Optimizing Parser Combinators

Jan Kurš[†], Jan Vraný[‡], Mohammad Ghafari[†], Mircea Lungu[¶] and Oscar Nierstrasz[†]

[†]Software Composition Group,
University of Bern,
Switzerland
scg.unibe.ch

[‡]Software Engineering Group,
Czech Technical University,
Czech Republic
swing.fit.cvut.cz

[¶]Software Engineering and
Architecture Group,
Universtiy of Groningen,
Netherlands
http://www.cs.rug.nl/search

Abstract

Parser combinators are a popular approach to parsing. Parser combinators 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, their universality and flexibility introduces a noticeable performance overhead. 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 still a significant constant multiplier. This can be further lowered using meta-programming techniques.

In this work we present a more traditional approach to optimization — a compiler — applied to the domain of parser combinators. A parser combinator compiler (pc-compiler) analyzes a parser combinator, applies parser combinator-specific optimizations and, generates an equivalent high-performance top-down parser. Such a compiler preserves the advantages of parser combinators while complementing them with better performance.

1. Introduction

Parser combinators (Wadler 1995; Moors et al. 2008) represent a popular approach to parsing. They are straightforward to construct, readable, modular, well-structured and easy to maintain. Parser combinators are also powerful in their expressiveness as they can parse beyond context-free languages (*e.g.*, layout-sensitive languages (Hutton and Meijer 1996; Adams and Ağacan 2014)).

Nevertheless, parser combinators yet remain a technology for prototyping and not for the actual deployment. Firstly, naive implementations do not handle left-recursive grammars, though there are already solutions to this issue (Frost et al. 2007; Warth et al. 2008). Secondly, the expressive power of parser combinators comes at a 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 machines or pushdown automata. That is, parser combinators cannot reach the peak performance of parser generators (Levine 2009), hand-written parsers or optimized code (see section 5 or (Béguet and Jonnalagedda 2014)§4).

Meta-programming approaches such as macros (Burmako 2013) and staging (Rompf and Odersky 2010) have been applied to Scala parser combinators (Moors et al. 2008) with significant performance improvements (Béguet and Jonnalagedda 2014; Jonnalagedda et al. 2014). In general, these approaches remove composition overhead and intermediate representations. Many other parser combinator implementations battle performance problems by using efficient structures, macros *etc.* (see *Parboiled 2*,[1] *attoparsec*[2] or *FParsec*[3]).

Nevertheless, none of the proposed optimizations yet reach the performance of hand-written code. In our work we focus on the performance of PetitParser (Renggli et al. 2010; Kurš et al. 2013) — a parser combinator framework utilizing packrat parsing (Ford 2002). We present *a parser combinator compiler* (pc-compiler)

[1] https://github.com/sirthias/parboiled2

[2] http://www.serpentine.com/blog/2014/05/31/attoparsec/

[3] http://www.quanttec.com/fparsec/

that utilizes standard compiler techniques and applies optimizations tailored to parser combinators. Based on a compile-time analysis of a grammar, we avoid composition overhead, reduce object allocations, and optimize choices to minimize the backtracking overhead whenever possible. The result of *"compilation"* is an optimized top-down parser that provides performance as fast as hand-written code. In particular, based on our benchmarks covering five different grammars for PetitParser,[4] a pc-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 20% slower compared to a highly-optimized, hand-written parser.

The paper makes the following contributions: i) application of compiler-technology approaches to the domain of parser combinators; ii) an identification of performance bottlenecks for PetitParser; iii) a description of optimization techniques addressing these bottlenecks; and iv) an analysis of their effectiveness.

The paper is organized as follows: In section 2 we introduce PetitParser, and in section 3 we describe its performance bottlenecks. In section 4 we introduce a pc-compiler — a framework to deal with problems described in section 3. In section 5 we provide a detailed performance analysis of a pc-compiler and its optimizations. In section 6 we describe other approaches to improve parser combinators performance, and we briefly discuss similarities and differences between them and our approach. Finally, section 7 concludes this paper.

2. Petit Parser

In this section we introduce PetitParser and inspect in detail its implementation. In the following section we describe performance bottlenecks of this implementation.

PetitParser (Renggli et al. 2010; Kurš et al. 2013) is a parser combinator framework (Hutton and Meijer 1996) that utilizes packrat parsing (Ford 2002), scannerless parsing (Visser 1997) and parsing expression grammars (PEGs) (Ford 2004). PetitParser is implemented in Pharo,[5] Smalltalk/X,[6] Java[7] and Dart.[8]

PetitParser uses an internal DSL similar to a standard PEG syntax as briefly described in Table 1. A grammar fragment describing a simple programming language is shown in Listing 1.

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

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␣]
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 map: *action*	Semantic Action
e token	Build a token

Table 1. PetitParser operators

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 `dednet` 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.[9] The class body contains a sequence of classes and methods.

2.1 Deep into PetitParser

Figure 1 shows a composition of parser combinators that are created after evaluating the code in Listing 1. 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 argu-

