ↑ retval
  ].
  (retval ← self method) isSuccess ifTrue: [
    ↑ retval
  ].
  ↑ Failure message: 'neither class nor method found'
```

Listing 9: The code produced from the `classOrMethod` choice after loop unrolling without any other optimizations.

4.3.1. *Context-sensitive parsing expressions*

In a combinator implementation the children of sequences and choices are called in a loop (see Listing B.8 and Listing B.5, noting that `children do:` is an iterative construct in Smalltalk). The parser compiler improves performance of sequences and choices by loop unrolling.

The choices are simply unrolled; no further analysis is performed. For a choice of two alternatives, *e.g.*, `class/method` (if no optimizations from the previous section are applied), the code without loop, which saves the loop bookkeeping,[11] is generated as in Listing 9.

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 Appendix B).

As an example, consider the `identifier` rule, a sequence of a letter and a letter or digit repetition. If no regular parsing expression optimization has been applied, the code as in Listing 10 can be generated.

[11] By unrolling the loop from Listing B.5.

```
 1 identifier
 2   | memento retval |
 3   memento ← context remember.
 4   retval ← Array new: 2.
 5   result ← self letter.
 6   result isFailure ifTrue: [
 7     context restore: memento.
 8     ↑ result
 9   ].
10   retval at: 1 put: result.
11   result ← self letterOrDigitStar.
12   result isFailure ifTrue: [
13     context restore: memento.
14     ↑ result
15   ].
16   retval at: 2 put: result
17   ↑ retval
```

Listing 10: The code produced from the `identifier` without applying regular parsing expression optimizations.

```
identifier
  | retval |
  retval ← Array new: 2.
  (result ← self letter) isFailure ifTrue: [
    ↑ result
  ].
  retval at: 1 put: result.
  retval at: 2 put: self letterOrDigitStar.
  ↑ retval
```

Listing 11: The code produced from the `identifier` after applying regular parsing expression optimizations.

Such code is rather inefficient. First of all, if `letter` fails, `letter` itself has to restore the context to the point upon its invocation, which is the point when `memento` was created. There is no need to restore (line 7) for the second time. Second of all, `letterOrDigitStar` never fails because it can accept epsilon, *i.e.*, it is nullable, therefore the restore (line 13) is never called. Last but not least, there is no need to create a memento (line 3) because it is not needed at all, and the code as in Listing 11 can be used instead.

To generate such code, 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.[12]

In practice, sequences can have more children. 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>>
```

The rewrite rules for sequences with three children are straightforward:

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

[12] If the first element of a sequence fails, it has to restore the context to its original state (see the `parseOn:` contract in Appendix B), which is the original state of the sequence itself.

```
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 12: The code produced from the `body` sequence after applying the context-sensitive parsing expressions optimization.

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

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 12.

4.3.2. *Context-free memoization*

Context-free memoization reduces the overhead of context-sensitive expressions by turning them into context-free expressions. 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 using grammar rewriting. The other one is tailored to our context-sensitive extension using the push `▽` and pop `△` operators to modify the parsing contexts (full details are in the PhD thesis of Kurš [30]).

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 a grammar modification. If a combinator refers to a context-sensitive combinator, the combinator is marked as context-sensitive as well:

```
<Any>→*<Any:cs> ⇒ <Any:cs>→*<Any:cs>
```

The *push–pop* analysis (see Definition A.9) also traverses the combinators, but takes advantages of well-defined context-manipulation semantics of push `▽` and pop `△`, 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 a consecutive `▽` and `△` result in a context-free expression (in the sense that the expression does not change the context). Furthermore, a sequence can be restored after a 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 1. After `#indent`, which pushes to the indentation stack (see Listing B.6), 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 restored by push, because we have lost the popped item that needs to be pushed back again. The solution we use is to restore from the full context-sensitive memento.

For simplicity, we discuss only one context-sensitive feature in this paper: *indentation*, for which we manage one default stack: *indentation stack*. Indent `▹` corresponds to push `▽` to the indentation stack and dedent `◃` corresponds to pop `△` from the indentation stack. In general, there might be more context-sensitive features and for each of them a separate stack has to be maintained. The *push–pop* analysis is then done for each of the stacks separately, and the `:push` and `:pop` properties have to be used for each of the stacks as well.

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>
```

```
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 13: The code produced from the `body` sequence after applying the context-free memoization optimization.

Even though the `body` sequence contains the context-sensitive rules `#indent` and `#dedent`, no context-sensitive memento is used in the generated code (see Listing 13).

For this optimization, we do not 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 [18], 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.

4.3.3. 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 combinator whose semantics is unknown to it. This might happen when a user of PetitParser has implemented their own extension of PetitParser.

The question is, how to include such an unknown combinator into the translated code so that it effectively interoperates well with the generated parser? Our parser compiler has no other choice but to call the `parseOn:` method of the combinator itself, and let it do whatever it wants. Nevertheless, such a simple approach is not very effective: the combinator itself probably invokes other combinators (its children), which are again (unoptimized) combinators. All the parsing done by the unknown combinator is not optimized at all.

Luckily, we can do better. If possible, the parser compiler inspects and optimizes the children of the unknown combinator. The result of such an optimization consists of new methods (one for each child) in the generated class. In the next step, the parser compiler rewires the links from an unknown combinator to its children and replaces them with links to the optimized methods. The children of the unknown combinator are substituted by a `Bridge` combinator whose `parseOn:` method invokes an method that is an optimized equivalent of the original child.

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 re-define `body` with a new combinator representing the longest choice `||`[13]

```
body ← (class||method)*
```

This definition of `body` is transformed to the following combinator graph (the rules `class` and `method` are compiled as usually):

```
body ← <Unknown<<Bridge<class> <Bridge<method>>*
```

In the final phase, the code generator produces the following code, where the variable `unknown` is an instance of a `LongestChoice` parser:

```
body
    | retval result |
```

[13] *I.e.*, a choice that returns the alternative that consumes the longest part of input.

```
    retval ← OrderedCollection new.
    [result ← unknown parseOn: context] isFailure
        whileFalse: [ retval add: result ].
    ↑ retval
```

By calling `unknown»parseOn:` an implementation of a `LongestChoice` combinator is called. The children of the longest choice are replaced by a `Bridge` combinator that forwards to the compiled code:

