Layout Sensitive Parsing in Petit Parser Framework

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Attieh Sadeghi Givi
October 30, 2013

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Jan Kurš
Institut für Informatik und angewandte Mathematik

Acknowledgments

I would like to thank my supervisor Jan Kurš for his steady input of new ideas – Prof. Oscar Nierstrasz for his calm way of handling things – my family for giving me positive energy – my husband for a nonstop support through thick and thin – Aubergine and unknown elements for reminding me my allergy – iranian people for electing Dr. Hasan Rohani as their president – Candy crush for keeping me awake – God for giving me everything.

Abstract

Most parser frameworks can parse a context-free language generated by different context-free grammars. However, many languages are not context-free. One important class of such languages is layout-sensitive languages (*e.g.* Python, Haskell), in which the structure of code depends on indentation and whitespace. The parsers (and lexers) of this kind of languages are not declaratively specified but hand-tuned to account for layout-sensitivity. To support parsing of layout-sensitive languages, we propose an extension of parsing expression grammars in which a user is able to declare layout-sensitive specifications. For example, a user can declare a consisting block of statements to be aligned and arbitrary positioned. We have implemented our extension in a Petit Parser framework. We evaluate the correctness and performance of our parser by parsing Python- and Haskell-like grammars.

Contents

1 Introduction

A parser structures a computer language by translating its textual representation, in order to facilitate the writing of compilers and interpreters. Language specifications are declarative and can be parsed by any parser. But one particular class of languages is not declaratively supported by any existing parser framework, namely layout-sensitive languages.

Layout-sensitive languages were first proposed by Landin in 1966[12]. In layout-sensitive languages, the structure of a translated textual representation depends on the code's layout and its indentation. Languages such as Python and Haskell use the indentation and layout of a code as part of their syntax. In the following example, we see a Python and Haskell program declaring the code's block structure by using the layout.

```
-- Python: Nested block structure.
if x <= y:
    if x == y:
        print y
else:
    print x

-- Haskell: Nested block structure.
if number < 0
    then do print "You win!"
            print "number is too low!"
```

Listing 1.1: Layout-sensitive languages indentation example

The layout of the Python program shows that the first `else` belongs to the outer if-statement. Similarly, the layout of the Haskell program shows to which do-block each statement belongs. It also shows the vertically aligned statement block starting at the same column in the code. Both of them use indentation instead of curly braces. Unfortunately, indentation is a context-sensitive feature and most of the research in grammars and parsing technologies were done for context-free grammars. Therefore, the programmer has to write *ad hoc* solutions to deal with layout and indentation. For example, Haskell introduces a new stage in the scanner-parser pipeline and Python extends the scanner with a stack to generate special indentation-specific tokens.

Recently, some research in layout-sensitive grammars has appeared. Erdweg *et al.* [4] proposed declarative layout-sensitive extensions for generalized LR parsing.

Adams[1] described an algorithm to create layout-sensitive LR(k) parsers. Mühlbacher and Brunauer[3] suggested an extension to context-free grammars and described how to construct the top-down layout-sensitive parser from such a grammar.

In this work we focus on layout-sensitive parsing expression grammars (PEGs) [6]. We suggest new parsing expressions, that allow us to define layout-sensitive grammars in PEGs. In addition, we describe implementation of such expressions and validate our approach with Python- and Haskell-like grammars.

The remainder of this paper is structured as follows: Chapter 2 briefly introduces the differences between context-free and context-sensitive grammars (respectively languages), and describes the parsing expression grammar and through examples, presents our idea of the extension. Chapter 3 introduces the Petit Parser framework, describes new implementation of its operators and expressions and functions that have to be updated. Chapter 4 demonstrates the proof and validation of the extended PEG with Python- and Haskell-like grammars. Chapter 5 presents an overview of the related works and Chapter 6 concludes the paper.

2
Parsing Expression Grammars

In this chapter we, *first)* briefly discuss the differences between grammars according to their context (free or sensitive) and related derived languages, *second)* shortly introduce the parsing expression grammar (PEG), which is used for our implementation and proof of the work and *third)* explain our idea of PEGs extension in order to be able to parse layout-sensitive languages like Python[13] and Haskell[9].

2.1 Context-free grammar (CFG) vs. Context-sensitive grammar (CSG)

Grammars generate languages or are a set of production rules for strings in a formal language (a set of strings of symbols that may be constrained by rules that are specific to it). In order to identify if a particular string is contained in the language, we can use an automaton as a recognizer. If there are multiple ways of generating the same single string, the grammar is said to be ambiguous.

For example, consider the following production rules: *1)* $S \to aSb$ *2)* $S \to ab$
We can choose a rule to create a string applying to S. If we choose two times the first rule followed by the second rule, we obtain the following string *aaabbb* (using symbols: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$). This grammar defines the Language as
$L(G) = \{\ a^n abb^n \mid n \geq 0\ \} = \{\ ab, aabb, aaabbb, aaaabbbb, ...\}$, where a^k is *a* repeated *k* times and *n* represents the number of times production rule 1 has been applied.

A context-free grammar (CFG)[1] is a grammar where each production rule has the form $\alpha \to \beta$, where α is a nonterminal and β is a string of terminals and/or nonterminals (β can be empty) and the nonterminal α can always be replaced by the right hand side. The formalism of context-free grammars was developed in the mid-1950s by Noam Chomsky[8]. A context-free language (CFL)[2] can be generated by different CFGs and vice versa. Some important subclasses of CFGs are:

i) LR(k) grammars[2] also known as deterministic context-free grammars (DCFG) can only describe a deterministic context-free language and parse with deterministic pushdown automata (PDA), which accept precisely the context-free languages.

[1]http://en.wikipedia.org/wiki/Context-free_grammar
[2]http://en.wikipedia.org/wiki/LR_parser

ii) LL(k) grammars[3] describe fewer languages and parse by direct construction of a leftmost derivation.

iii) Linear grammars[4] have no rules with more than one nonterminal in the right hand side. A subclass of the linear grammars are regular grammars[5], which describe regular languages.

```
-- Terminals→[,],(,),{,}
-- Nonterminal→S
-- The following sequence can be derived in this grammar
   {([ [ {[ ()() {} [ ][ ] ]} ]([ ]) ])}

S→S,S       S→(,S,)      S→[,S,]      S→{,S,}
S→(,)      S→[,]      S→{,}
```

Listing 2.1: Context-free grammar example: well-formed nested parentheses, square brackets and braces

A context-sensitive grammar (CSG)[6] is a grammar where each production has the form α,A, $\beta \rightarrow \alpha$,y, β, where α and β are strings of terminals and nonterminals, and y is a nonempty string of terminals and nonterminals, and A is a nonterminal. The fact of being context-sensitive is explained by forming the context of A with α and β. An additional rule is the $S \rightarrow \lambda$, where λ represents the empty string and S does not appear on the right-hand side of any permitted rule. The important difference from context-free grammar is that the context of a nonterminal A is needed to determine whether it can be replaced by y or not. The context-sensitive language (CSL)[7] can be defined by CSG and is equivalent to linear bounded automaton (LBA). It also should be noted that every context-free language is context-sensitive but not vice versa.

```
-- Terminals→a,b,c
-- Nonterminals →S,B,C,H
-- This grammar generates the following language
    {a^n b^n c^n | n ≥ 1}

S → a,S,B,C     S → a,B,C     C,B → H,B
H,B → H,C       H,C → B,C
a,B → a,b
b,B → b,b       b,C → b,c
c,C → c,c
```

Listing 2.2: Context-sensitive grammar example

To determine whether a string is contained in a CFG or a CSG, there exist parsing algorithms such as the pushdown automaton (PDA) for CFGs and the linear bounded automaton (LBA) for CSGs. More efficient parse algorithms for CFGs are for example, LL(k) or LR(k) parsers, which enable linear time parsing. Another parsing grammar that has a simplified syntax definition (because of the ordered choice operator), lacks the ambiguity of CFG, provides a rich set of operators for constructing grammars, avoids the complexity and fickleness of LR parser and can be parsed in linear time using a tabular or memoizing parser, is the parsing expression grammar (PEG), which will be further discussed in Section 2.2.