[4] arithmetic expressions, Java, Smalltalk and Python

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

[6] `https://bitbucket.org/janvrany/stx-goodies-petitparser`

[7] `https://github.com/petitparser/java-petitparser`

[8] `https://github.com/petitparser/dart-petitparser`

[9] `https://docs.python.org/3/reference/lexical_naalysis.html#indentation`

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

class        ← classToken  idToken  body
         map: [:classToken :idToken :body |
               ClassNode new
                   name: idToken value;
                   body: body]

body         ← indent
                   (class / method)*
               dedent
program      ← class+
```

Listing 1. Simple grammar in PetitParser DSL.

ments being collected from the result of an underlying `Sequence` parser.

The `idToken` and `classToken` rules are `Token` parsers, which consume whitespace and create token instances. The `'class'` rule is a `Literal` parser, which is a leaf combinator and does not refer to any other combinators. The `classToken` rule contains a sequence of `Literal` and `AndPredicate` combinators (omitted in the Figure 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. PetitParser combinators can be easily implemented by following the `parseOn: context` contract.

In the remainder of this section we discuss how PetitParser handles context-sensitive grammars, how are the parsers invoked, how the backtracking works and how PetitParser handles lexical elements as these are important for the subsequent performance analysis.

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 (Reis et al. 2012; Christiansen 2009) or adaptation of other of contextual in-

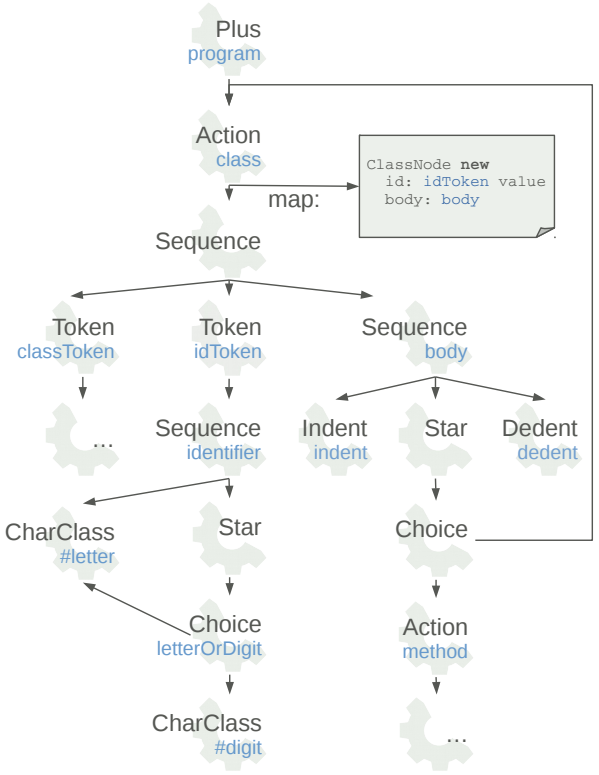

Figure 1. The structure of parser combinators created after evaluating the code of Listing 1.

formation — *parsing contexts* (Kurš et al. 2014) in the case of PetitParser.

Parser Invocation When a root parser is asked to perform a parse attempt 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 implementation in Listing 2. The underlying parser is invoked and its result is forwarded to the block. In case the underlying parser fails a failure is returned immediately.

Backtracking and Memoization PetitParser utilizes backtracking. Thanks to the backtracking capabilities, a top-down combinator-based parser is not limited to LL(k) grammars (Aho and Ullman 1972) 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 a memento, per-

forms 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 is the current position in the input stream. However, for context-sensitive parsers, a memento is a deep copy of the whole context (Kurš et al. 2014), *e.g.*, for layout-sensitive grammars it is a position in the input stream and a copy of the indentation stack (see Listing 10).

To see how PetitParser backtracks, consider the `Sequence` parser implementation in Listing 7. A sequence must iterate over all its children, collect their results and return a collection of the results. In case of a failure, the context is restored to the original state and a failure is returned.

Note that 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.

In PetitParser, the memoizing parser provides an *ad hoc* solution to detect the infinite recursive descent caused by left-recursion. The `Memoizing` parser monitors a number of mementos stored for a given parser and position and fails if a threshold is exceeded.

Trimming and Tokenization Because PetitParser is scannerless (Visser 1997), 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 (see Listing 9).

3. Performance Bottlenecks

In this section, we identify the most critical performance bottlenecks in PetitParser and explain them using the example from Listing 1.

3.1 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 in reality many methods are called. For each character, the following parsers are invoked: A `Star` parser (see Listing 8 in Appendix A), a `Choice` parser (see Listing 5 in Appendix A) and two `CharClass` parsers (see Listing 4 in Appendix A). 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 3) and `CharClass` (see Listing 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. Ideally, the result can be determined with a single comparison of a stream peek.

3.2 Superfluous intermediate objects

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

For example, consider an input `"IWST"` parsed by `idToken`. The return value of `idToken` is a `Token` object containing `"IWST"` as a value and `1` and `5` as start and end positions. Yet before a `Token` is created, a `Star` parser (see Listing 8) and a `Sequence` parser (see Listing 7) create intermediate collections resulting in `(I,(W,S,T))` nested arrays that are later flattened into `"IWST"` in `TokenParser` (see Listing 9) again. Furthermore `Sequence` creates a memento that is never used because the second part of the `identifier` sequence (*i.e.*, `letterOrDigit*`) never fails.[10]

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 2).

3.3 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 time 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 9), `Sequence` (see Listing 7), `CharClass` (see Listing 4). Furthermore, as `TokenParser` tries to consume a whitespace (*e.g.*, using `#space*`), another `Star` (see Listing 8) and `CharClass` (see Listing 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.

[10] Zero repetitions are allowed, which means that empty strings are also valid.

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

```
((’class’ token) idToken body) /
((’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.

3.4 Context-Sensitivity Overhead

In case of PetitParser, the most of the context-sensitivity overhead is produced in case a context contains complex objects (*e.g.*, an indentation stack). When memoizing a parser combinator deep-copies the whole context (see Listing 10). The copy is needed in `remember` as well as in `restore:`. If we restore a stack from a memento without the copy and the stack is modified, the memento is modified as well. If the memento is accessed in the future for another restore operation, we will need the unmodified version.

This is a valid approach in some cases, *e.g.*, the `body` sequence where `indent` modifies the stack (see Listing 6): if any subsequent rule in `body` fails, the original stack has to be restored. In some other cases, *e.g.*, the `identifier` sequence where none of the underlying combinators modify the indentation stack, the deep copy of the context is superfluous.

4. A Parser Combinator Compiler

The goal of a pc-compiler is to provide a high-performance parser from a parser combinator while preserving the advantages of parser combinators.