```
Bridge>>parseOn: context
    "in our example, the selectors are 'method' or 'class'"
    ↑ compiledParser perform: selector
```

5. Performance analysis

In this section we report on the performance of compiled parsers compared to the performance of plain PetitParser.[14] We also report on the impact of a particular optimization on the overall performance. Moreover, we measure the performance impact of a *push–pop* analysis and scanning as well. Last but not least, we provide a case-study of performance of Smalltalk parsers available in Pharo [7].

5.1. PetitParser compiler

The PetitParser compiler applies the parser compiler techniques described in this paper and outputs a Smalltalk class that serves as a top-down parser equivalent to the input combinator.

PetitParser Compiler is available online[15] for Pharo and Smalltalk/X. It is being already used in two production environments: the language for Live Robot Programming environment[16] and the Pillar markup language.[17]

Validation The PetitParser compiler is covered by more than three thousand unit tests. Furthermore, we validated the parser compiler by taking several existing PetitParser combinators and comparing their results with the results produced by their equivalent compiled variant. In particular, we validated results of four parsers: (i) a Java 6 parser,[18] (ii) a Smalltalk parser,[19] (iii) a Ruby semi-parser,[20] and (iv) a Python semi-parser.[21] The parsers were validated on several open-source projects that were also used in the performance benchmarks.

5.2. Benchmarks

We measure performance on the following benchmarks:

1. **Expressions** is a benchmark measuring performance of arithmetic expressions. Input consists of expressions with operators `(`, `)`, `*`, `+` and integers. The parentheses must be balanced. Operator priorities are considered. The grammar is not in a deterministic form, *i.e.*, it uses unlimited lookahead and backtracks heavily. The parser contains eight rules.
2. **Smalltalk** is a benchmark measuring the performance of a Smalltalk parser. Input consists of the source code from a Pharo 5 image.[22] The parser contains approximately eighty rules.
3. **Java** is a benchmark measuring performance of a Java parser provided by the Moose analysis platform community [40]. We used version 167.[23] Input consists of the standard JDK 6 library files. The parser contains approximately two hundred rules.
4. **Ruby** is a benchmark measuring the performance of the Ruby parser. Input consists of several GitHub Ruby projects.[24] The parser contains approximately forty rules. The parser is not complete. It uses indentation [22], bounded seas [32], and a special version of island parsing to extract modules, classes, methods, method calls, and their receivers.

[14] A replication package containing an image with sources and benchmarks can be downloaded from http://scg.unibe.ch/research/petitcompiler/scp2016.
[15] http://scg.unibe.ch/research/petitcompiler.
[16] http://pleiad.cl/research/software/lrp.
[17] http://smalltalkhub.com/#!/~Pier/Pillar.
[18] http://smalltalkhub.com/#!/~Moose/PetitJava/.
[19] http://smalltalkhub.com/#!/~Moose/PetitParser.
[20] http://smalltalkhub.com/#!/~JanKurs/PetitParser.
[21] Ibid.
[22] http://files.pharo.org/get-files/50/sources.zip.
[23] http://smalltalkhub.com/#!/~Moose/PetitJava/.
[24] Rails, Discourse, Diaspora, Cucumber and Vagrant.

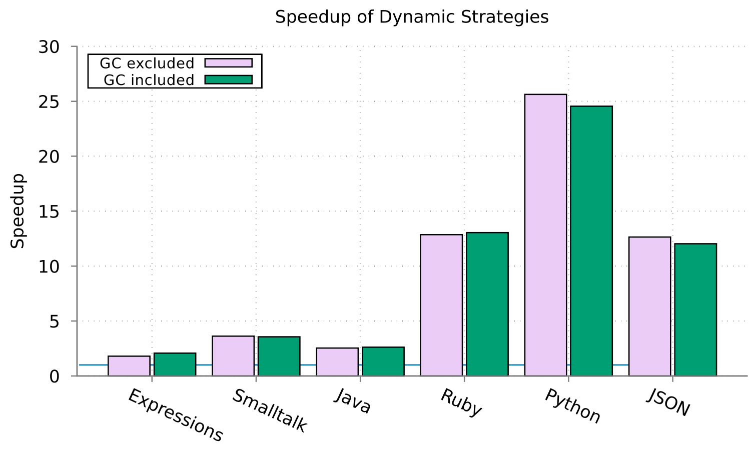

Fig. 5. The speedup of compilation for different grammars.

5. **Python** is a benchmark measuring performance of an indentation-sensitive Python parser. Input consists of several open source Python projects.[25] The parser contains approximately forty rules. The parser is not complete. It uses islands [38] to extract structural elements and skips the rest. The structural elements that are extracted are: classes, methods, `if`, `while`, `for` and `with` statements.
6. **JSON** is a benchmark measuring performance of a standard JSON parser. Input consists of randomly generated JSON files. The parser contains approximately twenty rules.

The presented benchmarks cover a variety of grammars from small ones to complex ones, ranging in size from eight to two hundred grammar rules. They cover grammars with possibly unlimited lookahead (arithmetic expressions) and almost LL(1) grammars (Smalltalk, Java, JSON).[26] They also cover standard grammars (Java, Smalltalk, JSON), island grammars (Python, Ruby), context-free grammars (Java, Smalltalk, JSON), and context-sensitive ones (Python, Ruby).

How we measure We run each benchmark ten times using the release of the Pharo VM for Linux from September 8, 2016. All the parsers and inputs are initialized in advance, and then we measure the time to parse. The input size is set up so that even the fastest benchmark runs at least a second.

We report on speedup (the ratio between the best time original PetitParser serving as a baseline and the best time of its compiled version) and time per character. To estimate the impact of a garbage collector, we report total run time with and without garbage collection included.

Results The speedup of a compiled version compared to an original version is shown in Fig. 5.

The Expressions parser shows a 2× speedup. We attribute this result to the fact that the Expressions parser performs a lot of backtracking and the simplicity of the underlying grammar does not allow many optimizations.

The Smalltalk and Java parsers exhibit a speedup slightly below a factor of four, respectively three. The Python, Ruby and JSON parsers undergo a 10× speedup. The Python and Ruby parsers are context sensitive, and their original versions are very slow (see time per character in Fig. 6). This is caused by the overhead of copying an indentation stack, which is to a great extent removed by a parser compiler. The JSON parser itself is relatively fast (see time per character in Fig. 6), but its performance can be greatly improved, mainly thanks to the tokenization, as we show in subsection 5.3.

5.3. Performance details

In order to provide insight into how particular parsing strategies affect the overall performance, we divide the parsing strategies into three main groups;

1. Regular (RE) – presented in Section 4.1
2. Context-free (CF) – presented in Section 4.2
3. Context-sensitive (CS) – presented in Section 4.3

[25] Django, Tornado and Reddit.
[26] The implementors did not bother to make it LL(1) as parser combinators allow for unlimited lookahead.

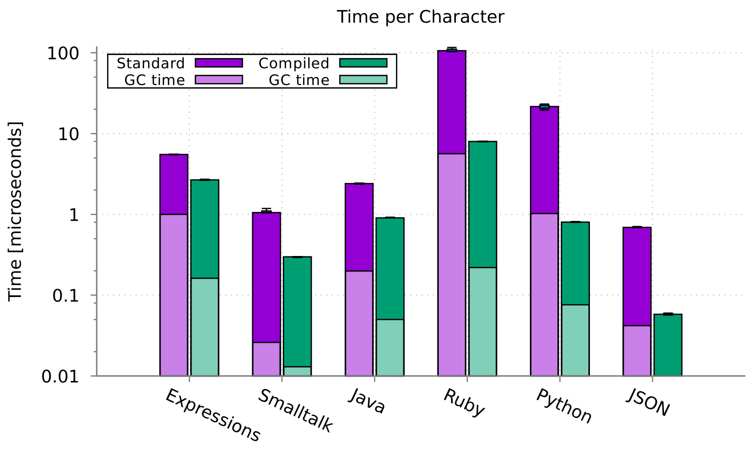

Fig. 6. Time per character of plain PetitParser and its compiled version for different grammars. Please note the logarithmic scale.

We enable different parsing strategies during compilation and report on these. Because of the limitations of PetitParser compiler, some parsing strategies cannot be applied without transforming the parser into the context-sensitive parsing expressions (PE). The context-sensitive parsing expressions are therefore always included.

We use the configurations summarized in the following table:

	PE	RE	CF	CS
PE	●			
PE+RE	●	●		
PE+CF	●		●	
PE+CS	●			●
PE+RE+CF	●	●	●	
PE+RE+CS	●	●		●
PE+CF+CS	●		●	●
All	●	●	●	●

For example, configuration PE transforms combinators of PetitParser to context-sensitive parsing expressions, performing loop unrolling as described in subsubsection 4.3.1, and configuration PE+RE applies all the transformations from subsection 4.1 to context-sensitive parsing expressions (PE).

We briefly summarize the configurations here:

PE+RE If possible regular parsing expressions are used (4.1.1), otherwise specializations (4.1.2) and recognizer strategies (4.1.3) are used.

PE+CF Only character-based deterministic choices (4.2.1) and guards (4.2.2) are used because RE techniques (4.1) are not applied and therefore tokens cannot be detected.

PE+CS Parsing expressions are analyzed with the *push–pop* analysis. For context-sensitive parsing expressions (4.3.1) standard mementos are used and for context-free parsing expressions (4.3.2) context-free mementos are used. For unknown parsing expressions, combinators are used (4.3.3).

PE+RE+CF If possible regular parsing expressions are used (4.1.1), otherwise specializations (4.1.2) and recognizer strategies (4.1.3) are used. Only character or token based deterministic choices (4.2.1) and guards (4.2.2) are used.

PE+RE+CS If possible regular parsing expressions are used (4.1.1), otherwise specializations (4.1.2) and recognizer strategies (4.1.3) are used. For context-sensitive parsing expressions (4.3.1) standard mementos are used and for context-free parsing expressions (4.3.2) context-free mementos are used. For unknown parsing expressions, combinators are used (4.3.3).

PE+CF+CS Only character-based deterministic choices (4.2.1) and guards (4.2.2) are used because RE techniques (4.1) are not applied and therefore tokens cannot be detected. For context-sensitive parsing expressions (4.3.1) standard mementos are used and for context-free parsing expressions (4.3.2) context-free mementos are used. For unknown parsing expressions, combinators are used (4.3.3).