[3]http://en.wikipedia.org/wiki/LL_parser
[4]http://en.wikipedia.org/wiki/Linear_grammar
[5]http://en.wikipedia.org/wiki/Regular_grammar
[6]http://en.wikipedia.org/wiki/Context-sensitive_grammar
[7]http://en.wikipedia.org/wiki/Context-sensitive_language

2.2 Parsing expression grammar

The parsing expression grammars (PEGs) formalism was first introduced by Ford in 2004[6]. It recognizes strings in the language according to the described analytic formal grammar and is (compared to other grammars such as context-free grammars (CFGs) (see Section 2.1)) not ambiguous, because the first matched choice will be selected by the operator and the second alternative is ignored. Instead of using an unordered choice operator '|' in expression grammars, PEGs use a prioritized choice operator '/' to select the first successful match.

```
S→ab|a      S→a|ab    —— both equivalent in CFG
S←ab/a      S←a/ab    —— different rules in PEG
```

Listing 2.3: Comparing choice operators in CFG and PEG

In Listing 2.3, in the first line the extended Backus-Naur form (EBNF)[8] rules are equivalent in a CFG, but the PEG rules in the second line are different. In fact if the input string to be recognized begins with 'a', the first choice is always taken and the second choice in both rules will never succeed. The PEG consists of a set of definitions of the form *'$A \leftarrow e$'*, where A is a nonterminal and e is a parsing expression. Basic operators for constructing the parsing expressions used in this work are mentioned in Table 2.1. The single *'abc'* and double *"abc"* quotes delimit string literals, and square brackets '[ ]' indicate character classes. The constant '.' matches any single character. The '?', '*' and'+' operators behave as in common regular expression syntax, however, these operators behave more greedily, consuming as much input as possible and never backtracking. The expression '&e' attempts to match pattern e, and '!e' fails if e succeeds and succeeds if e fails. The sequence expression'e_1 , e_2' looks for a match of e_1, immediately followed by a match of e_2, backtracking to the starting point if either pattern fails. The choice operator 'e_1 / e_2' first attempts pattern e_1, and then e_2 from the same starting point if e_1 fails. The PEG expressions are constructed from terminal expressions and operators, which can form complex composite expressions.

Operator	**Description**
$'abc'$	Literal string
$''abc''$	Literal string
[ ]	Character class
.	Any character
(e)	Grouping
$e?$	Optional, also (e optional)
$e*$	Zero-or-more, also (e star)
$e+$	One-or-more, also (e plus)
$\&e$	And-predicate, also (and e)
$!e$	Not-Predicate, also (not e)
e_1 , e_2	Sequence
e_1 / e_2	Prioritized choice

Table 2.1: Operators for constructing parsing expressions

```
—— Matches the following language  {a^n b^n: n≥1}
—— Terminals: a,b        Nonterminals: S
S ← a,S?,b
——Matches the following language    {a^n b^n c^n: n≥ 1}
——Terminals : a,b,c     Nonterminals : S, A, B
```

[8]http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form

```
S ← &(A,c),a+,B,!(a/b/c)
A ← a,A?,b
B ← b,B?,c
```

Listing 2.4: PEG forming complex composite expressions example

As an example in Listing 2.4 for a complex composite expression, in the first production rule, *S* is a nonterminal, *'a'* and *'b'* are terminals, and *S?* is an optional expression, which consumes as much input as possible, and the expression grammar describes the simple context-free matching language { $a^n b^n$: n $\geq$ 1 }. In the second group of rules, *S, A* and *B* are nonterminals, *'a', 'b'* and *'c'* are terminals, *'&'* attempts to match *A* followed by a literal string *'c'*, and *B* fails if it is followed by one of these terminals *'a', 'b'* or *'c'*, and the expression grammar describes the classic non-context-free language { $a^n b^n c^n$: n $\geq$ 1 }.

Another classic example is the inescapable "dangling ELSE"[9] problem, which requires either an informal meta-rule or severe expansion and obfuscation of the CFG. Instead of determining whether two alternatives in a CFG are ambiguous, PEGs easily express this by reordering with the prioritized choice operator '/' without affecting the language. Examples are shown as follows:

```
-- Simple, not ambiguous if-else statement
Statement ← 'if',a,'then',s
Statement ← 'if',a,'then',s1,'else',s2

-- Nested, ambiguous if-else statement
Statement ← 'if',a,'then','if',b,'then',s,'else',s2

-- Not ambiguous, nested if-else statement in PEG
Statement ← 'if',a,'then','if',b,'then',s,'else',s2
           /'if',a,'then',s1,'else',s2
           /'if',a,'then',s
```

Listing 2.5: "dangling ELSE" problem

To obtain a better performance for any parsing expression grammar, the recursive descent parser can be converted to a packrat parser, which always runs in linear time, at the cost of substantially greater storage space requirements.

2.2.1 Packrat Parsing

Packrat parsing is a parsing technique introduced by Ford[5]. To exclude extra parsing steps, PEG is mostly executed by packrat parsing, which uses memoization (an optimization technique that avoids repeating the calculation of results for previously processed inputs), requires an amount of memory proportional to the length of the input, parses the PEGs in a linear time. Any language defined by an LL(k) or LR(k) grammar can be recognized by a packrat parser, and also others that are not LR but require unlimited look ahead.

Packrat parsing provides the simplicity, elegance, and generality of the backtracking model (a top-down strategy that instead of making decisions speculatively by trying different alternatives in succession, it "backtracks" to the original input position, if one fails to match, and tries another) but eliminates the risk of super-linear parse time, by saving all intermediate parsing results as they are computed and ensuring that no result is evaluated more than once. This will be more discussed and illustrated in subsection 3.4.1.

[9] `http://en.wikipedia.org/wiki/Dangling_else`

2.3 Layout-sensitive parsing expression grammar

Layout-sensitive languages were first proposed by Landin in 1966[12]. Many languages such as Haskell and Python use the indentation and layout of code as part of their syntax and because CFGs cannot express the indentation rules, parsers use ad hoc techniques (*e.g.* they are often coded by hand instead of being generated by a parser generator) to handle the layout. Landin[12] introduced the concept of the off-side rule for indentation, which requires that all tokens in an expression be indented at least as far as the first token of the expression. Examples are shown as follows:

```
--(We replace tabulators with whitespaces)
-- e1: x = a, e2: y = b
-- e1 starts on a separate line, e1 and e2 are exactly
   aligned (correct indentation)
let
    x = a
    y = b

-- e1 starts inline, e1 and e2 are exactly aligned
     (correct indentation)
let x = a
    y = b

-- e1 starts inline, e1 and e2 are not aligned
       (wrong indentation)
let x = a
   y = b
```

Listing 2.6: Off-side rule concept

In the first two examples in Listing 2.6, expression 1 and 2 are exactly aligned and correctly indented, but in the third example, expression 2 is not correctly indented, therefore it is not following the off-side concept.

2.3.1 Indentation-sensitive languages

This section will present layout rules of some indentation-sensitive languages like Haskell and Python.

Python [13]

It is explicitly line oriented and features new line in its grammar as a terminal that separates statements. The grammar uses indent and dedent tokens to delimit indentation-sensitive forms. An indent token is emitted by the parser (and lexer) whenever the start of a line is at a strictly greater indentation than the previous line. Matching dedent tokens are emitted when a line starts at a lesser indentation. In Python, indentation is used only to delimit statements, and there are no indentation-sensitive forms for expressions. Normally, each new line of Python code starts a new statement. If, however, the preceding line ends in a backslash (\), then the current line is "joined" with the preceding line and is a continuation of the preceding line. Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. The indentation of the continuation lines is not important. This means that indent and dedent tokens must not be emitted by the lexer between paired delimiters.

Haskell [9]

Its indentation-sensitive blocks (e.g. the bodies of do, case, or where expressions) are made up of one or more statements or clauses that are not only indented relative to the surrounding code but are also indented to the same column as each other. Thus, lines that are more indented than the block continue the current clause, lines that are at the same indentation as the block start a new clause, and lines that are less indented than the block are not part of the block. In addition, semicolons (;) and curly braces ({and}) can explicitly separate clauses and delimit blocks, respectively. Explicitly delimited blocks are exempt from indentation restrictions arising from the surrounding code.