Figure 2 shows the workflow of parser development with the approach we present here. First, flexibility and comprehensibility of combinators is utilized (*prototype phase*). Then, a pc-compiler applies domain-specific optimizations, builds a top-down parser (*compile phase)* and allows an expert to further modify at her will[11] to further improve the performance (*hand-tune phase)*. In the end, the resulting parser can be deployed as an ordinary class and used for parsing with peak performance (*deployment phase)*.

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

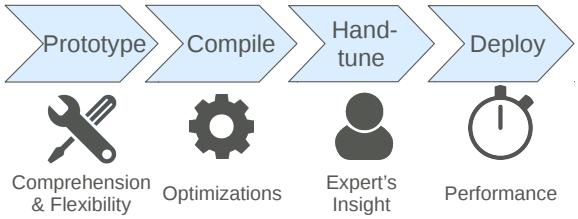

Figure 2. Workflow of parser development.

Similar to standard compilers, a pc-compiler transforms from one code to another while performing various optimizations on an intermediate representation resulting in more efficient code. However, a pc-compiler compiles into host code (*i.e.*, the language in which the parsing framework is implemented), whereas a standard compiler compiles into machine code.

In the following, we describe the intermediate representation of a pc-compiler, and what optimizations do we perform.

4.1 Intermediate Representation

In general, parser combinators form a graph with cycles. A pc-compiler uses the very same graph as its intermediate representation. The optimizations themselves are implemented as a series of passes over the graph, each performing a transformation using pattern matching. Particular nodes are moved, replaced with more appropriate alternatives, or changed and extended with additional information. In the final phase, these nodes are visited by a code generator to produce the optimized host code.

Contrary to other representations used for performance optimizations (*e.g.*, AST,[12] Bytecode,[13] SSA[14] or LLVM IR[15]) our intermediate representation is high-level, directly represent the target domain and therefore allows for different kinds of optimizations. In particular, the intermediate representation of a pc-compiler has the following advantages: First of all, parser combinators allow us to analyze directly the properties of a grammar, *e.g.*, to check if a choice is deterministic, or to determine the set of expected characters.

Second of all, any unknown parser combinator can be used as is, serving as an intermediate representation node. A parser combinator allows the referencing parser combinators to be replaced with an available compiled

[12] https://en.wikipedia.org/wiki/Abstract_syntax_tree

[13] https://en.wikipedia.org/wiki/Bytecode

[14] https://en.wikipedia.org/wiki/Static_single_assignment_form

[15] https://en.wikipedia.org/wiki/LLVM#LLVM_intermediate_representation

version which gives the pc-compiler itself great flexibility.

Third of all, an intermediate representation is executable, *i.e.*, at any point it can be used as a parser. If, for some reason, the compilation to host code fails, the intermediate representation can be used instead. This simplifies the development of new optimizations, helps with testing and allows pc-compiler to optimize even grammars with custom or third-party extensions.

4.2 Optimizations

In this section we describe pc-compiler optimizations applied to the performance problems of PetitParser (section 3). These optimizations are mostly orthogonal, *i.e.*, they do not depend on each other and can be implemented without mutual dependencies.

We use the following syntax for rewriting rules. The specific class of combinators is in angle brackets `<>`, *e.g.*, all the character class combinators are marked as `<CharClass>`. Any parser combinator is `<Any>`. A parser combinator that is a child of a parent `P` is marked `P`$\rightarrow^*$`<Any>`.

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

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

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

4.2.1 Specializations

Specializations reduce *composition overhead* by replacing a hierarchy of combinators by a single specialized combinator with the corresponding semantics. This specialized combinator can be implemented more efficiently and thus improves the performance.

Returning to the problem with `letterOrDigit*` in subsection 3.1, the whole rule is specialized as an instance of the `<CharClassStar>` combinator. The `#digit / #letter` rule is specialized using a single `CharClass` combinator `[a-zA-Z0-9]`, and a repetition of the character class is replaced by a specialized `CharClassStar` combinator, which can be implemented as a while loop. The `letterOrDigit*` rule is rewritten to the following:

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

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

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

Returning to the problem with `&#space` in subsection 3.1, the whole rule is specialized as a single `AndCharClass` combinator. The `classToken` rule is rewritten as follows:

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

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

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

We implement several similar specializations, including the following:

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

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

- A new `CharClass` is created from the negation of a char class, *e.g.*, `[a-z] negate`:

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

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

```
<CharClass>* ⇒ <CharClassStar>
```

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

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

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

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

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

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

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

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

4.2.2 Data Flow

Data-Flow analyses target the problem of *superfluous object allocations*. This improves performance since object initialization methods do not need to be run and because it improves the efficiency of a typical Smalltalk garbage collector (Wilson 1992).

We perform data-flow analysis in three different domains: (i) in tokens to improve the performance of lexical analysis; (ii) in sequences to avoid superfluous mementos that are never used; and (iii) in action blocks that are inlined and consequently analyzed for superfluous object allocations.

Token Combinators Combinators forming a `Token` parser are marked to avoid generating intermediate representations — *recognizers*, because they only return whether a string is in their language or not.

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

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

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

This results in the following code being generated:

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

Sequences `Sequence` combinators are inspected to determine whether a memento is necessary or not.