All Applies all the possible optimizations from subsection 4.1, subsection 4.2 and subsection 4.3.

The speedup of a particular configuration is shown in Fig. 7. The graphs illustrate how much a particular configuration contributes to the overall performance of the parsers.

Different strategies have different impact on the parsers. Regular expression strategies (PE+RE), for example, optimize the JSON and Smalltalk parsers well, but they are worse than standalone parsing expressions (PE) in the case of Expressions.

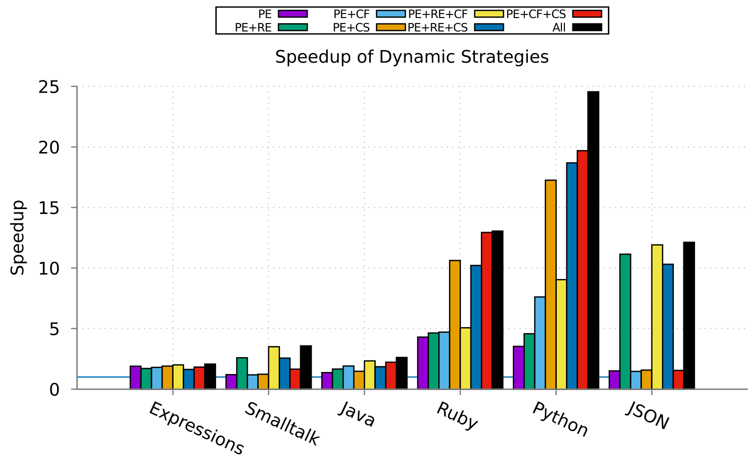

Fig. 7. Speedup against plain PetitParser for different configurations.

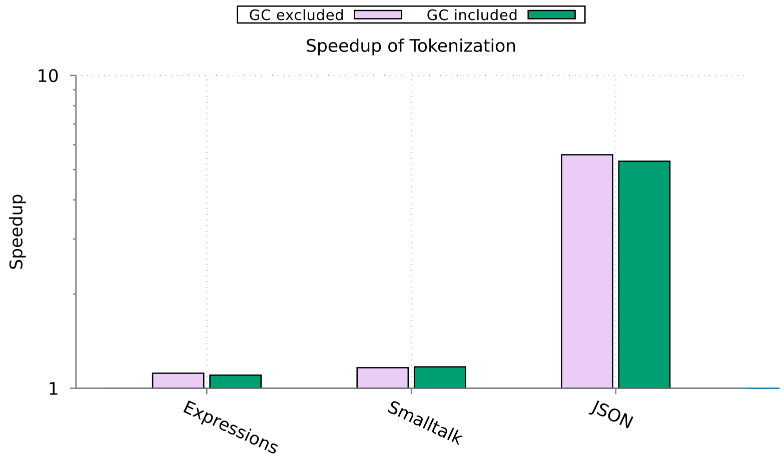

Fig. 8. Speedup of a scanning strategy when applied to the Expressions and Smalltalk parsers.

This is caused by a scanner that provides the best performance when combined with the context-free optimizations as we illustrate in subsection 5.4.

Context-free strategies (PE+CF) affect Python and Java parsers a lot. Context-sensitive strategies (PE+CS) significantly affect the context-sensitive parsers, *i.e.*, Ruby and Python parsers. In other cases context-sensitive strategies slightly improve performance because of more efficient memoization: mementos are integers and not objects.

Combination of strategies brings even better results. The regular and context-free strategies (PE+RE+CF) almost reach the top performance of context-free grammars (Expressions, Smalltalk and Java), while regular and context-sensitive (RE+CS) strategies almost reach the top performance of context-sensitive grammars (Ruby, Python).

5.4. Scanner impact

In order to evaluate the impact of scanning strategies, we compare a parser compiled with the configuration that uses scanning strategies with one that does not (and which serves as a baseline). The speedup using scanning strategies varies greatly from 10% to 600% (see Fig. 8), depending on the grammar.

The current implementation of the scanner limits its usage to the Expressions, Smalltalk, and JSON parsers. Other grammars either do not use regular parsing expressions to consume input (Python and Java), or are context-sensitive (Ruby and Python).

5.5. Memoization impact

The context-sensitive optimizations affect the context-sensitive parsers for Ruby and Python. In order to investigate in detail the impact of the *context-sensitive* analysis and *push–pop* analysis as described in subsection 4.3 we compare the

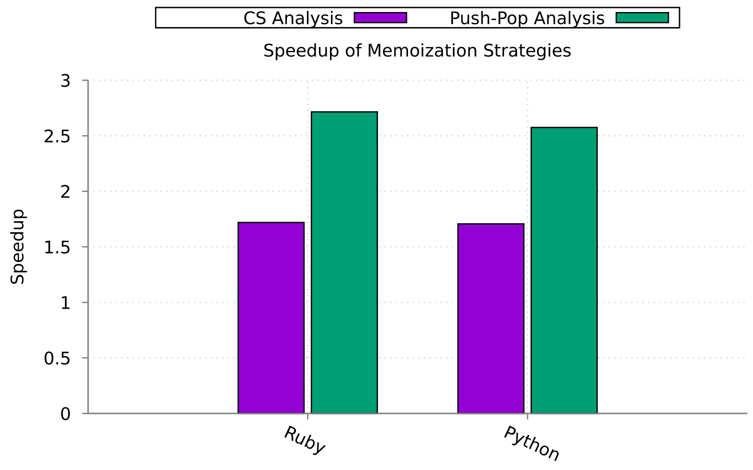

Fig. 9. The impact of *context-sensitive* analysis and *push–pop* analysis on context-sensitive memoization.

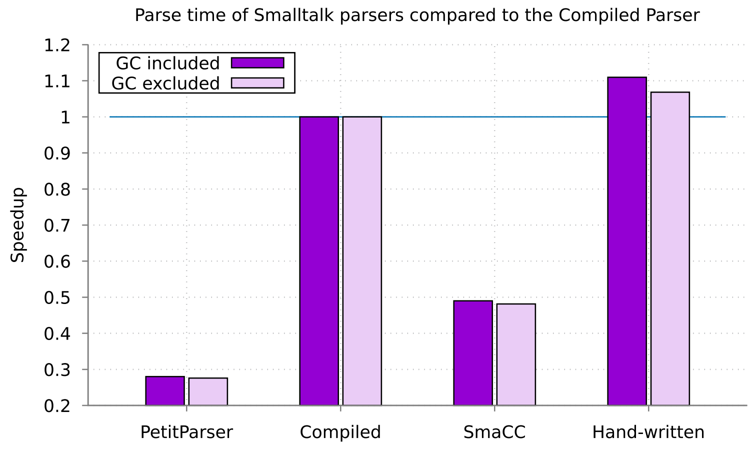

Fig. 10. Performance speedup of Smalltalk parsers.

performance of parsers compiled using both of the 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 Fig. 9. In the case of Ruby and Python, *context-sensitive* analysis shows a speedup of 1.5, and *push–pop* analysis offers a speedup of 2.5.

5.6. Smalltalk case study

In this case study we compare the performance of a Smalltalk parser compiled by a parser compiler (*i.e.*, serving as a baseline) with other implementations of Smalltalk parsers available in the Pharo environment. All of the parsers create an identical abstract syntax tree from the given Smalltalk code:

1. **PetitParser** is an implementation of a Smalltalk parser in PetitParser.
2. **Compiled** PetitParser is a version of the above parser compiled with the PetitParser compiler. This parser serves as a baseline.
3. **SmaCC** is a scanning table-driven parser compiled by a SmaCC [8] framework from an LALR(1) Smalltalk grammar.
4. **Hand-written** parser is a parser used natively by Pharo. It is a hand-written and optimized parser and uses a scanner. We believe it to be 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 Fig. 10. The only parser that can keep up with the compiled version is the hand-written parser, which is approximately 10% faster. The SmaCC parser is approximately two times slower, and the original PetitParser is roughly 3.5 times slower. The native parser's time per character is 0.27 µs, the compiled parser's time is 0.29 µs, the SmaCC parser time is 0.59 µs, while PetitParser's time is 1.33 µs.

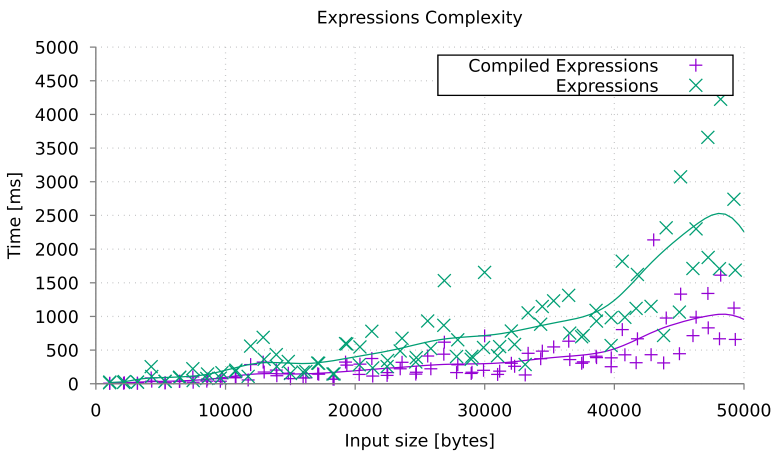

Fig. 11. Time to parser Expressions compared to the input size.

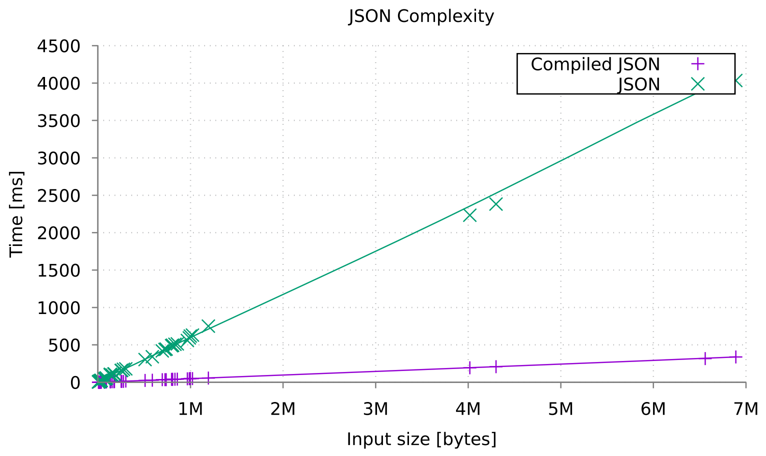

Fig. 12. Time to parse JSON source compared to the input size.

5.7. Time complexity

In this section we report on the complexity of the parser execution time compared to the input size. As the parser compiler does not modify the underlying grammar (it only optimizes the parser execution) the complexity of the parser does not change. The difference is in the constant overhead. This can be seen in (i) Fig. 11, where compiled Expressions[27] simply reduce the exponential complexity by a constant (compared to its original version), and (ii) Fig. 12, where compiled JSON[28] reduces the linear complexity by a constant as well.

6. Discussion and related work

In this section we summarize the limitations of a parser compiler and related technologies, focusing on the performance of parsers.

6.1. Limitations

The idea of adaptable parsing strategies does not impose any limitations on the underlying grammar. The worst case scenario is that a parser is not optimized at all and its performance remains unchanged. This might happen, for example, if all of the parser combinators are created on-the-fly, *i.e.*, there is no combinator graph to be analyzed and optimized before parsing starts, or if none of the combinators can be specialized.

[27] The slowest grammar.

[28] The fastest grammar.

Another limitation arises when a framework user implements a new combinator with unknown semantics. In such a case, the analysis framework has to expect the worst case scenario and use the combinator as is, without optimizations. It is up to the implementor of the new combinator to extend the analyses to properly optimize their combinator.

6.2. Extensibility

Grammar implementors might extend our framework to support their own parser combinators. This include several steps: (i) extend the first/follow analysis; (ii) extend the push–pop analysis; (iii) extend abstract simulation analysis; (iv) add dedicated specializations rules (if applicable) with specialized intermediate nodes; and last but not least (v) write a routine to compile the new combinator to Smalltalk code. In practice, this might be complicated, because the parser compiler architecture has not been especially designed with extensibility in mind.

We consider this to be an engineering issue, with no fundamental obstacles preventing extensibility. In fact, we have started a new engineering project, PetitParser2,[29] building on our experience with the parser compiler, and which allows for such extensibility. The analyses are implemented in the form of visitors, which can be easily extended by overriding the default (universal) implementation. Furthermore, PetitParser2 itself reports unknown nodes and *hotspots* (frequently invoked or memoization intensive combinators). Nevertheless, extending the framework is work for experienced users with good knowledge of PEGs and their analyses.

6.3. Hand-tuning

Our parser compiler accommodates a use case in which a grammar engineer notices a potential for optimization even in the generated code. In such a case, the engineer can manually tune an arbitrary method generated by the parser compiler. The compiler remembers the modified method and does not override the changes made by the engineer. The parser compiler provides basic tooling to identify frequently executed methods, so that engineers know on which methods they should focus.

The disadvantage is that the optimized code does not get updated if the underlying grammar is changed. Therefore, we recommend this step only in the final stages of development, when the original grammar is stable. We used this feature to quickly verify if it is worthwhile to implement a particular optimization (*e.g.*, to quickly measure the impact of postponing instantiations of collections).

6.4. Future work

There are several issues we would like to address in the future. Our first concern is the complexity of the parser compiler. In the current (first) implementation, analysis and optimization phases have many hidden dependencies, are hard to understand, extend and debug. We would like to experiment with a simplified architecture that can offer reasonably good performance improvements (as we started with PetitParser2, mentioned above).

Furthermore, we would like to make the scanner more universal. Currently the whole grammar has to be scannable (all input has to be consumed by tokens) for the scanner to work with good performance. We would like to investigate possibilities for integrating a scanner into an arbitrary grammar, using the scanner only for rules that are scannable.