Haskell and Python parsers (and lexers) are not declaratively specified but hand-tuned to account for layout-sensitivity, therefore Erdweg[4] *et al.* proposed a parsing framework in which a user can annotate the layout in a grammar as constraints on the relative positioning of tokens in the parsed subtrees. Adams[1] presented a simple extension to CFGs that expressed these layout rules and derives GLR and LR(k) algorithm for parsing these grammars. Instead we try to present an extension of PEG, which is used by Petit Parser to parse these kind of indentation-sensitive languages. More is explained in subsection 2.3.2 and Chapter 3.

2.3.2 The basic idea of the extension

Petit Parser is a parsing framework (more in Section 3.1), which uses PEGs to parse all indentation-insensitive languages but not indentation-sensitive languages,and in which the structure of a code depends on indentation and whitespace instead of curly braces (see Listing 2.7).

```
-- Nested if-statement in indentation-insensitive
    language (e.g. Java)
if (x <= y) {
    if (x== y)
            System.out.println(y);
    else
            System.out.println(-x);
}
else
    System.out.println(x);

-- Nested if-statement without curly braces
if (x <= y)
    if (x == y)
        System.out.println(y);
else
    System.out.println(-x);
    else
        System.out.println(-x);

-- Nested if-statement in indentation-sensitive language
          (e.g. Python)
if x <= y:
    if x == y:
        print y
else:
    print -x
    else:
        print x
```

Listing 2.7: If-statement in Java and Python

All examples in Listing 2.7, which illustrate a nested if-statement in Java- and Python-like grammars will be parsed in Petit Parser without any errors, since it does not care for indentation levels and considers the spaces as whitespaces. Instead if we try to run the third example written in Python with a Python interpreter, it fails due to a syntax error, since it differentiates between whitespaces, indents and dedents, and the first and second `else` are not properly indented dedented respectively. In order to parse such indentation-sensitive languages we suggest an extension of PEG as follows:

Terminal Expression	**Description**
␣	Space
→	Tabulation
↔	Whitespace (space or tabulation)
↩	New line

Table 2.2: Terminal expressions used to define layout-sensitive expression grammars

Terminal Expression	**Description**
$\rhd$	Indent
$\lhd$	Dedent
$\unrhd$	Set Indent level
$\unlhd$	Remove Indent level

Table 2.3: New Terminal expressions used to define layout-sensitive expression grammars

New required features related to indentation-sensitiveness

Table 2.2 and Table 2.3 represent used expressions for making the explanations more clear (we will use '< >' to emphasize the composite, non-terminal expression). We use similar examples as mentioned in Listing 1.1 and define the process of new required features "line by line". This process makes it easier to follow our idea of the extension.

```
if␣x <= y:↩             //    if ,space ,<e> ,: ,newline
→if␣x == y:↩            //    tab ,if ,space ,<e> ,: ,newline
→→print␣y↩              //    tab ,tab ,<s>,newline
→else:↩                 //    tab ,else ,: ,newline
→→print␣-x↩             //    tab ,tab ,<s>,newline
else:↩                  //    else ,: ,newline
→print␣x↩               //    tab ,<s>,newline
                        //    End of input
```

Listing 2.8: Nested If-statement in Python

Indent ($\rhd$) If we consider the first 3 lines in Listing 2.8,

```
if␣x <= y:↩        // if ,space ,<e> ,: ,newline
→if␣x == y:↩       // tab ,if ,space ,<e> ,: ,newline
→→print␣y↩         // tab ,tab ,<s>,newline
```

line 1: The first `if` starts at [column 0]. Then at the end of the current line we have a new line followed by a tab in the second line.

line 2: The second inner `if` starts at [column 1] and in the current line we have a column level increase by one. Then again at the end of the current line we have a new line followed by a tab in the third line.

line 3: It is followed by another tab, the statement starts at [column 2], in the current line we have again a column level increase by one. Then again at the end of the current line we have a new line followed by a tab in the fourth line.

Indent '▷' is happening, whenever the begin level of a current line is higher than the begin of the previous line. The higher is the begin level of the current line compared to the begin of the previous line, the more indents are signaled.

Dedent (◁) If we consider the following lines in Listing 2.8,

```
→else: ↩                    // tab,:,newline
→→print ␣ -x ↩              // tab,tab,<s>,newline
else: ↩                     // else,:,newline
→print ␣ x ↩                // tab,<s>,newline
                            // End of input
```

line 1: The first inner `else` starts at [column 1], in the current line compared to the previous line we have a column level decrease by one. Then at the end of the current line we have a new line followed by a tab in the fifth line.

line 2: It is followed by another tab, the statement starts at [column 2], again in the current line we have a column level increase by one. Then again at the end of the current line we have a new line followed by a tab in the sixth line.

line 3: The second `else` starts at [column 0], in the current line compared to the previous line we have a column level decrease by two. Then at the end of the current line we have a new line followed by a tab in the seventh line.

line 4: The statement starts [column 1], in the current line compared to the previous line we have a column level increase by one. Then at the end of the current line we have just a new line.

line 5: End of the input is at [column 0], in the current line compared to the previous line we have a column level decrease by one.

Dedent '◁' is happening, whenever the begin level of a current line is lower than the begin level of the previous line (we have an indent before a dedent). The lower is the begin level of the current line compared to the begin of the previous line, the more dedents are signaled. The defined indent and dedent might work with Python-like grammars, since it simulates the indent and dedent tokens generated by Python parser (and lexer) and the rest of the grammar, which is indentation-insensitive and definable with PEGs, but we still need more features to be able to parse Haskell-like grammars.

Move Indent level (⊵, ⊴) If we consider the following example in Listing 2.9,

```
if number < 0 ↩
 ↔ then do print "You win!" ↩
 ↔  ↔  ↔  print "number is too low!"
```

Listing 2.9: Haskell: Do block

line 1: The `if` starts at [column 0]. Then at the end of the current line we have a new line followed by whitespace in the second line.

line 2-3: The `then` starts at [column 1]. It is followed by `do`, then the first statement 'print "You win!"' in our statement block starts at a new column level. It is followed by a new line and some whitespaces, then followed by the second statement 'print "number is too low!"'. The second statement starts also at a new column level. We cannot force the second statement to begin at the same column level as the first statement in a block. Therefore we need to set an indent level while being in a statement block, in order to ensure that all statements begin at the given indent level as the first statement.

The defined indent '▷' can only set an indentation level to a column of the first non-whitespace character on the line, and not to a column in the middle of a line of any Haskell-like grammar. Therefore we need something, which treats separately with a starting block in the middle of a line, to set and remove a new indentation level, without considering the existing whitespaces before and after this block. We define two more features, set indent level '⊵', whenever the current indentation level is set to the column of a current position in a stream and remove indent level '⊴', whenever the current indentation level is removed and the previous one is restored. Since a block is more than a single line, we need more features to preserve the indent level in all lines appearing in the mentioned block.

Preserve indent level If we consider the example in Listing 2.9,

```
1 if number < 0 ↩
2  ↔ then do  ⊵ print "You win!" ↩
3                // ⊵ ,<s>
4             print "number is too low!"
5           ⊴   // <s>PreserveIndent , ⊴
```

line 2-5: The first statement 'print "You win!"' of the statement block starts at [column 9]. We set the indent level '⊵' to the current column position. The second statement in the block starts also at the same level. We sign the end of the block by removing the Indent level '⊴'.

In order to preserve the existing indent level in an expression block in a Haskell-like grammar, we define PreserveIndent as follows:

```
<e>PreserveIndent  ≃ (! ▷ ),(! ◁ )
                     ,( ↔ )*,( ↩ ),( ↔ )*,<e>
```

Listing 2.10: Preserve indent level

Aligned If we consider the following example,

```
if number < 0 ↩
 ↔ then do ↩
          ⊵ print "You win!" ↩ //  ⊵  <s>+ aligned
            print "You win!" ↩
            print "You win!" ↩
            print "You win!"
          ⊴  //  ⊴
```

Whenever we have a sequence or repeating of expressions ('print "You win!"' in the example is repeated 4 times), we take each expression and compare it with the first set indentation level, and ensure if it starts at the same level. We define Aligned as follows:

```
<e>Aligned ≃  ⊵ ,(<e>eachElement replaceWith:
               (<e>PreserveIndent))
              , ⊴

<e>PreserveIndent ≃(!▷),(!◁)
                    ,(↔)*,(↩),(↔)*,<e>
```

Listing 2.11: Aligned

We can also align an expression with another one. We define AlignWith: as follows:

```
<e₁>AlignWith:<e₂> ≃  ⊵
                      ,(<e₁>,(<e₂>PreserveIndent))
                      , ⊴
```

Listing 2.12: AlignWith:

The occurance of the second expression could be optional. We define AlignWithOptional: as follows:

```
<e₁>AlignWithOptional:<e₂> ≃
                ⊵
               ,(<e₁>,(<e₂>PreserveIndent)?)
               , ⊴
```

Listing 2.13: AlignWithOptional:

Since the block position in an indentation-sensitive layout is indented (equally dedented) arbitrarily, we need to define more features.