Following PEG semantics, if the first parser of a sequence fails, it restores the context to the initial state of the sequence and therefore the sequence can return the failure directly. If the first parser succeeds and the second parser of the sequence fails, the context is now that after the invocation of the first parser. Therefore, the sequence has to restore the context to the state as it was *before the invocation of the first parser* and return the failure. Consider the code generated from `identifier`:

```
identifier
    | memento result1 result2 |
    memento ← context remember.
    result1 ← self letter.
    result1 isFailure ifTrue: [
        "no restore, letter did it"
        ↑ result1.
    ]
    result2 ← self letterOrDigitStar.
    result2 isFailure ifTrue: [
        "but restore here"
        context restore: memento.
        ↑ result2
    ]
    ↑ Array with: result1 with: result2
```

A *nullable* combinator accepts an empty string ϵ and thus cannot fail (Backhouse 1979; Fabio Mascarenhas 2013). If a sequence is formed of combinators where all but first are nullable, the sequence is marked to not create a memento, because such a memento would never be used.

```
<Sequence<Any><Any:nullable>>  ⇒
    <Sequence<Any><Any>+nobacktack>
<Sequence<Any:nullable><Any:nullable>>  ⇒
    <Sequence<Any><Any>+nobacktack>
```

For example, the `identifier` sequence is marked to avoid a memento. It is also marked to avoid an intermediate representation, because it is inside a token:

```
letterOrDigit* ←
    CharClassStar[a-zA-Z0-9]:recognizer,nullable>
identifier     ←
    <Sequence <#letter>
    <letterOrDigit*>:nobacktrack,recognizer>
```

This results in the following code being generated:

```
identifier
    | result1 |
    result1 ← self letter.
    result1 isFailure ifTrue: [ ↑ result ].
    self letterOrDigitStar.
```

Actions PetitParser actions introduce a level of indirection that is difficult to optimize because an action can be an arbitrary user code. Consider the rule `class` from Listing 1. Without specialized optimizations the generated code looks like this (error handling has been omitted for clarity):

```
class
    | collection |

    collection ← OrderedCollection new.
```

```
    collection at: 1 put: self classToken.
    collection at: 2 put: self idToken.
    collection at: 3 put: self body.

    ↑ block valueWithArguments: collection
```

Clearly the `collection` is used only to pass arguments to the action block via the reflective API (`valueWithArguments:`). Both the allocation and the block invocation impose a performance overhead.

To reduce this overhead, the parser compiler (i) inlines the action, and (ii) replaces `collection` by local variables. After these optimizations the generated code for `class` rule looks like this:

```
class
    | idToken body |
    self classToken.
    idToken ← self idToken.
    body ← self body.
    ↑ ClassNode new
        name: idToken value;
        body: body
```

This way the superfluous allocation and block invocation are avoided. Furthermore, such code can be further optimized by the underlying just-in-time compiler (Hölzle 1994).

4.2.3 Guards

Guards prevent unnecessary invocations and allocations caused by *backtracking*. Guards allow for an early failure of a parse attempt using the peek character of an input stream retrieved from a precomputed first-set.

During a dedicated optimization phase, character-based *first sets* (Redziejowski 2009; Grune and Jacobs 2008b) are computed.[16] The nodes with a reasonably small first set (*e.g.*, digits only) are marked to be suitable for a guard (`+guard`). A guard is a combinator that prepends an underlying combinator with a code that fails immediately, without entering the underlying combinator.

Sequences Sequences with a guard property are wrapped with a `SeqGuard` combinator:

```
<Sequence:guard> ⇒ <SeqGuard<Sequence>>
```

For example, `identifier` is prepended with the following code, which prevents extra invocations:

```
identifier     ←
    <SeqGuard <Sequence <#letter>
    <letterOrDigit*>:nobacktrack,recognizer,guard>>
```

This results in the following code being generated:

```
context peek isLetter ifFalse: [
    ↑ Failure message: 'letter expected' ]
self letter.
self letterOrDigitStar.
```

Choices A variant of guards is used to guide `Choice` combinators. A choice option is entered only if the stream peek is in the *first set* of the given option. Any choice option marked for guarding `<Any:guard>` is wrapped with `OptionGuard` . Some options do not need to be guarded:

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

For example the `class / method` choice is wrapped with `OptionGuard` like this:

```
body ← indent
            (<OptionGuard<class:guard>>/
             <OptionGuard<method:guard>>)*
        dedent
```

This results in the following code being generated from the `class / method` choice:

```
(context peek == $c) ifTrue: [ self class ].
(context peek == $m) ifTrue: [ self method ].
```

4.2.4 Context-Sensitive Memoization

Context-sensitive memoization reduces the overhead of context-sensitive grammars. The deep copy of a context is performed only if necessary, *i.e.*, for the *context-sensitive* parts of a grammar. The context-free expressions use only a position in a stream as a memento.

By default, combinators are marked as context-free (`+contextFree`). The context can be changed only in `Action` parsers, therefore the action blocks of `Action` are analyzed to search for mutating operations. If they mutate the context, they are marked as context-sensitive (`+contextSensitive`). The dedicated indentation combinators `Indent` and `Dedent` are also marked as context sensitive. Finally, all the combinators delegating to the context-sensitive combinators are marked as context-sensitive:

```
<Any>→*<Any:contextSensitive> ⇒
    <Any+contextSensitive>→*<Any:contextSensitive>
```

The memoization strategy of context-free parser combinators is changed to use only a position in a stream as a memento.

As an example, the sequence `'class' &#space` inside `classToken` is marked as context-free and the position is used instead of a full memento.

[16] Because PetitParser is scannerless, the characters represent tokens.

```
class      ← <Sequence
    <classToken> <id> <body>:contextSensitive>
classToken ← <Sequence
    <'class'> <AndCharClass[\t\n␣]>:contextFree>
```

Therefore `classToken` can be optimized using only position as a memento:

```
classToken
    | memento result predicate |
    memento ← context position.
    result ← self classLiteral.
    result isPetitFailure ifTrue: [
        ↑ result.
    ]
    predicate ← self andSpace.
    predicate isPetitFailure ifTrue: [
        context position: memento.
        ↑ predicate
    ]
    ↑ Array with: result with: predicate
```

5. Performance analysis

In this section we report on performance of compiled parsers compared to the performance of plain PetitParser.[17] We also report on the impact of a particular optimization on the overall performance. Lastly, we provide a detailed case-study of Pharo's native Smalltalk parser.