We would like to include ideas of LL(*) to properly guide choices and limit backtracking as well. Last but not least, there is an interesting and exciting question of run-time compilation (instead of our ahead-of-time approach), which imposes challenges in hot-spot analysis and fast-path compilation.

6.5. Related work

There has been recent research in Scala parser combinators [41,39] that is closely related to our work. The general idea is to perform compile-time optimizations to avoid unnecessary overhead of parser combinators at run-time. In the work *Accelerating Parser Combinators with Macros* [6] the authors argue for the use of macros [9] to remove the composition overhead. In *Staged Parser Combinators for Efficient Data Processing* [28] the authors use a multi-stage programming [49] framework LMS [47] 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 our work we implemented a dedicated tool from scratch. In contrast, the other approaches use meta-programming to manipulate compiler expression trees to optimize parser combinators. In our work, we consider several types of domain-based optimizations guided by a need to produce fast and clean top-down parsers.

While the authors of *Accelerating Parser Combinators with Macros* report throughput of JSON to be 16 MB/s, and the authors of *Staged Parser Combinators for Efficient Data Processing* report throughput of JSON to be 58 MB/s, our tokenized version

[29] http://www.webcitation.org/6r063VwVZ.

of JSON parser has throughput of roughly 15 MB/s. The reported performance of JSON parser written in Parboiled 2[30] is 5 MB/s [6,28]. These numbers show that the performance of parsers produced by parser compiler is comparable to optimized Scala parser combinators, even though the underlying Java VM is considered roughly 3–30× times faster than the Smalltalk VM.[31]

Other approaches leading to better combinator performance are memoization [17] and Packrat Parsing [15] (which are already used by PetitParser). In *Efficient combinator parsers* [29] Koopman et al. use the continuation-passing style to avoid intermediate list creation.

In Faster Scannerless GLR parsing [12] the authors apply the Right-Nulled Generalized LR parsing algorithm (RNGLR) [48, 13] to scannerless parsing [52]. They adapt Scannerless Generalized LR parsing and a filtering algorithm based on Farshi's version of GLR [50].

RNGLR is a recent derivate of Tomita's GLR algorithm that limits the cost of non-determinism in GLR. The authors show that using RNGLR for SGLR is faster on real applications than SGLR, with speedup up to 16% for mainstream languages such as C, C++, Java and Python. This is a performance improvement similar to that we measured when evaluating the impact of the adaptable scanner in subsection 5.4.

There are table-driven or top-down parser generators such as YACC [26], SmaCC [8], Bison [37], ANTLR [42] or Happy [21] that provide very good performance but they do not easily support context-sensitivity. The table-driven approaches cannot compete with the peak performance of top-down parsers [43].

Our work is also related to compilers supporting custom DSLs and providing interfaces for optimizations, especially specializations such as in Truffle [23]. Our parser compiler also provides specializations for different parsing expressions resulting in better overall performance.

7. Conclusion

In this work we investigate the idea of using adaptable parsing strategies to optimize the performance of parser combinators. As a proof of concept, we present a parser compiler – an ahead-of-time source-to-source optimizer for PetitParser. The speedup of parsers produced by the parser compiler ranges from a factor of two to four for context-free grammars and ten to twenty for context-sensitive grammars. The parser compiler does not impose any restrictions on the underlying grammars and preserves the advantages and flexibility of parser combinators. Based on our Smalltalk case study, the parser compiler provides two times better performance than a table-driven parser compiled by SmaCC, and approximately 10% worse performance than a hand-written optimized parser that uses a scanner.

Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Analysis” (SNSF project No. 200020-162352, Jan. 1, 2016–Dec. 30, 2018).

Appendix A. Parsing expression grammars

Parsing expression grammars (PEGs), developed by Ford [16], offer an alternative, recognition-based formal foundation for language syntax. PEGs are stylistically similar to CFGs with regular expression-like features resembling Extended Backus-Naur Form (EBNF) notation [55,25]. 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 are more syntactically expressive than the LL(k) language class typically associated with top-down parsers 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 memoizing packrat parser [15].

The set of operators used to express PEG definitions is in Table A.1. An example of a definition describing identifiers can be seen in Listing A.1. Table A.1 is extended with two context-sensitive operators: indent and dedent. They are part of our context-sensitive extension of parsing expression grammars based on parsing contexts [33].

[30] http://www.webcitation.org/6k6195CiS.

[31] http://www.webcitation.org/6rUPEZhDh.

Table A.1
PEG operators.

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
▷	Indent
◁	Dedent

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

Listing A.1: Example of a PEG definition.

The PEG syntax resembles EBNF. In addition to (i) prioritized choice `/`, the differences are (ii) syntactic predicates `!` and `&`, and (iii) non-terminal assignment `←` instead of `→`. The semantics are 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).[32] 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 [27], which are ambiguous in CFGs and have to be disambiguated by the longest-match meta-rule).

Definition A.1 *(Layout-Sensitive Parsing Expression Grammar (LS-PEG)).* A layout-sensitive 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, 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, $A \in V_N$
- $e_1 e_2$, a sequence
- e_1/e_2, a prioritized choice
- `e*`, zero-or-more repetitions
- `!e`, a not-predicate
- ▷, an indent
- ◁, a dedent □

Definition A.2 *(PEG semantics).* To formalize the semantics of a layout-sensitive grammar $G = (N, \Sigma, R, e_s)$, we extend the standard semantics of PEGs (defined by Ford [16]) as follows: input is a triple (e, x, S_{in}) (expression, input, stack), output is a triple (o, y, S_{out}) (output, suffix, new stack), where (i) e is a parsing expression, (ii) $x \in \Sigma^*$ is an input string to be recognized, (iii) S is an indentation stack of indentation levels $i_n : \ldots : i_2 : i_1 : [\,]$, (iv) $[\,]$ denotes an empty stack, (v) i_1 is the bottom element, (vi) i_n is the top element, (vii) $((i{:}S))$ denotes a stack S with i on top, (viii) $o \in \Sigma^* \cup \{f\}$ indicates the result of a recognition attempt, and (ix) the distinguished symbol $f \notin \Sigma$ indicates failure. The function $col(x)$ returns column on which the string x begins.

Empty:
$$\frac{x \in \Sigma^*}{(\epsilon, x, S) \Rightarrow (x, \epsilon, S)}$$

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

Terminal *(success case):*
$$\frac{a \in \Sigma, x \in \Sigma^*}{(a, ax, S) \Rightarrow (x, a, S)}$$

Terminal *(failure case):*
$$\frac{a \neq b, \quad (a, \epsilon, S) \Rightarrow (a, f, S)}{(a, bx, S) \Rightarrow (bx, f, S)}$$

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

Sequence *(success case):*
$$\frac{\begin{array}{c}(e_1, x, S) \Rightarrow (y_1, o_1, S_1)\\ (e_2, y_1, S_1) \Rightarrow (y_2, o_2, S_2)\end{array}}{(e_1 e_2, x, S) \Rightarrow (y_2, o_1 o_2, S_2)}$$

Sequence *(failure 2):*
$$\frac{(e_1, x, S) \Rightarrow (x, f, S)}{(e_1 e_2, x, S) \Rightarrow (x, f, S)}$$

Sequence *(failure 2):*
$$\frac{\begin{array}{c}(e_1, x, S) \Rightarrow (y, o, S_1)\\ (e_2, y, S_1) \Rightarrow (y, f, S_1)\end{array}}{(e_1 e_2, x, S) \Rightarrow (x, f, S)}$$

Alternation *(case 1):*
$$\frac{(e_1, x, S) \Rightarrow (y, o, S')}{(e_1/e_2, x, S) \Rightarrow (y, o, S')}$$

Alternation *(case 2):*
$$\frac{\begin{array}{c}(e_1, x, S) \Rightarrow (x, f, S)\\ (e_2, x, S) \Rightarrow (y, o, S')\end{array}}{(e_1/e_2, x, S) \Rightarrow (y, o, S')}$$

Repetitions *(repetition):*
$$\frac{\begin{array}{c}(e, x, S) \Rightarrow (y_1, o_1, S_1)\\ (e*, y_1, S_1) \Rightarrow (y_2, o_2, S_2)\end{array}}{(e*, x, S) \Rightarrow (y_2, o_1 o_2, S_2)}$$

Repetitions *(termination):*
$$\frac{(e, x, S) \Rightarrow (x, f, S)}{(e*, x, S) \Rightarrow (x, \epsilon, S)}$$

Not predicate *(success):*
$$\frac{(e, x, S) \Rightarrow (y, o, S')}{(!e, x, S) \Rightarrow (x, f, S)}$$

Not predicate *(failure):*
$$\frac{(e, x, S) \Rightarrow (x, f, S)}{(!e, x, S) \Rightarrow (x, \epsilon, S)}$$

Indent *(success):*
$$\frac{i = col(x) \quad i > i'}{(\triangleright, x, (i'{:}S)) \Rightarrow (x, \epsilon, (i{:}(i'{:}S)))}$$

Indent *(failure):*
$$\frac{col(x) <= i'}{(\triangleright, x, (i'{:}S)) \Rightarrow (x, \epsilon, (i'{:}S))}$$

Dedent *(success):*
$$\frac{j = col(x) \quad j = i'}{(\triangleright, x, (i{:}(i'{:}S))) \Rightarrow (x, \epsilon, (i'{:}S))}$$

Dedent *(failure):*