Arbitrary indentation If we consider the following example,

```
1 if number < 0 ↩
2  ↔ then do ↔ print "You win!"
```

line 2: A statement block can begin in the same line as ‘do’ followed by some whitespaces.

Arbitrary Indentation can be defined as follows:

```
<e>ArbitrarilyIndented ≃ (↔)*,<e>,(↔)*
```

The `((<->)*,<e>,(<->)*)` allows $< e >$ to be placed on the same line with some whitespaces (space or tab) around. There is a second possibility to place a statement block as shown in the following example:

```
1 if number < 0 ↩
2  ↔ then do ↩
3         print "You win!"     //<e>PreserveIndent
```

line 3: A statement block can begin in a new line but with the same indentation level as ‘do’.

Therefore Arbitrary Indentation can also be defined as follows:

```
<e>ArbitrarilyIndented  ≃  <e>PreserveIndent
```

The (<e>PreserveIndent) allows $< e >$ to be placed on a new line with the same indentation level. There is a third possibility to place a statement block as shown in the following example:

```
if number < 0 ↩
 ↔ then do ↩
          ▷print "You win!"
```

line 3: A statement block can begin in a new line but with a different indentation level as ‘do’. Here the indentation level of the statement block is higher than the indentation level of ‘do’.

Therefore Arbitrary Indentation can also be defined as follows:

```
<e>ArbitrarilyIndented  ≃  <eIndent>

<eIndent>← ( ▷ , (<e>/<eIndent>) , ◁ )
```

The <eIndent> is an expression that allows $< e >$ to be indented one or more levels (making sure that all indents ‘▷’ will be consumed by appropriate number of dedents ‘◁’). There is a fourth possibility to place a statement block as shown in the following example:

```
if number < 0 ↩
 ↔ then do ↩
     ◁print "You win!"
```

line 3: A statement block can begin in a new line but with a different indentation level than ‘do’. Here the indentation level of the statement block is lower than the indentation level of ‘do’.

Therefore Arbitrary Indentation can also be defined as follows:

```
<e>ArbitrarilyIndented  ≃  <eDedent>

<eDedent>  ←  ( ◁ , (<e>/<eDedent>) , ▷ )
```

The <eDedent> is an expression that allows $< e >$ to be dedented one or more levels (making sure that all dedents ‘◁’ will be consumed by appropriate number of indents ‘▷’). In order to arbitrarily place a block as in any Haskell-like grammar, we take all defined possibilities and present an Arbitrary Indentation as follows:

```
<e>ArbitrarilyIndented  ≃  (( ↔ )*,<e>,( ↔ )*)
                           /(<e>PreserveIndent)
                           /<eIndentDedent>

<eIndentDedent> ←  (<eIndent> / <eDedent>)
<eIndent> ←  ( ▷ , (<e>/<eIndentDedent>) , ◁ )
<eDedent> ←  ( ◁ , (<e>/<eIndentDedent>) , ▷ )
```

Listing 2.14: Arbitrary Indentation

Trim We finally present our idea for a new feature, which enables us to differentiate between whitespaces, new lines and indents and dedents. In Python- and Haskell-like grammar expressions, there are cases, in which a piece of code should be parsed normally and not in an indentation-sensitive way. Therefore we define TrimWithIndents and TrimWithoutIndents as follows:

```
<e>TrimWithIndents  ≃  ( ▷ / ◁ / ↔ )*,
                        <e>,
                        ( ▷ / ◁ / ↔ )*

<e>TrimWithoutIndents  ≃ (! ▷ , ! ◁ , ( ↔ / ↩ ))*,
                          <e>,
                          (! ▷ , ! ◁ , ( ↔ / ↩ ))*
```

Listing 2.15: Trim With/Without Indents

In the next chapter, we will present and prove our ideas by implementing them in a Petit Parser framework, and discuss the results.

3 Implementation

In this chapter we, *first)* briefly introduce the Petit Parser framework which is used for our implementations, *second)* present the new implemented indentation-sensitive methods in Petit Parser, *third)* present the new implemented operators and expressions in Petit Parser and *fourth)* explain the updated functions in Petit Parser.

3.1 Petit Parser

Petit Parser[14] is a parsing framework implemented in Smalltalk[1] by Lukas Renggli[2]. It uses a combination of four existing parser methodologies: *a) Scannerless parsers*[15] *b) Parser combinators*[11] *c) Parsing expression grammars (PEGs)*[6] and *d) Packrat parsers*[5].

Writing grammars with Petit Parser is as simple as writing Smalltalk code. It provides a large set of ready-made parsers that can be composed to consume and transform arbitrary complex indentation-insensitive languages. Terminal parsers are the most simple ones (some terminal parsers used in this paper are shown in Table 3.1).

Terminal Parsers	**Description**
$'abc'$ asParser	Parses the string $'abc'$.
$\$a$ asParser	Parses the character a.
$\#word$ asParser	Parses a digit or a letter.
$\#digit$ asParser	Parses one digit (0..9).
$\#letter$ asParser	Parses a digit or a letter.

Table 3.1: Some of Petit Parser's pre-defined terminal parsers

Another set of parsers are used to combine parsers together, and with some others we do an action or transformation on a parser (some parser combinators and action parsers used in this paper are shown in Table 3.2 and Table 3.3).

For example, to define a context-free grammar that parses a simple nested if-statement, similar to the first example in Listing 2.7, some rules are defined as follows:

[1]`http://www.pharo-project.org`
[2]`http://www.lukas-renggli.ch/blog/petitparser-1`

Parser Combinators	**Description**
p1, p2	Sequence Parser (parses p1 followed by p2).
p1 / p2	Choice Parser (parses first p1, if it fails parses p2).
p star	Possessive Repeating Parser (parses zero or more p).
p plus	Possessive Repeating Parser (parses one or more p).
p optional	Optional Parser (parses p if possible).
p and	And Parser (parses p but does not consume its input).
p not	Not Parser (parses p and succeeds when p fails, but does not consume its input).
p end	EndOfInput Parser (parses p and succeeds only at the end of the input).

Table 3.2: Some of Petit Parser's pre-defined parser combinators

Action Parsers	**Description**
p flatten	Creates a string from the result of p.
p trim	Trims whitespaces before and after p.
p ==> aBlock	Performs the transformation given in aBlock.

Table 3.3: Some of Petit Parser's pre-defined action parsers

```
|statement expression identifier start|
identifier := DelegateParser new.
expression := DelegateParser new.
statement := DelegateParser new.

identifier setParser:
                ($- asParser optional)
                ,(#word asParser)star.

expression setParser:
             ((identifier trim)
              ,(('==' asParser)/('<=' asParser))trim
              ) optional
             ,
              (identifier trim).

statement setParser:
    ('System.out.println' asParser trim,
    $(asParser trim,expression trim,$)asParser trim,
    $; asParser trim)
    /
    (${asParser trim,statement trim star,$}asParser trim)
    /
    ('if' asParser trim,
    $( asParser trim, expression trim,$) asParser trim,
    statement trim, 'else' asParser trim,statement trim
    )flatten.

start := statement end.
start parse:
'if (x <= y) {
    if (x == y)
            System.out.println(y);
    else
            System.out.println(-x);
}
else
    System.out.println(x);'
```

Listing 3.1: Simple nested if-statement grammar in Petit Parser

line 2-4: We instantiate the rules as `DelegateParser`, because they recursively refer to each other. The method `setParser:` then resolves this recursion.

line 6-9: `Identifier` is defined as a SequenceParser, parsing

(1)'-' character followed by

(2) a word.