5.1 PetitParser Compiler

PetitParser Compiler is an implementation of a pc-compiler for PetitParser. The PetitParser Compiler applies the pc-compiler techniques and outputs a class that can serve as a top-down parser equivalent to the input combinator.

PetitParser Compiler is available online[18] for Pharo and Smalltalk/X. It is being used in real environments, for example a language for Live Robot Programming[19] and the Pillar markup language.[20]

Validation. The PetitParser pc-compiler is covered by more than two thousand unit tests. Furthermore, we validated the pc-compiler by taking several existing PetitParser combinators and comparing their results with results produced by the compiled variant of a particular parser. In particular, we validated results of:

- Java parser[21] on OpenJDK 6 source code,[22]
- Smalltalk parser[23] on Pharo source code,[24]
- Ruby and Python semi-parsers[25] on several github projects: Cucumber, Diaspora, Discourse, Rails, Vagrant, Django, Reddit and Tornado.

The validation corpus with all the validation test-cases is available online.[26]

5.2 Benchmarks

We measure performance on the following benchmarks:

1. **Identifiers** is a microbenchmark measuring the performance of a simple parser. The input consists of a list of identifiers. The parser contains one single grammar rule — a repetition of an identifier tokens.
2. **Arithmetics** is a benchmark measuring performance while parsing arithmetic expressions. The input consists of expressions with operators `(`, `)`, `*`, `+` and integer numbers. The brackets must be balanced. The operator priorities are considered. Backtracking is heavily used as the grammar is not refactored to be LL(1). The parser contains eight rules.
3. **Smalltalk** is a benchmark measuring performance of a Smalltalk parser. The input consists of a source code from a Pharo 5 image.[27] The parser contains approximately eighty rules.
4. **Java** is a benchmark measuring the performance of a Java parser. The input consists of standard JDK library files. The parser contains approximately two hundred rules.
5. **Python** is a benchmark measuring the performance of an indentation-sensitive semi-parser. The input consists of several github Python projects.[28] The parser contains approximately forty rules. The parser utilizes islands (Moonen 2001) — it is not a full Python parser, but extracts structural elements and skips the rest.

The presented benchmarks cover a variety of grammars from small ones to complex ones, ranging in size from one grammar rule to two hundred. They cover grammars with possibly unlimited lookahead (arithmetic expressions) and almost LL(1) grammars (Smalltalk).[29] They also cover standard grammars

[17] The pre-prepared image with sources and benchmarks can be downloaded from `http://scg.unibe.ch/research/petitcompiler/iwst2016`.

[18] `http://scg.unibe.ch/research/petitcompiler`

[19] `http://pleiad.cl/research/software/lrp`

[20] `http://smalltalkhub.com/#!/~Pier/Pillar`

[21] `http://smalltalkhub.com/#!/Moose/PetitJava/`

[22] `http://download.java.net/openjdk/jdk6`

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

[24] `http://files.pharo.org/get-files/50/sources.zip`

[25] `http://smalltalkhub.com/#!/~JanKurs/PetitParser`

[26] `http://scg.unibe.ch/research/petitcompiler/iwst2016`

[27] `http://files.pharo.org/get-files/50/sources.zip`

[28] Cucumber, Diaspora, Discourse, Rails, Vagrant, Django, Reddit and Tornado.

[29] The implementors did not bother to make it LL(1) as parser combinators allow for unlimited lookahead.

(Java, Smalltalk), island grammars, context-free grammars (Identifier, Java, Smalltalk), and context-sensitive ones (Python).

How we measured. We ran each benchmark ten times using the latest release of the Pharo VM for Linux. We measured only parse time, all parsers and inputs being initialized in advance. We computed mean and standard deviation of the results.

We report on the speedup (ration between original PetitParser and compiled version) and average time per character. The speedup and time per character are represented by boxes. The standard deviation is visualized using error bars.

Results. The speedup of a compiled version compared to an original version is shown in Figure 3. We also visualize the average time per character for each of the grammars in Figure 4.

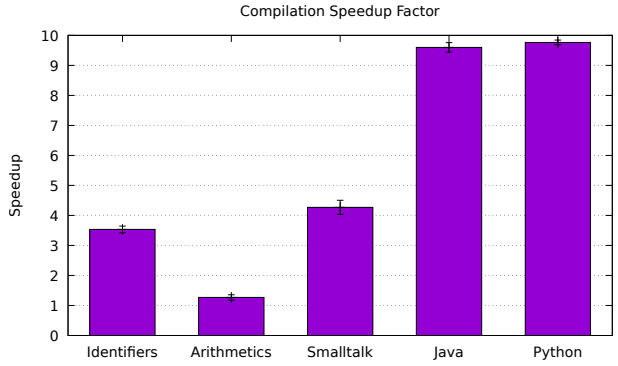

Figure 3. The speedup of compilation for different grammars.

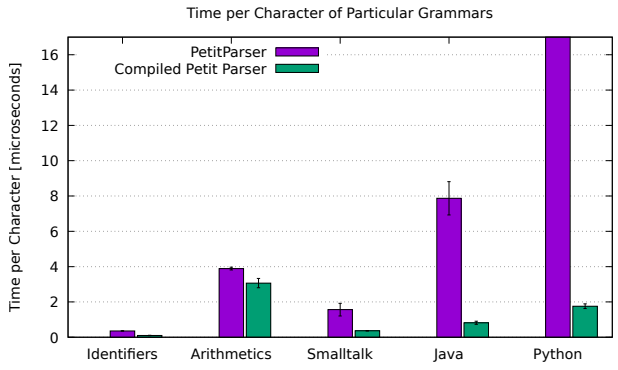

Figure 4. The time per character for compiled and non-compiled versions of the measured grammars.