$$\frac{j = col(x) \quad j \neq i'}{(\triangleright, x, (i{:}(i'{:}S))) \Rightarrow (x, \epsilon, (i{:}(i'{:}S)))}$$

Syntactic sugar The following expressions are syntactic sugar and can be expressed as follows: (i) an optional expression `e?` is equivalent to `e/ϵ`, (ii) one or more repetitions `e+` is equivalent to `ee∗`, and (iii) a character class is equivalent to a choice of one-character literals `'a'/'b'/'c'/...`.

Reductions of repetitions As in CFGs, repetition expressions can be eliminated from a PEG by converting them into right-recursive nonterminals [16]. If expression e in repetition $e*$ accepts ϵ, the nonterminal is left-recursive and cannot be handled by PEGs in general. To simplify our definitions we treat repetitions as a possibly infinite sequence of the same expression.

Parsing Expression Languages Expression e *accepts* a string x if $\exists(e, x,) \Rightarrow^{+} (\epsilon, x,)$. Expression e *succeeds* on a string xy if $\exists(e, xy,) \Rightarrow^{+} (y, x,)$.

Definition A.3 *(Parsing Expressions Languages (PELs)).* Parsing expression language $\mathcal{L}(e)$ (as defined by Ford) of a parsing expression e over the alphabet Σ is the set of strings $x \in \Sigma^*$ for which the e *succeeds* on x:

$$\mathcal{L}(e) = \{xy \mid (e, x,) \Rightarrow^{+} (y, x,)\} \qquad \square$$

Note that in this definition, `e` does not need to 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.

A.1. 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 can be recognized by FSAs.

Definition A.4 *(Regular Parsing Expressions (RPEs)).* A regular parsing expression (RPE) is a parsing expression that can be recognized by a finite state automaton. $\square$

Note that it is not trivial to recognize which parsing expressions are recognizable by FSAs and which are not. Consider the rather simple parsing expression `('PetitParser'/'Petit')'P'`. The language $\mathcal{L}$ of this expression looks like this:

$$\mathcal{L}(\texttt{('PetitParser'/'Petit')'P'}) = \left\{ \begin{array}{c} \texttt{'PetitP'} \\ \texttt{'PetitPa'} \\ \ldots \\ \texttt{'PetitParse'} \\ \texttt{'PetitParserP'} \\ \ldots \end{array} \right\}$$

Note that the string `'PetitParser'` (or any string starting with `'PetitParser'` and not followed by `'P'`) is not in the language even though other strings starting with `'Petit'` are in the language. Such cases have to be carefully checked, because they cannot be expressed with traditional FSAs.

To be able to use traditional FSAs, we consider regular parsing expression such parsing expressions that satisfy the following two properties:

1. An expression e cannot refer to itself, *e.g.*, $A \leftarrow aAa$.[33]
2. If a parsing expression e accepts x, e also accepts any string xy (x being its prefix).

A.2. PEG analysis

For the parser compiler optimizations, we 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 grammar optimizations as demonstrated in section 5.

A.2.1. First set

The *first set* from traditional parsing theory [20, pp. 235–361] can be computed even for PEs [44]. For example, any character in the `[a−zA−Z]` character class is in the first set of `id` (see Listing A.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`.

The following first set analysis is used to optimize choices of parsing expressions (see section 4.2):

Definition A.5 *(First set).* We define the first set *FIRST*(e) of an expression e as a set of expressions such that:

Nonterminal $\quad \dfrac{a \in \Sigma}{FIRST(a) = \{a\}}$

[33] Note that $A \leftarrow aA$ can be expressed as $a*$.

Empty String	$\dfrac{}{\mathit{FIRST}(\epsilon) = \{\epsilon\}}$
Sequence *(case 1):*	$\dfrac{\epsilon \notin \mathit{FIRST}(e_1)}{\mathit{FIRST}(e_1 e_2) = \mathit{FIRST}(e_1)}$
Sequence *(case 2):*	$\dfrac{\epsilon \in \mathit{FIRST}(e_1)}{\mathit{FIRST}(e_1 e_2) = \mathit{FIRST}(e_1) \cup \mathit{FIRST}(e_2)}$
Choice	$\dfrac{}{\mathit{FIRST}(e_1/e_2) = \mathit{FIRST}(e_1) \cup \mathit{FIRST}(e_2)}$
Repetition	$\dfrac{}{\mathit{FIRST}(e*) = \mathit{FIRST}(e) \cup \{\epsilon\}}$
Not Predicate *case 1*	$\dfrac{\epsilon \notin \mathit{FIRST}(e)}{\mathit{FIRST}(!e) = \{!f_1\, !f_2\, \ldots !f_n \mid f_i \in \mathit{FIRST}(e)\}}$
Not Predicate *case 2*	$\dfrac{\epsilon \in \mathit{FIRST}(e)}{\mathit{FIRST}(!e) = \{\}}$
Indent	$\dfrac{}{\mathit{FIRST}(\triangleright) = \{\triangleright\}}$
Dedent	$\dfrac{}{\mathit{FIRST}(\triangleleft) = \{\triangleleft\}}$

A.2.2. Abstract simulation

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 [16]. 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.

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 of 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) $e_1 e_2 \rightharpoonup 0$ if $e_1 \rightharpoonup 0$ and $e_2 \rightharpoonup 0$.
   (b) $e_1 e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup 1$ and $e_2 \rightharpoonup s$.
   (c) $e_1 e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup s$ and $e_2 \rightharpoonup 1$.
   (d) $e_1 e_2 \rightharpoonup f$ if $e_1 \rightharpoonup f$.
   (e) $e_1 e_2 \rightharpoonup f$ 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 $e \rightharpoonup f$
8. (a) $\triangleright \rightharpoonup 0$
   (b) $\triangleright \rightharpoonup f$

9. (a) $\triangleleft \rightharpoonup 0$
   (b) $\triangleleft \rightharpoonup f$ $\square$

Nullable and accepts epsilon analyses are used in a parser compiler to reduce the overhead of backtracking. There can be different definitions of nullability [5,14]; we define nullability with the help of abstract simulation:

Definition A.7 *(Nullable expression).* We call an expression e *nullable*, if

$$e \rightharpoonup 0 \wedge e \not\rightharpoonup f \qquad \square$$

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.

Definition A.8 *(Accepts epsilon).* We say that an expression e *accepts epsilon*, if

$$e \rightharpoonup 0 \qquad \square$$

In other words, e accepts epsilon if it can succeed on some input string while consuming no input. For example, the start of a line `^` 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.

A.2.3. Push–pop analysis

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 for standard parsing expressions.
2. If an expression e pushes to a stack, the result is $\underline{\triangledown}$.
3. If an expression e pops from a stack, the result is $\underline{\vartriangle}$.
4. If an expression modifies a stack in some other way, *e.g.*, $\underline{\vartriangle}*$, 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 A.9.

Definition A.9 (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 $o \in \{0, \underline{\triangledown}, \underline{\vartriangle}, 1\}$. If $e \hookrightarrow_S 0$, then e does not modify the stack S. If $e \hookrightarrow_S \underline{\triangledown}$, then e pushes an element to the stack S. If $e \hookrightarrow_S \underline{\vartriangle}$, then e pops an element from the stack S. If $e \hookrightarrow_S 1$, then 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. $\triangleright \hookrightarrow_S \underline{\triangledown}$
4. $\triangleleft \hookrightarrow_S \underline{\vartriangle}$
5. (a) $e_1e_2 \hookrightarrow_S 0$ if $e_1 \hookrightarrow_S 0$ and $e_2 \hookrightarrow_S 0$.
   (b) $e_1e_2 \hookrightarrow_S 0$ if $e_1 \hookrightarrow_S \underline{\triangledown}$ and $e_2 \hookrightarrow_S \underline{\vartriangle}$.
   (c) $e_1e_2 \hookrightarrow_S \underline{\triangledown}$ if $e_1 \hookrightarrow_S \underline{\triangledown}$ and $e_2 \hookrightarrow_S 0$.
   (d) $e_1e_2 \hookrightarrow_S \underline{\triangledown}$ if $e_1 \hookrightarrow_S 0$ and $e_2 \hookrightarrow_S \underline{\triangledown}$.
   (e) $e_1e_2 \hookrightarrow_S \underline{\vartriangle}$ if $e_1 \hookrightarrow_S \underline{\vartriangle}$ and $e_2 \hookrightarrow_S 0$.
   (f) $e_1e_2 \hookrightarrow_S \underline{\vartriangle}$ if $e_1 \hookrightarrow_S 0$ and $e_2 \hookrightarrow_S \underline{\vartriangle}$.
   (g) $e_1e_2 \hookrightarrow_S 1$ otherwise
6. (a) $e_1/e_2 \hookrightarrow_S 0$ if $e_1 \hookrightarrow_S 0$ and $e_2 \hookrightarrow_S 0$
   (b) $e_1/e_2 \hookrightarrow_S \underline{\triangledown}$ if $e_1 \hookrightarrow_S \underline{\triangledown}$ and $e_2 \hookrightarrow_S \underline{\triangledown}$
   (c) $e_1/e_2 \hookrightarrow_S \underline{\vartriangle}$ if $e_1 \hookrightarrow_S \underline{\vartriangle}$ and $e_2 \hookrightarrow_S \underline{\vartriangle}$
   (d) $e_1/e_2 \hookrightarrow_S 1$ otherwise
7. (a) $e* \hookrightarrow_S 0$ if $e \hookrightarrow_S 0$
   (b) $e* \hookrightarrow_S 1$ otherwise
8. $!e \hookrightarrow 0$
9. $k(e) \hookrightarrow o$ if $e \hookrightarrow_S o$ $\square$

Table B.2
PetitParser operators.

Operator	Description
`''`	Literal string
`[]`	Character class
`[]negate`	Complement of a character class
`#letter`	Characters [a-zA-Z]
`#digit`	Characters [0-9]
`#space`	Characters [\t\n␣]
`#indent`	Python-like indent
`#dedent`	Python-like dedent
`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 token`	Trim spacing and build a token
`e map:action`	Semantic Action

As a practical example, consider the `body` rule from Listing 1:

```
heredoc     ← #indent
                 (method / class)*
              #dedent
```

The `indent` and `dedent` rules are push $\underline{\nabla}$ and pop $\underline{\Delta}$ respectively. The whole `body` sequence does not change the context, because the initial push in the rule `indent` is reverted by the last pop in `dedent`.

Appendix B. PetitParser

In this section we discuss in detail the implementation of PetitParser. PetitParser is implemented in Pharo Smalltalk,[34] Smalltalk/X,[35] Java[36] and Dart.[37] PetitParser uses an internal DSL similar to a standard PEG syntax as briefly described in Table B.2.

Parsing contexts Parser combinator frameworks are very flexible, allowing for modifications of a parser combinator graph itself. This gives them the expressiveness of context-free formalisms. Context-sensitivity facilitates grammar adaptability [45, 10] or adaptation of other of contextual information – *parsing contexts* [33] in the case of PetitParser.

A parsing context is a list of stacks that (i) can be used to steer a parser's decisions, (ii) can be manipulated via push and pop operators, and (iii) allow context-sensitive restrictions to be expressed.

Parsing contexts are used by the `#indent` and `#dedent` combinators (see the implementation in Listing B.6 and Listing B.7 in subsection B.1). Whenever `#indent` or `#dedent` is recognized, the indentation stack in `context` is accessed and a new column is pushed or popped.

Parser invocation When a root parser is asked to attempt a parse on an input, (i) `context` is created, (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 an `Action` parser (its implementation is in Listing B.2) used in the `class`. The underlying parser (*i.e.*, a sequence of `classToken`, `idToken` and `body`) is invoked and its result is evaluated by the block. In case the underlying parser fails, a failure result is returned immediately without evaluating the block.

Backtracking and memoization PetitParser uses backtracking. Thanks to the backtracking capabilities, a top-down combinator-based parser is not limited to LL(k) grammars [3], and handles unlimited lookahead.

In PetitParser, before a 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). A dedicated `Memoizing` parser combinator creates

[34] http://smalltalkhub.com/#!/~Moose/PetitParser.
[35] http://www.webcitation.org/6k62sGRlg.
[36] http://www.webcitation.org/6k62uAxHz.
[37] http://www.webcitation.org/6k62vRsWA.