Since (1) is optional, therefore an identifier can also be a word string, which can occur many times or not at all.

line 10-16: `Expression` is defined as a SequenceParser, parsing

(1) an identifier followed by

(2)'==' or '<=' strings followed by

(3) an identifier.

Since (1) and (2) are grouped and optional, therefore an expression can also be an identifier.

line 17-27: `Statement` is defined as a ChoiceParser, parsing

(1) 'System.out.println (' followed by an expression then followed by ') ;'

or (2) ´{' character followed by a `statement` then followed by ´}' character

or (3) 'if (' followed by an expression and followed by ')' then followed by a `statement` followed by 'else' and again followed by a `statement`.

`Statement` is also recursive and calls `statement` again in its definition, which makes it possible to create, for example, nested if-statements. The whole group (3) is flattened, which creates a string of the parsing print result (see the example bellow).

```
"print result of our parsed Stream
        without a flatten action:"
 #('if' $( #(#(#(nil #($x)) '<=') #(nil #($y))) $
    ) #(${ #(#('if' $( #(#(#(nil #($x)) '==') #(
    nil #($y))) $) #('System.out.println' $( #(
    nil #(nil #($y))) $) $;) 'else' #('System.out
    .println' $( #(nil #($- #($x))) $) $;))) $})
    'else' #('System.out.println' $( #(nil #(nil
    #($x))) $) $;))
"print result of our parsed Stream
            with a flatten action:"
 'if (x <= y) {
    if (x == y)
            System.out.println(y);
    else
            System.out.println(-x);
}
else
    System.out.println(x);'
```

line 29-38: We set the end of the input stream to the end of our statement and define `start` as an EndOfInputParser. Then we call the `parse` method of our Parser (see Listing 3.2), which takes a stream as an argument for the first receiving parser and calls its `parseOn:` method (see Listing 3.3).

```
Parser>>parse: anObject
    ↑ self parseOn: anObject asPetitStream
```

Listing 3.2: parse method for Parser

```
DelegateParser>>parseOn: aStream
    ↑ parser parseOn: aStream
```

Listing 3.3: parseOn: method for DelegateParser

As mentioned in the example in Listing 3.1, parsers recognize or analyze their input by calling their defined `parseOn:` method, and if the expected input is not parsed, the parser returns a failure and stops parsing. For example, if we consider the `identifier` in Listing 3.1, which is defined as a SequenceParser, containing a list of other parsers (an OptionalParser and a PossessiveRepeatingParser). For parsing '-x', the OptionalParser has to call a LiteralObjectParser (for parsing the '-' character), and the PossessiveRepeatingParser a PredicateObjectParser (for parsing the 'x' character) and the results are returned and accepted by SequenceParser and this process goes on until the end of an input stream (see a list of parseOn: methods called by related parsers in Section A.3).

All indentation-insensitive context-free languages can be parsed by PEGs, but in order to have a proper indentation-sensitive parse result, we should implement our aforementioned ideas in subsection 2.3.2 to the Petit Parser framework and prove with an appropriate new defined indentation-sensitive grammar, if we are able to parse indentation-sensitive languages with our extended PEG. The new implemented operators and expressions are explained further in Section 3.3.

3.2 Indentation-sensitive Petit Parser

In order to parse through an indentation-sensitive input stream, we need a new stream with indentation- and context-sensitive features like having the column position, which will be discussed in the following sections. Therefore we define a new stream 'IndentStream' and use the standard parser's `parseOn:` methods by changing their stream (as parameter) with our indentation-sensitive input stream.

3.2.1 Indentation-sensitive stream

Before we start to implement new operators, we need to define some new features for our indentation-sensitive input stream 'IndentStream'. Some of the features used in this paper are shown in Section A.3.

If we consider column (see Section A.3), we update the list of positions with newlines while reading characters from a stream. The `column` is then computed as the difference between the current position and the closest new line position that is smaller than the current position.

We also extend the 'IndentStream' with another instance variable `indentStack` (we initialize it with zero) and add more methods like `pushIndent:` (to set the column to the specified value) and `popIndent` (to restore the column to the previous value).

3.3 New implemented operators and expressions in Petit Parser

Now that we have an indentation-sensitive stream, we can implement new required features for having a proper indentation-sensitive parsing result. The new-implemented parsers and expressions are mentioned in the following sections.

3.3.1 Indent Parser

'Indent' is happening, whenever the begin level of a current line is higher than the begin of the previous line. The higher is the begin level of the current line compared to the begin of the previous line, the more indents are signaled (see Section A.4.IndentParser).

3.3.2 Dedent Parser

'Dedent' is happening, whenever the begin level of a current line is lower than the begin level of the previous line (we have an indent before a dedent). The lower is the begin level of the current line compared to the begin of the previous line, the more dedents are signaled (see Section A.4.DedentParser).

3.3.3 Set and Remove Indent Parser

'Set indent' sets the current indentation level to the column of a current position in a stream and 'remove indent' removes the current indentation level the previous one is restored (see Section A.4.SetIndentParser and Section A.4.RemoveIndentParser).

We will use the stack to remember indentation values. It is used by `SetIndentParser` to set the indentation level to the 'given current value' and also by `RemoveIndentParser` to restore the indentation level to the 'previous value'.

3.3.4 Preserve Indent

The 'preserve indent' ensures that an expression starts on the same column as is the current indentation level (see Section A.4.PreserveIndent).

3.3.5 Aligned

We present three options for being aligned as follows:

Aligned

The 'aligned' ensures that all expressions from a sequence (or all occurrences of a repeating expression) will start at the same column as the first expression (or the first occurrence of an expression, see Section A.4.Aligned).

Aligned With

The 'align with' connects two expressions and the second expression starts at the same column as the first expression (see Section A.4.AlignedWith).

Aligned With Optional

The 'align with optional' is the same as 'align with'. The only difference is that the occurrence of the second expression is optional (see Section A.4.AlignedWithOptional).

3.3.6 Arbitrary Indentation

We present two options for arbitrary indentation as follows:

Arbitrary Indent Dedent

The 'arbitrary indent dedent' allows an expression to be placed on the same line or on a new line with the same, higher or lower indent level (see Section A.4.ArbitraryIndentDedent).

Arbitrary Indent

The 'arbitrary indent' allows an expression to be placed only on the same line or on a new line with the same or higher indent level (see Section A.4.ArbitraryIndent).

3.3.7 Trim Without Indents

The 'trim without indents' will trim all the whitespace characters including new lines before and after an expression, but it will stop at the moment an indent or dedent appears (see Section A.4.TrimWithoutIndents).

Trim With Indents

Another trimming option, which trims besides whitespaces and new lines also indents and dedents before and after an expression (see Section A.4.TrimWithIndents).

3.4 Updating Petit Parser functions

Some methods related to packrat parsing should be updated as follows:

3.4.1 Backtracking

We mentioned in subsection 2.2.1, that backtracking is intensively used by PEGs. Since in the standard PEG it is more considered for parsing indentation-insensitive grammars, therefore it has to be updated in our extended PEG, in order to have a proper backtracking in indentation-sensitive grammars. We had to introduce a `StreamMomento` class, which allows the input stream to be remembered and restored properly. In order to ensure immutability of the memento, the stack is always copied so that no other object can modify the value of the stack referenced by memento (see Section A.3.Backtracking and Section A.3.Remember and restore).

3.4.2 Memoizing

Memoization in the standard PEG remembers the result of the parsing for a particular position in a stream. Once the result is computed, it is not re-computed any more, but it is looked up in the memoization table. In general, the key of the memoization table is a stream-position tuple. In an indentation-sensitive Haskell-like grammar the same stream and the same position may have a different indent level (see Section 3.3.6), therefore we extend the key of the memoization table with the indentStack information (see Section A.3.Memoizing).

In the next chapter, we will use our implemented operators and expressions and give an example of a defined grammar for Python- and Haskell-like languages, and show that it is possible to parse now indentation-sensitive grammars with our extended PEG.

4 Validation

In this chapter we evaluate correctness and performance of our extended PEG by parsing layout-sensitive languages like *first)* Python and *second)* Haskell, *third)* declare the off-side rule with our extended PEG and *fourth)* summarize what we have explained in the previous sections.