The speedup in Identifiers is caused by the fact that we avoid unused collection allocations when parsing the identifier token. The speedup of Arithmetics is only a factor of two. We attribute such a mediocre result to the fact that Arithmetics performs a lot of backtracking and the simplicity of the grammar does not allow for many optimizations.

The trend for Arithmetics, Smalltalk and Java grammars shows that the more complex the grammar is, the more speedup is gained. The Python grammar is not very complicated, yet it is very slow (see time per character in Figure 4). This is caused by the overhead of copying an indentation stack, which our pc-compiler mostly removes.

5.3 Performance Details

The speedup with a particular optimization turned off is shown in Figure 5. We also visualize the average time per character in Figure 6. The graphs show how much a particular optimization contributes to the overall performance of the grammars.

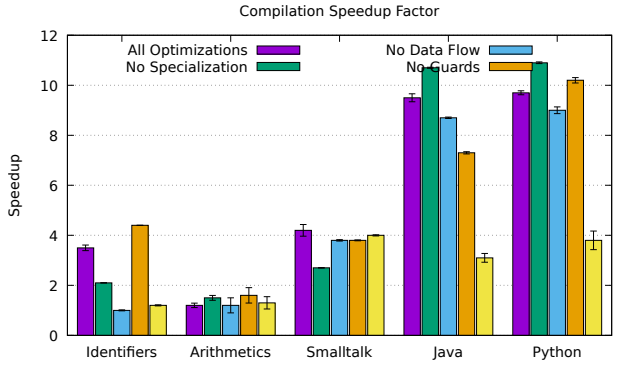

Figure 5. The speedup of compilation for different grammars with a specific optimization turned off.

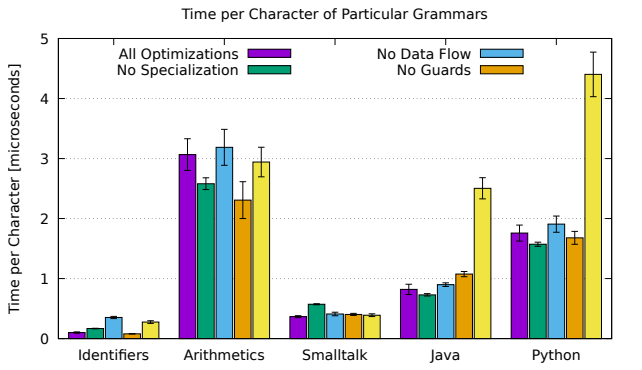

Figure 6. The time per character for compiled and non-compiled versions of the measured grammars with specific optimization turned off.

Different optimizations have different effects on the benchmarked grammars. The Identifiers grammar is optimized mostly by specializations and data-flow optimizations.

Interestingly, the Arithmetics grammar performs even better when guards are turned off. This can be explained by the fact that tokens in the Arithmetic grammar are very short and the guards may do redundant work, so their early reporting on failures does not outweigh this overhead.

The Smalltalk grammar is slightly improved by each of the optimizations. The major impact for the Java grammar is caused by the context analysis which suggests that the Java grammar creates a lot of mementos. The fact that specializations have no impact on the Java grammar indicates that the tokens in Java are complex and not sufficiently optimized in the current implementation.

The Python grammar is optimized only by the context analysis. Probably, other optimizations are not suitable for a semi-parsing nature of a grammar.

5.4 Smalltalk Case Study

In this case study we compare performance of a Smalltalk parser implemented in PetitParser (plain and compiled versions) and that of a hand-written Smalltalk parser used natively in Pharo. 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. **PetitParser Compiled** is a compiled version of the above parser.
3. **Hand-written parser** is a parser used natively by Pharo. It is a hand-written and optimized parser and contrary to the previous parsers, it utilizes a scanner. This parser can serve as a baseline. It is probably close to the optimal performance of a hand-written parser as it is heavily used throughout the system and has therefore been extensively optimized by Pharo developers.
4. **SmaCC** is a scanning and table-driven parser compiled by a SmaCC tool (Brant and Roberts) from a LALR(1) Smalltalk grammar.

The speedup comparison is shown in Figure 7. Average time per character for each of the parsers is shown in Figure 8.

The native parser is approximately five times faster than its PetitParser counterpart. The compiled version is approximately four times faster. The SmaCC parser is approximately two times faster. The native parser's time per character is 0.27μs, the compiled parser's time is 0.33μs, the SmaCC parser time is 0.59μs, while PetitParser's time is 1.33μs. The compiled version is approximately 20% slower than the hand-written parser and two times faster than the SmaCC parser.

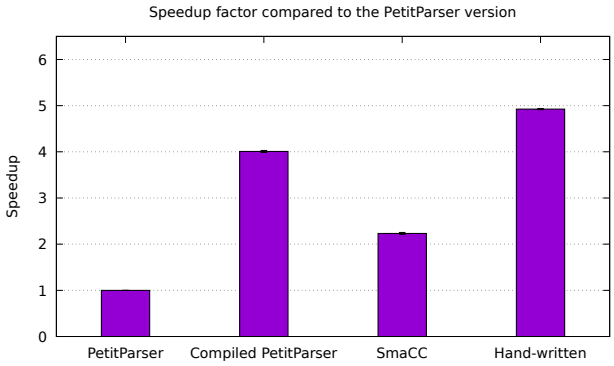

Figure 7. Performance speedup of Smalltalk parsers

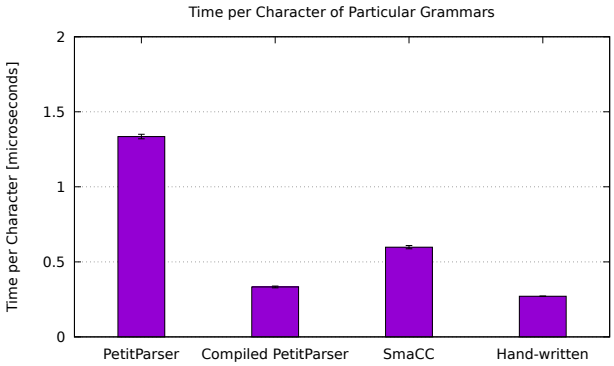

Figure 8. Time per character of Smalltalk parsers

6. Discussion and Related Work

In this section we discuss limitation of a pc-compiler and other approaches focusing on high-performance parsers.