```
ActionParser>>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 B.2: Implementation of `ActionParser`.

```
AndPredicateParser>>parseOn: context
    | memento |
    memento ← context remember.
    result ← parser parseOn: context.
    context restore: memento.
    ↑ result isFailure ifTrue: [
        ↑ result
    ] ifFalse: [
        ↑ nil
    ]
```

Listing B.3: Implementation of `AndPredicate`.

```
CharClassParser>>parseOn: context
    (context atEnd not and:
    [charClass includes: context peek]) ifTrue: [
      ↑ context next
    ] ifFalse: [
      ↑ PPFailure message: 'Character not expected'

    ]
```

Listing B.4: Implementation of `CharClassParser`.

```
ChoiceParser>>parseOn: context
    | result |
    self children do: [:child |
        result ← child parseOn: context.
        result isSuccess ifTrue: [
            ↑ result
        ]
    ].
    ↑ result
```

Listing B.5: Implementation of `ChoiceParser`.

a memento, performs the parse attempt and stores the *memento-result* pair into a buffer. Later, if the memoizing parser is invoked again and a memento is found in the buffer, the result is returned directly.

Creating a memento of a context-free parser is easy: it stores the current position in the input stream. However, for context-sensitive parsers, a memento holds a deep copy of the whole context [33], *e.g.*, for layout-sensitive grammars it is a position in the input stream together with a copy of the indentation stack (see Listing B.11 in subsection B.1).

To understand how PetitParser backtracks, consider the sequence of `'class'` and `&#space` in `classToken` (the `Sequence` parser implementation is in Listing B.8). If the parsed input is not the `class` keyword (*e.g.*, `'classifier'`) the first rule `'class'` consumes `'class'` and increases the position by five and the second rule `&#space` fails. The sequence restores the position into the original state before `'class'` and returns a failure. Note that in PetitParser a parser returning a failure from `parseOn:` is responsible for restoring the context to its initial state, *i.e.*, as it was on the `parseOn:` invocation.

Trimming and tokenization Because PetitParser is scannerless [52], a dedicated `TokenParser` is at hand to deal with whitespaces. It trims the whitespaces (or comments if specified) from input before and after a parse attempt. As a result a `Token` instance is returned holding the parsed string and its start and end positions. As an example, consider the `classToken` rule, which (when provided with the `'class'` input) returns an instance of `Token` with start and stop positions and `'class'` as a value (see Listing B.10 in subsection B.1).

```
IndentParser>>parseOn: context
  | memento lastIndentation indentation |
  memento ← context remember.
  self consumeWhitespace.
  lastIndentation ← context indentationStack top.
  indentation ← context column.
  (lastIndentation < indentation) ifTrue: [
    context indentationStack push: indentation.
    ↑ #indent
  ] ifFalse: [
    context restore: memento.
    ↑ Failure message: 'No indent found'
  ]
```

Listing B.6: Implementation of `Indent`.

```
DedentParser>>parseOn: context
  | memento lastIndentation indentation |
  memento ← context remember.
  self consumeWhitespace.
  lastIndentation ← context indentationStack top.
  indentation ← context column.
  (lastIndentation >= indentation) ifTrue: [
    context indentationStack pop.
    ↑ #dedent
  ] ifFalse: [
    context restore: memento.
    ↑ Failure message: 'No dedent found'
  ]
```

Listing B.7: Implementation of `Dedent`.

```
SequenceParser>>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 B.8: Implementation of `SequenceParser`.

```
StarParser>>parseOn: context
    | retval result |
    retval ← OrderedCollection new.
    [
        result ← child parseOn: context.
        result isSuccess
    ] whileTrue: [
        retval add: result
    ]
    ↑ retval
```

Listing B.9: Implementation of `StarParser`.

B.1. Implementation of combinators

In this section we provide an implementation of all the combinators mentioned in this work to help the reader better understand the overhead of a combinator library.

```
TokenParser>>parseOn: context
    | memento result |
    memento ← context remember.
    whitespace parseOn: context.
    result ← parser parseOn: context.

    result isFailure ifTrue: [
        context restore: memento
        ↑ result
    ].
    whitespace parseOn: context.

    ↑ Token new
        value: result flatten;
        start: memento position;
        end: context position
```

Listing B.10: Implementation of `TokenParser`.