4.1 Python

In the following examples, we demonstrate how a nested if-statement in Python can be properly parsed by our extended PEG:

```
|ifStatement statement suite expression identifier start|
identifier := ($- asParser optional )
             , (#word asParser)star.

expression :=
            ((identifier trim)
              , (('==' asParser)/('<=' asParser))trim
              ) optional
             ,
             (identifier trim).

statement := ('print' asParser trimWithoutIndents
             , (identifier asParser)optional)
             / ifStatement.

suite := (#indent asParser
        , statement plus
        , #dedent asParser).

ifStatement :=
    'if' asParser trim, expression trim
    , ':' asParser trimWithoutIndents
    , suite
    ,('else' asParser trim
    , ':' asParser trimWithoutIndents
    , suite) optional.
```

```
27
28 start := ifStatement trim star enableIndents.
```

Listing 4.1: Python: Nested if-statement in extended PEG

We will declare the defined grammar in Listing 4.1 while parsing through the given examples as follows:

```
1 start parse:
2 'if x <= y :
3     if x == y :
4         print y
5 else :
6     print -x
7 '
```

line 1: It starts parsing the input stream until the end of the `ifStatement`.

line 2: We parse the first `if` followed by an expression followed by ':', then we trim all whitespaces including a new line until an indent appears (see lines 20-23 in Listing 4.1).

line 3: We have an indent, since the third line starts at [column 1]. We have a `statement`, which can be again an `ifStatement`, starting with an `if` followed by an expression followed by ':', then we trim all whitespaces including a new line (see lines 12-18 in Listing 4.1).

line 4: We have another indent, since the fourth line starts at [column 2]. The first choice in `statement` is selected, which is 'print' followed by an `identifier`, we have then two dedents remaining from the third part of the `suite` sequence rule (see lines 12-18 in Listing 4.1).

line 5: The indent level is 0 and the fifth line starts at [column 0]. The first `else` occurring is not nested and belongs to the first `if`. It is followed by ':', then we trim all whitespaces including a new line (see lines 26-28 in Listing 4.1).

line 6: We have an indent, since the sixth line starts at [column 1]. The first choice in `statement` is selected, which is 'print' followed by an identifier. We have a remaining dedent from the third part of the `suite`. We are at the end of the `ifStatement`, which is also the end of our input stream.

```
1 start parse:
2 'if x <= y :
3     if x == y :
4         print y
5     else :
6         print x
7 '
```

lines 1-4: Are the same as in the previous example.

line 5: The indent level of the fourth line was 2, since the fifth line starts at [column 1], we have a decrease by one, which is a dedent. The first `else` occurring is nested and belongs to the nested second `if`. It is followed by ':', then we trim all whitespaces including a new line.

line 6: The indent level of the fifth line was 1, since we had a dedent. The sixth line starts at [column 2]. It takes the first choice of `statement`, which is ‘print’ followed by an identifier. We have two remaining dedents from the third part of the `suite`. We are at the end of the `ifStatement`, which is also the end of our input stream.

The “dangling ELSE” problem mentioned in Listing 2.7 (fourth example) is now solved in Python language by introducing layout-sensitive rules as shown in Listing 4.1.

4.2 Haskell

In the following examples, we demonstrate how an if-do-statement in Haskell can be properly parsed by our extended PEG:

```
1  |ifStatement statement statementBlock expression
       identifier start|
2  identifier := (#word asParser)star.
3  expression :=  ((identifier trim),($< asParser))trim
4                 ) optional
5                ,(identifier trim).
6  statement := ('print' asParser trimWithoutIndents
7               ,('"' asParser trim
8               ,(identifier asParser)trim
9               ,'"' asParser trim)optional.
10
11 statementBlock := statement plus aligned.
12 ifStatement :=
13     'if' asParser trim, expression trimWithoutIndents
14     ,'then' asParser trim ,'do' asParser trim
15     , (statementBlock indentedDedentedArbitrary).
16
17 start := ifStatement trim star enableIndents.
```

Listing 4.2: Haskell: if-do-statement in extended PEG

We will declare the defined grammar in Listing 4.2 while parsing through the given examples as follows:

```
1 start parse:
2 'if number < 0
3     then do print "You win!"
4             print "number is too low!"
5 '
```

line 1: It starts parsing the string until the end of the `ifStatement`.

line 2: We parse an `if` followed by an expression, then we trim all whitespaces including a new line (see line 13 in Listing 4.2).

line 3: We have a whitespace, since the third line starts at [column 1]. We parse ‘then’ followed by ‘do’ followed by a `statementBlock`. Before we start to parse the first `statement`, we set an indent to the current column position 9. We start parsing the first `statement` at the same line (see Section A.4.ArbitraryIndentDedent), which is ‘print ”You win!”’. Since the `statementBlock` should be aligned, therefore we trim all whitespaces including a new line with no indents and dedents occurring and preserve the indent level (see lines 11-15 in Listing 4.2).

line 4: The indent level is set to 9, therefore the `statement` in the fourth line should also start at [column 9]. The indent level is preserved while being in the `statementBlock`. The `statement` is finished in the fourth line, we remove the indent set at the beginning of the `statementBlock` (see line 11 in Listing 4.2).

line 5: The indent level of the previous line was 9. The indent level is now 0 and we are at the end of the `ifStatement`, which is also the end of our input stream.

```
"Example of an arbitrary statement block in the same line."
start parse:
'if number < 0
    then do print "You win!"
            print "number is too low!"'
"Example of an arbitrary statement block in a new line but same indent level as do."
start parse:
'if number < 0
    then do
         print "You win!"
         print "number is too low!"'
"Example of an arbitrary indented statement block in a new line."
start parse:
'if number < 0
    then do
            print "You win!"
            print "number is too low!"'
"Example of an arbitrary dedented statement block in a new line."
start parse:
'if number < 0
    then do
print "You win!"
print "number is too low!"'
```

As shown in the previous examples, the position of the first statement in Haskell-like grammars is arbitrary. Therefore we can have some valid do statements as above, with arbitrarily positioned statement blocks (see Section A.4.ArbitraryIndentDedent). Haskell has also another rule where all the expressions in a group should be aligned on the same indentation level as the first expression (see Section A.4.Aligned). As we see in the following example, the second statement 'print "number is too low!"' in line 6 is not aligned with the statement block, therefore it will be not considered as a part of the block.

```
1 "Statement in line 6 in not aligned with the statement block."
2 start parse:
3 'if number < 0
4     then do
5             print "You win!"
6         print "number is too low!"'
```

The parsing of a Haskell language and its mentioned rules is now solved by introducing indentation-sensitive rules as shown in Listing 4.2.

Haskell with Python-like if rule

It is also possible to combine Haskell-like blocks with Python nested if-statements. An example with the defined grammar is shown as follows:

```
 1 |ifStatement statement statementBlock expression elseIf
       identifier start|
 2 identifier := ($- asParser optional )
 3                  , (#word asParser)star.
 4 expression :=
 5              ((identifier trim)
 6                 , (('==' asParser)/('<=' asParser))trim
 7                 ) optional
 8                ,
 9                (identifier trim).
10 statement := ('print' asParser trimWithoutIndents
11               , (identifier asParser) optional)
12               / ifStatement.
13
14 statementBlock := statement plus aligned.
15
16 elseIf := 'else' asParser trim , ':' asParser trim
17         ,(statementBlock indentedDedentedArbitrary).
18
19 ifStatement :=
20     'if' asParser trim, expression trim
21     , ':' asParser trimWithoutIndents
22     ,((statementBlock indentedDedentedArbitrary)
23     )(alignWith: elseIf) optional.
24
25 start := ifStatement trim star enableIndents.
```

Listing 4.3: Haskell-like blocks with Python nested if-statement

In the following example, the `else` is aligned with the first `if` (see line 23 in Listing 4.3), and its statement block is arbitrarily indented in a new line (see line 17 in Listing 4.3).

```
start parse:
'if x <= y :
    if x == y :
        print 1
        print 2
else :
        print 3
        print 4'
```

In the following example, the `else` is aligned with the second `if` (see line 23 in Listing 4.3), and its statement block is arbitrarily dedented in a new line (see line 17 in Listing 4.3).

```
start parse:
'if x <= y :
    if x == y :
print 1
print 2
    else :
  print 3
  print 4'
```