6.1 Limitations

Even though the pc-compiler is designed with flexibility in mind, there are some limitations. The pc-compiler does not compile non-functional methods, *i.e.*, parsers that invokes methods referencing instance variables. The pc-compiler also cannot handle grammar adaptations, *i.e.*, situations, when the parser graph changes *on-the-fly*. Last but not least, the support for hand-tuning is in the current implementation limited and still experimental.

6.2 Related Work

There has been recent research in Scala parser combinators (Odersky 2007; Moors et al. 2008) 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* (Béguet and Jonnalagedda 2014) the authors argue to use

macros (Burmako 2013) to remove the composition overhead. In *Staged Parser Combinators for Efficient Data Processing* (Jonnalagedda et al. 2014) the authors use a multi-stage programming (Taha 2003) framework LMS (Rompf and Odersky 2010) to eliminate intermediate data structures and computations associated with a parser composition. Both works lead to a significant speedup at least for the analyzed parsers: an HTTP header parser and a JSON parser.

Similarly to our approach, ahead-of-time optimizations are applied to improve the performance. In contrast, our work does not utilize meta-programming to manipulate compiler expression trees in order to optimize parser combinators. Instead we implemented a dedicated tool from scratch. In our work, we consider several types of optimizations guided by a need to produce fast and clean top-down parsers.

Other approaches leading to better combinator performance are memoization (Frost and Szydlowski 1996) and Packrat Parsing (Ford 2002) (already utilized by PetitParser). In *Efficient combinator parsers* (Koopman and Plasmeijer 1998) Koopman *et al.* use the continuation-passing style to avoid intermediate list creation.

There are table-driven or top-down parser generators such as YACC (Johnson 1975), SmaCC (Brant and Roberts), Bison (Levine 2009), ANTLR (Parr and Quong 1995) or Happy (Happy) 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 (Pennello 1986).

Our work is also related to compilers supporting custom DSLs and providing interfaces for optimizations, *e.g.*, Truffle (Humer et al. 2014). Yet our approach is focused on concrete optimization techniques for parser combinators and we do not aspire for general DSL support.

7. Conclusion

In this work we present a pc-compiler, an ahead-of-time optimizer for PetitParser. The pc-compiler performance speedup ranges from a factor of two to ten while preserving the advantages and flexibility of PetitParser. Based on our Smalltalk case study, the pc-compiler provides two times better performance than a table-driven parser compiled by SmaCC, and approximately 20% worse performance than a hand-written optimized parser.

Even though the results of our PetitParser pc-compiler are satisfying, there is a lot of room for improvements. The Arithmetics grammar was not compiled into an optimal one and the Smalltalk grammar is still two times slower than the hand-written parser.

```
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 2. Implementation of `ActionParser`.

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

Listing 3. Implementation of `AndPredicate`.

The Java and Python grammars were not sufficiently optimized by specializations as well. We would like to focus on these shortcomings and improve them.

We would like to apply some grammar rewriting rules (*e.g.*, to refactor choices to be LL(1)), add more advanced specializations, improve the support for semi-parsers, add smarter guards that do not perform redundant operations and improve the support for user optimizations. Last but not least, we would like to experiment with just-in-time optimizations.

Acknowledgments

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).

A. 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.

References

M. D. Adams and O. S. Ağacan. Indentation-sensitive parsing for Parsec. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 121–132, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3041-1. doi: 10.1145/2633357.2633369. URL `http://doi.acm.org/10.1145/2633357.2633369`.

```
CharClassParser>>parseOn: context
    (context atEnd not and:
     [predicate value: context peek]) ifTrue: [
        ↑ context next
    ] ifFalse: [
        ↑ PPFailure message: 'Predicate expected'
    ]
```

Listing 4. Implementation of `CharClassParser`.

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

Listing 5. Implementation of `ChoiceParser`.

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

Listing 6. Implementation of `IndentParser`.

```
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 7. Implementation of `SequenceParser`.

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

Listing 8. Implementation of `StarParser`.

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

    result isPetitFailure ifTrue: [
        context restore: memento
        result
    ].
    whitespace parseOn: context.

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

Listing 9. Implementation of `TokenParser`.

```
Context>>remember
    ↑ Memento new
        position: position
        indentationStack: stack copy

Context>>restore: memento
    self position: memento position
         indentationStack: stack copy
```

Listing 10. Implementation of `Context` remember and restore.

A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling Volume I: Parsing*. Prentice-Hall, 1972. ISBN 0-13-914556-7.

R. C. Backhouse. *Syntax of Programming Languages: Theory and Practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979. ISBN 0138799997.

E. Béguet and M. Jonnalagedda. Accelerating parser combinators with macros. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA '14, pages 7–17, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2868-5. doi: 10.1145/2637647.2637653. URL `http://doi.acm.org/10.1145/2637647.2637653`.

J. Brant and D. Roberts. SmaCC, a Smalltalk Compiler-Compiler. URL `http://www.refactory.com/Software/`

SmaCC/. http://www.refactory.com/Software/SmaCC/.

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, pages 3:1–3:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1. doi: 10.1145/2489837.2489840. URL http://doi.acm.org/10.1145/2489837.2489840.

H. Christiansen. Adaptable grammars for non-context-free languages. In J. Cabestany, F. Sandoval, A. Prieto, and J. Corchado, editors, *Bio-Inspired Systems: Computational and Ambient Intelligence*, volume 5517 of *Lecture Notes in Computer Science*, pages 488–495. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02477-1. doi: 10.1007/978-3-642-02478-8_61. URL http://dx.doi.org/10.1007/978-3-642-02478-8_61.