```
Context>>remember
    ↑ Memento new
        position: position
        stacks: stacks deepCopy

Context>>restore: memento
    self position: memento position
         stacks: stacks deepCopy
```

Listing B.11: Implementation of `Context` remember and restore.

References

[1] H. Abelson, G.J. Sussman, J. Sussman, Structure and Interpretation of Computer Programs. MIT Electrical Engineering and Computer Science Series, McGraw-Hill, 1991.
[2] M.D. Adams, O.S. Ağacan, Indentation-sensitive parsing for Parsec, in: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, Haskell'14, ACM, New York, NY, USA, 2014, pp. 121–132.
[3] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation and Compiling. Volume I: Parsing, Prentice-Hall, 1972.
[4] A.V. Aho, J.D. Ullman, Principles of Compiler Design, Addison-Wesley Ser. Comput. Sci. Inform. Process., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.
[5] R.C. Backhouse, Syntax of Programming Languages: Theory and Practice, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.
[6] E. Béguet, M. Jonnalagedda, Accelerating parser combinators with macros, in: Proceedings of the Fifth Annual Scala Workshop, SCALA '14, ACM, New York, NY, USA, 2014, pp. 7–17.
[7] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009.
[8] J. Brant, D. Roberts, SmaCC, a Smalltalk Compiler–Compiler, http://www.refactory.com/Software/SmaCC/.
[9] E. 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, ACM, New York, NY, USA, 2013, pp. 3:1–3:10.
[10] H. Christiansen, Adaptable grammars for non-context-free languages, in: J. Cabestany, F. Sandoval, A. Prieto, J. Corchado (Eds.), Bio-Inspired Systems: Computational and Ambient Intelligence, in: Lect. Notes Comput. Sci., vol. 5517, Springer, Berlin, Heidelberg, 2009, pp. 488–495.
[11] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Trans. Program. Lang. Syst. 13 (4) (1991) 451–490.
[12] G. Economopoulos, P. Klint, J. Vinju, Compiler construction, in: 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software. Proceedings, ETAPS 2009, York, UK, March 22–29, 2009, Springer, Berlin, Heidelberg, 2009, pp. 126–141.
[13] G.R. Economopoulos, Generalised LR Parsing Algorithms, PhD thesis, University of London, 2006.
[14] R.I. Fabio Mascarenhas, Sérgio Medeiros, On the relation between context-free grammars and parsing expression grammars, CoRR, arXiv:1304.3177 [abs], 2013.
[15] B. Ford, Packrat parsing: simple, powerful, lazy, linear time, functional pearl, in: ICFP 02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, vol. 37/9, ACM, New York, NY, USA, 2002, pp. 36–47.
[16] B. 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, ACM, New York, NY, USA, 2004, pp. 111–122.
[17] R.A. Frost, B. Szydlowski, Memoizing purely functional top-down backtracking language processors, Sci. Comput. Program. 27 (3) (Nov. 1996) 263–288.
[18] E. Gamma, Extension object, in: Pattern Languages of Program Design 3, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997, pp. 79–88.
[19] D. Grune, C.J. Jacobs, Deterministic top–down parsing, in: Parsing Techniques – A Practical Guide, vol. 1, 2008, pp. 235–361, Chapter 8.
[20] D. Grune, C.J. Jacobs, Parsing Techniques – A Practical Guide, Springer, 2008.
[21] Happy – the parser generator for Haskell, http://tfs.cs.tu-berlin.de/agg/index.html, 2010.
[22] A. Hindle, M.W. Godfrey, R.C. Holt, Reading beside the lines: indentation as a proxy for complexity metrics, in: ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension, 2008, IEEE Computer Society, Washington, DC, USA, 2008, pp. 133–142.
[23] C. Humer, C. Wimmer, C. Wirth, A. Wöß, T. Würthinger, A domain-specific language for building self-optimizing AST interpreters, SIGPLAN Not. 50 (3) (Sept. 2014) 123–132.
[24] G. Hutton, E. Meijer, Monadic Parser Combinators, Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
[25] ISO, Information Technology – Syntactic Metalanguage – Extended BNF, ISO 14997, International Organization for Standardization, Geneva, Switzerland, 1996.

[26] S. Johnson, Yacc: Yet Another Compiler Compiler, Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.
[27] S.P. Jones, Haskell 98 Language and Libraries: The Revised Report, Cambridge University Press, 2003.
[28] M. Jonnalagedda, T. Coppey, S. Stucki, T. Rompf, M. 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, ACM Press, New York, NY, USA, 2014, pp. 637–653.
[29] P. Koopman, R. Plasmeijer, Efficient combinator parsers, in: Implementation of Functional Languages, in: Lect. Notes Comput. Sci., Springer-Verlag, 1998, pp. 122–138.
[30] J. Kurš, Parsing For Agile Modeling, PhD thesis, University of Bern, Oct. 2016.
[31] J. Kurš, G. Larcheveque, L. Renggli, A. Bergel, D. Cassou, S. Ducasse, J. Laval, PetitParser: building modular parsers, in: Deep Into Pharo, Square Bracket Associates, Sept. 2013.
[32] J. Kurš, M. Lungu, R. Iyadurai, O. Nierstrasz, Bounded seas, in: Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014), Comput. Lang. Syst. Struct. 44 (Part A) (2015) 114–140.
[33] J. Kurš, M. Lungu, O. Nierstrasz, Top-down parsing with parsing contexts, in: Proceedings of International Workshop on Smalltalk Technologies, IWST 2014, 2014.
[34] J. Kurš, J. Vraný, M. Ghafari, M. Lungu, O. Nierstrasz, Optimizing parser combinators, in: Proceedings of International Workshop on Smalltalk Technologies, IWST 2016, 2016, pp. 1:1–1:13.
[35] C. Lattner, V. Adve, LLVM: a compilation framework for lifelong program analysis & transformation, in: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, IEEE Computer Society, Washington, DC, USA, 2004, p. 75.
[36] M. Lesk, E. Schmidt, Lex – A Lexical Analyzer Generator, Computer Science Technical Report #39, Bell Laboratories, Murray Hill, NJ, 1975.
[37] J. Levine, Flex & Bison: Text Processing Tools, O'Reilly Media, 2009.
[38] L. Moonen, Generating robust parsers using island grammars, in: E. Burd, P. Aiken, R. Koschke (Eds.), Proceedings Eighth Working Conference on Reverse Engineering, WCRE 2001, IEEE Computer Society, Oct. 2001, pp. 13–22.
[39] A. Moors, F. Piessens, M. Odersky, Parser Combinators in Scala, Technical Report, Department of Computer Science, K.U. Leuven, Feb. 2008.
[40] O. Nierstrasz, S. Ducasse, T. Gîrba, The story of Moose: an agile reengineering environment, in: Proceedings of the European Software Engineering Conference, ESEC/FSE'05, ACM Press, New York, NY, USA, Sept. 2005, pp. 1–10, Invited paper.
[41] M. Odersky, Scala Language Specification v. 2.4, Technical report, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, Mar. 2007.
[42] T.J. Parr, R.W. Quong, ANTLR: a predicated-LL(k) parser generator, Softw. Pract. Exp. 25 (1995) 789–810.
[43] T.J. Pennello, Very fast LR parsing, SIGPLAN Not. 21 (7) (July 1986) 145–151.
[44] R.R. Redziejowski, Applying classical concepts to parsing expression grammar, Fundam. Inform. 93 (1–3) (Jan. 2009) 325–336.
[45] d.S. Reis, L. Vieira, d.S. Bigonha Roberto, D. Iorio, V. Oliveira de Souza Amorim, L. Eduardo, Adaptable parsing expression grammars, in: F. de Carvalho Jr., L. Barbosa (Eds.), Programming Languages, in: Lect. Notes Comput. Sci., vol. 7554, Springer, Berlin, Heidelberg, 2012, pp. 72–86.
[46] L. Renggli, S. Ducasse, T. Gîrba, O. Nierstrasz, Practical dynamic grammars for dynamic languages, in: 4th Workshop on Dynamic Languages and Applications, DYLA 2010, Malaga, Spain, June 2010, pp. 1–4.
[47] T. Rompf, M. 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, ACM, New York, NY, USA, 2010, pp. 127–136.
[48] E. Scott, A. Johnstone, Right nulled GLR parsers, ACM Trans. Program. Lang. Syst. 28 (4) (July 2006) 577–618.
[49] W. Taha, A gentle introduction to multi-stage programming, in: Domain-Specific Program Generation, 2003, pp. 30–50.
[50] M. Tomita, Generalized LR Parsing, Springer US, 2012.
[51] E. Visser, A Family of Syntax Definition Formalisms, Technical Report P9706, Programming Research Group, University of Amsterdam, July 1997.
[52] E. Visser, Scannerless Generalized-LR Parsing, Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.
[53] P. Wadler, Monads for functional programming, in: J. Jeuring, E. Meijer (Eds.), Advanced Functional Programming, in: Lect. Notes Comput. Sci., vol. 925, Springer-Verlag, 1995.
[54] P.R. Wilson, Uniprocessor garbage collection techniques, 1992.
[55] N. Wirth, What can we do about the unnecessary diversity of notation for syntactic definitions?, Commun. ACM 20 (11) (1977) 822–823.