4.3 Off-side rule

The off-side rule declared in Listing 2.6 is also definable with our extended PEG. An example with the defined grammar is shown as follows:

```
|expression rest restOffside|
expression := ((#digit asParser plus)
              , (rest star)
              , restOffside optional)
              /expression.

rest := $+ asParser, expression
       /$- asParser, expression.

restOffside := #indent
               , rest indentedArbitrary star
               , #dedent

start := expression trim star enableIndents.
```

Listing 4.4: PEG: Off-side rule

In the following example, the '-' in the third line is occurring after an indent and is arbitrarily indented in the same line. It follows the `restOffside` rule in Listing 4.4.

```
start parse:
'1+2
   -3+4
'
```

In the following example, the second line indent level is 1, but the '-' in the third line starts at [column 2] and is arbitrarily indented in the same line. It follows the `restOffside` rule in Listing 4.4.

```
start parse:
'    1+2
       -3+4
'
```

In the following example, the second and third line indent level are 1, therefore it fails while parsing and does not follow the `restOffside` rule in Listing 4.4.

```
start parse:
'    1+2
     -3+4
'
```

4.4 Performance

There are several issues regarding performance of layout-sensitive grammars in our implementation and layout-sensitive grammars in general. At first the input stream has to be extended to be column- and line-aware providing the `column` accessor. So far, we have not done any measurements to figure out how much time is spent during these operations and this is a subject of our further research.

We introduced our extension to a Petit Parser framework in Section 3.3. The implementation has probably a negative impact on the performance of the layout-sensitive grammar expressions. We would like to investigate the impact of these expressions and we would like to improve the performance (if necessary) in the future.

Last but not least, there is an unknown performance impact of a memoization function, which uses more complex key (because an indentation is included in a key). This increases number of entries and lookup-time in a memoization table, but the real impact on real grammars is still unknown.

All the previous performance issues are related only to the grammars with the layout-sensitive expressions. Our extension of PEGs does not affect original parts of the parsing framework. Even the input stream can be converted into the column-aware stream on demand (during the first occurrence of layout-sensitive rule), as a result the non-layout-sensitive grammar does not have to use the slower column-aware input stream.

5 Related Work

Layout-sensitive Generalized Parsing

Erdweg *et al.* [4] proposed a parsing framework, in which a user can annotate layout in a grammar in order to support declarative specifications of grammars. They have integrated layout constraints into SDF[7][16] and implemented a layout-sensitive generalized parser as an extension of scannerless generalized LR parsing (SGLR)[15]. According to their validation on open-source Haskell files, the layout-sensitive generalized parsing is easy to use and its performance overhead is small compared to layout-insensitive parsing (approximately two times slower) for practical application.

They use special tokens selectors (as *first* or *last* for a first and last line, *left* or *right* for a leftmost or rightmost token), and position selectors (as *line* and *col* for line and column) to define the shape of a subtree (result of the parsing process). For example, one can specify constrains corresponding to the off-side rule like this: `first.col <= left.col`. For example, the $(1 + 2) - (3 * 4)$ expression can be split into the multiple-line expression like this:

```
(1+2)
    - (3x4)
```

Erdweg *et al.* modified a standard SGLR parser. The generalized LR parser processes all the possible interpretations of the input stream in parallel, returning all the possible results. In the next phase, all the results that do not correspond to the layout constrains are filtered out.

The main difference between our layout-sensitive approach and this work is that Erdweg *et al.* use token and position selectors with relation operators to define shape of a subtree, while with our approach a user can define the shape of delimiters between parsers. In other words, with the approach taken by Erdweg *et al.*, one restricts the shape of a parsed input in a root rule, while in our work the root rule can restrict only the delimiters of direct children. The other difference is that Erdweg *et al.* use generalized LRs parser for context-free grammars and we use top-down LL parsers specified by parsing expression grammars. Erdweg *et al.* approach filters out results that do not correspond to the layout after the parsing (thanks to the fact that GLR returns all the results for ambiguous grammars) while in the case of PEGs, the decision has to be taken during the parsing (since PEGs are unambiguous).

Principled Parsing for Indentation-Sensitive Languages

Adams [1] presents a simple extension to the context-free grammars (CFG) — indentation sensitive context-free grammars (IS-CFG) — that can express layout-sensitive rules. In the IS-CFGs terminals and non-terminals are annotated with the column (Adams calls this indentation). The grammar specifies a numerical relation where the indentation of each non-terminal on the left-hand side (LHS) must correspond with the indentation of its immediate children on the right-hand side (RHS). For example, the grammar where all the brackets should be indented to the same level can be expressed like: $A \rightarrow \text{'('}^{=} A^{>} \text{')'}^{=}$.

Adams provides a description how to develop an GLR and LR(k) algorithm for the IS-CFG claiming that CYK, SLR, LALR, GLL can be constructed as well. His experiments on Haskell shows that the indentation-sensitive parser generated from the IS-CFG runs approximately three times slower than a parser using traditional *ad hoc* techniques for handling indentation sensitivity.

The difference between our work and Adams, is that Adams specifies the relation between column of a LHS non-terminal and column of RHS terminals and non-terminals while we specify the shape of a spacing between the terminals and non-terminals on a RHS. There is no direct relation between LHS and the RHS in our approach.

Of course, Adams work is focused on parser generators (LR, LL, etc) while we focus on parsing expression grammars that are parser recognizers. Our approach and Adam's are similar in a sense that the LHS has impact only on the immediate children, opposed to Erdweg *et al.* [4] where the LHS can have impact on any children in a subtree.

Indentation Sensitive Languages

Brunauer and Mühlbacher [3] suggested an indentation-sensitive related extension to context-free grammars (CFG). Their approach is based on counters. In their extended BNF notation, one can specify the indentation relation between the right- and left-hand side of a BNF rule using the counter $C^{n}_{\rightarrow}$ where the upper index (n) is a number of repetitions and the lower index ($\rightarrow$) is a character that is supposed to be repeated (the $\rightarrow$ states for tabulator). For example, indented statements in the `if` statement can be expressed as follows:

$$C^{n}_{\rightarrow} < If > \rightarrow\ C^{n}_{\rightarrow} \text{"}if\text{"} < Cond > \text{" : "} newline\ C^{n+1}_{\rightarrow} < Stmt >$$

They also suggested a method to construct scannerless, top-down parser passing the counters between the production rules.

The difference between Brunauer and Mühlbacher and our work is that our approach works with the concept of columns and not counters. Brunauer and Mühlbacher's approach cannot align a token to the column of a token in the middle of a previous line because with counters you cannot track the column. This is important for Haskell-like rules. Another distinction is that they use counters to specify an indentation level, whereas we focus more on delimiters between the parsers. On the other hand, our indent and dedent tokens use counters internally in the implementation to consume appropriate number of whitespace characters at the beginning of the line.

Monadic Parser Combinators

Hutton [10] and Hutton and Meijer [11] describes how to extend a parser monad to handle the off-side rule. They changed the parser monad to include positional information. They modified *whitespace* parser combinator, which accepts only tokens that are "on-side" – tokens that begin on a column equal or higher than actual indentation level. Using the updated whitespace combinator, they defined a new parser combinator that fails if the underlaying parser is in “off-side” position and returns the underlying parser result otherwise.

6
Conclusion

In this paper we introduced four primitive terminal parsing expressions for PEGs — indent $\triangleright$, dedent $\triangleleft$, set indent $\trianglerighteq$ and remove indent $\trianglelefteq$. We used these primitive parsers to define more complex parsers suitable for defining layout-sensitive grammars. We also demonstrated and validated expressiveness of our extensions on Python- and Haskell-like grammars. We further implemented our ideas in the Petit Parser framework by defining Python- and Haskell-like grammars.

6.1 Future work

In the future, we would like to revise the layout-sensitive expressions of PEGs. Currently, the indent (or dedent) operations are defined as new line and increased (or decreased) indentation. These semantics were inspired by Python. We would like to investigate the possibility of expressing layout-sensitive grammars with other, even simpler operators and expressions. Furthermore, we would like to use our extensions to define full indentation-sensitive grammar to better understand the drawbacks of the current formalism and suggest a better one. Last but not least, we would like to investigate performance in more detail and suggest more efficient implementations.