R. I. Fabio Mascarenhas, Sérgio Medeiros. On the relation between context-free grammars and parsing expression grammars. *CoRR*, abs/1304.3177, 2013. URL http://arxiv.org/abs/1304.3177.

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*, volume 37/9, pages 36–47, New York, NY, USA, 2002. ACM. doi: 10.1145/583852.581483. URL http://pdos.csail.mit.edu/~baford/packrat/icfp02/packrat-icfp02.pdf.

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*, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964011. URL http://pdos.csail.mit.edu/~baford/packrat/popl04/peg-popl04.pdf.

R. A. Frost and B. Szydlowski. Memoizing purely functional top-down backtracking language processors. *Science of Computer Programming*, 27(3):263–288, Nov. 1996. ISSN 01676423. doi: 10.1016/0167-6423(96)00014-7.

R. A. Frost, R. Hafiz, and P. C. Callaghan. Modular and efficient top-down parsing for ambiguous left-recursive grammars. In *Proceedings of the 10th International Conference on Parsing Technologies*, IWPT '07, pages 109–120, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics. ISBN 978-1-932432-90-9. URL http://dl.acm.org/citation.cfm?id=1621410.1621425.

D. Grune and C. J. Jacobs. *Parsing Techniques — A Practical Guide*. Springer, 2008a. ISBN 038720248X. URL http://www.cs.vu.nl/~dick/PT2Ed.html.

D. Grune and C. J. Jacobs. *Parsing Techniques — A Practical Guide*, chapter 8: Deterministic Top-Down Parsing, pages 235–361. Volume 1 of Grune and Jacobs (2008a), 2008b. ISBN 038720248X. URL http://www.cs.vu.nl/~dick/PT2Ed.html.

Happy. Happy — the parser generator for Haskell, 2010. URL https://www.haskell.org/happy/. http://tfs.cs.tu-berlin.de/agg/index.html.

U. Hölzle. Adaptive optimization for self: Reconciling high performance with exploratory programming. Technical report, Stanford, CA, USA, 1994.

C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A domain-specific language for building self-optimizing AST interpreters. *SIGPLAN Not.*, 50 (3):123–132, Sept. 2014. ISSN 0362-1340. doi: 10.1145/2775053.2658776. URL http://doi.acm.org/10.1145/2775053.2658776.

G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996. URL citeseer.ist.psu.edu/hutton96monadic.htmlhttp://eprints.nottingham.ac.uk/237/1/monparsing.pdf.

S. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

M. Jonnalagedda, T. Coppey, S. Stucki, T. Rompf, and 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*, pages 637–653, New York, New York, USA, 2014. ACM Press. doi: 10.1145/2660193.2660241.

P. Koopman and R. Plasmeijer. Efficient combinator parsers. In *In Implementation of Functional Languages, LNCS*, pages 122–138. Springer-Verlag, 1998.

J. Kurš, G. Larcheveque, L. Renggli, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. PetitParser: Building modular parsers. In *Deep Into Pharo*, page 36. Square Bracket Associates, Sept. 2013. ISBN 978-3-9523341-6-4. URL http://scg.unibe.ch/archive/papers/Kurs13a-PetitParser.pdf.

J. Kurš, M. Lungu, and O. Nierstrasz. Top-down parsing with parsing contexts. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014. URL http://scg.unibe.ch/archive/papers/Kurs14a-ParsingContext.pdf.

J. Levine. *Flex & Bison: Text Processing Tools*. O'Reilly Media, 2009. ISBN 9781449391973. URL https://books.google.ch/books?id=nYUkAAAAQBAJ.

L. Moonen. Generating robust parsers using island grammars. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings Eighth Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society, Oct. 2001. doi: 10.1109/WCRE.2001.957806. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.1027.

A. Moors, F. Piessens, and M. Odersky. Parser combinators in Scala. Technical report, Department of Computer Science, K.U. Leuven, Feb. 2008.

M. Odersky. Scala language specification v. 2.4. Technical report, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, Mar. 2007.

T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25:

789–810, 1995. doi: 10.1002/spe.4380250705. URL http://www.antlr.org/article/1055550346383/antlr.pdf.

T. J. Pennello. Very fast LR parsing. *SIGPLAN Not.*, 21(7):145–151, July 1986. ISSN 0362-1340. doi: 10.1145/13310.13326. URL http://doi.acm.org/10.1145/13310.13326.

R. R. Redziejowski. Applying classical concepts to parsing expression grammar. *Fundam. Inf.*, 93(1-3):325–336, Jan. 2009. ISSN 0169-2968. URL http://dl.acm.org/citation.cfm?id=1576070.1576093.

d. S. Reis, L. Vieira, d. S. Bigonha, Roberto, D. Iorio, V. Oliveira, de Souza Amorim, and L. Eduardo. Adaptable parsing expression grammars. In F. de Carvalho Junior and L. Barbosa, editors, *Programming Languages*, volume 7554 of *Lecture Notes in Computer Science*, pages 72–86. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33181-7. doi: 10.1007/978-3-642-33182-4_7. URL http://dx.doi.org/10.1007/978-3-642-33182-4_7.

L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, pages 1–4, Malaga, Spain, June 2010. URL http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf.

T. Rompf and 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, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868314. URL http://doi.acm.org/10.1145/1868294.1868314.

W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. URL http://www.cs.uu.nl/people/visser/ftp/P9707.ps.gz.

P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995.

A. Warth, J. R. Douglass, and T. Millstein. Packrat parsers can support left recursion. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 103–110, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-977-7. doi: 10.1145/1328408.1328424. URL http://doi.acm.org/10.1145/1328408.1328424.

P. R. Wilson. Uniprocessor garbage collection techniques, 1992.