Bibliography

[1] M. D. Adams. Principled parsing for indentation-sensitive languages: Revisiting Landin's offside rule. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, page 511–522, New York, NY, USA, 2013. ACM. ISBN 978–1–4503–1832–7. doi: 10.1145/2429069.2429129.

[2] J. Berstel and L. Boasson. Context-free languages. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, chapter Context-free languages, pages 59–102. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-444-88074-7. URL `http://dl.acm.org/citation.cfm?id=114891.114893`.

[3] L. Brunauer and B. Mühlbacher. Indentation sensitive languages. http://www.cs.uni-salzburg.at/ ck/content/classes/TCS-Summer-2006/index.html, 2006. URL `http://www.cs.uni-salzburg.at/~ck/content/classes/TCS-Summer-2006/index.html`.

[4] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-sensitive generalized parsing. In *SLE*, pages 244–263, 2012. ISBN 978-3-642-36088-6. doi: 10.1007/978-3-642-36089-3_14. URL `http://www.informatik.uni-marburg.de/~seba/projects/sugarj/layout-parsing.pdf`.

[5] 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`.

[6] 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`.

[7] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf – reference manual–. *SIGPLAN Not.*, 24(11):43–75, Nov. 1989. ISSN 0362-1340. doi: 10.1145/71605.71607. URL `http://doi.acm.org/10.1145/71605.71607`.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, Mar. 2001. ISSN 0163-5700. doi: 10.1145/568438.568455. URL `http://doi.acm.org/10.1145/568438.568455`.

[9] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language haskell — A non-strict, purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992. URL `http://www.haskell.org/`.

[10] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[11] 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`.

[12] P. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, Mar. 1966. ISSN 0001-0782. doi: 10.1145/365230.365257. URL `http://www.cs.utah.edu/~eeide/compilers/old/papers/p157-landin.pdf`.

[13] Python. Python. http://www.python.org.

[14] 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)*, Malaga, Spain, June 2010. URL `http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf`.

[15] 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`.

[16] E. Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, jul 1997. URL `http://www.wins.uva.nl/pub/programming-research/reports/1997/P9706.ps.gz`.

List of Tables

Listings

Appendices

A.1 Abbreviations

A list of most used abbreviations in this paper.

CFG stands for context-free grammar.

CFL stands for context-free language.

CSG stands for context-sensitive grammar.

CSL stands for context-sensitive language.

PEG stands for parsing expression grammar.

A.2 Operators for constructing parsing expressions

We use these operators for defining parsing expressions. The meaning of operators is briefly described in the following text.

indent $\rhd$ is an expression recognizing an indent.

dedent $\lhd$ is an expression recognizing a dedent.

Set Indent $\trianglerighteq$ is an expression setting an indent level.

Remove Indent $\trianglelefteq$ is an expression removing an indent level.

repetition $e*$ is an expression recognizing with zero or more occurrences of e.

repetition $e+$ is an expression recognizing one or more occurrences of e.

optionality $e?$ is an expression recognizing optional occurrence of e.

negation $!e$ is not-predicate — an expression e cannot be recognized to recognize $!e$.

and $\&e$ is and-predicate — $\&e$ attempts to match an expression e.

choice e_1/e_2 is a prioritized choice between e_1 and e_2.

sequence e_1, e_2 is a sequence expression, the e_1 is followed by e_2.

composite expression e will be enclosed in $<$ and $>$. We will use this syntax to emphasize the fact that a complex expression might be hidden behind e and to distinguish from other textual operators.

literal string will be enclosed with $'$ or $''$. To create expression recognizing *EXP-NAME* we use the expression: `'EXP-NAME'`.

grouping e will be enclosed with (and).

character class e will be enclosed with [and].

whitespace character $\leftrightarrow$ is an expression that recognizes a space or a tabulator.

new line $\hookleftarrow$ is an expression that recognizes a new line character.

space ␣ is an expression that accepts only space.

tabulator $\rightarrow$ is an expression that accepts only tabulator.

any character $\cdot$ is an expression that recognizes any character.

assignment $P \leftarrow e_1/e_2$ defines the expression a that recognizes e_1 or e_2. Right side of the rule can be arbitrary complex expression using the PEG operators.

equivalence $e_1 \simeq e_2$ is an expression that states that e_1 is equivalent to e_2. Two parsing expressions are equivalent if they can recognize the same set of input strings even though they have different definition.

A.3 The definition of some Petit Parser methods

The definition of parseOn: methods called by some parsers.

SequenceParser

```
SequenceParser>>parseOn: aStream
    | start elements element |
    start := aStream position.
    elements := Array new: parsers size.
    1 to: parsers size do: [ :index |
        element := (parsers at: index)
            parseOn: aStream.
        element isPetitFailure ifTrue: [
            aStream position: start.
            ↑ element ].
        elements at: index put: element ].
    ↑ elements
```

OptionalParser

```
OptionalParser>>parseOn: aStream
    | element |
    element := parser parseOn: aStream.
    ↑ element isPetitFailure ifFalse: [ element ]
```

LiteralObjectParser

```
LiteralObjectParser>>parseOn: aStream
    ↑ (aStream atEnd not and:
        [ literal = aStream uncheckedPeek ])
        ifFalse: [ Failure message: message
                             at: aStream position ]
        ifTrue: [ aStream next ]
```

PossessiveRepeatingParser

```
PossessiveRepeatingParser>>parseOn: aStream
    | start element elements |
    start := aStream position.
    elements := OrderedCollection new.
    [ elements size < min ] whileTrue: [
        (element := parser parseOn: aStream)
    isPetitFailure ifTrue: [
                aStream position: start.
                ↑ element ].
        elements addLast: element ].
    [ elements size < max ] whileTrue: [
        (element := parser parseOn: aStream)
    isPetitFailure
                ifTrue: [ ↑ elements asArray ].
        elements addLast: element ].
    ↑ elements asArray
```

PredicatePbjectParser

```
PredicateObjectParser>>parseOn: aStream
    ↑ (aStream atEnd not and:
        [ predicate value: aStream uncheckedPeek ])
        ifFalse: [ Failure message: predicateMessage
                                at: aStream position ]
        ifTrue: [ aStream next ]
```

The definition of some indentation-sensitive methods.

Column

```
IndentStream>>column
    |column|
"The first position in a line is initialized with 0"
    column := 0.
"It moves through the line, until the given position is
    found and returns the column value"
    (1 to: position) do:
    [:index |
        column := column + 1.
        ((collection at: index) == Character cr
        or: [ (collection at: index) == Character lf ])
        ifTrue: [ column := 0 ]
    ].
    ↑ column
```

PushIndent

```
IndentStream>>pushIndent: value
"Pushes the value given as a parameter to the Stack"
    indentStack push: value.
    ↑ indentStack

IndentStream>>pushIndent
"Increase the stack level by one"
    ↑ self pushIndent: (indentStack top + 1)
```

PopIndent

```
IndentStream>>popIndent
"Decrease the stack level by one"
    indentStack pop.
    ↑ indentStack
```

Initialize

```
"Stack and indent level are initialized with 0."
IndentStream>>initialize
    indentStack := Stack new push: 0; yourself.
```

Backtracking

```
"How the string is remembered in standard PEG (with
    position)"
StringParser>>parseOn:aStream
    | position size |
    position := aStream position.
    size := string size.
    (aStream next: size) = string ifTrue: [
        ↑ string
    ].
    aStream position: position.
    ↑ Failure new

"How the string is remembered and restored in our extended
     PEG (with position, IndentStack)"
StringParser>>parseOn:aStream
    | memento size |
    memento := aStream remember.
    size := string size.
    (aStream next: size) = string ifTrue: [
        ↑ string
    ].
    aStream restore: memento.
    ↑ Failure new
```

Remember and restore

```
IndentStream>>remember
    ↑ StreamMomento
        position: position
    stack: indentStack

IndentStream>>restore: aStreamMemento
    position: aStreamMemento position.
    indentStack := aStreamMemento indentStack.
```

Memoizing

```
"remember method of Memoizing Parser in standard PEG "
MemoizingParser>>remember:result in:aStream
    key := Tuple with: aStream
                 with: aStream position
    memoizationTable at: key put: result.

"remember method of Memoizing Parser in our extended PEG "
MemoizingParser>>remember:result in:aStream
    key := Triple with: aStream
                  with: aStream position
                  with: aStream indentStack.
    memoizationTable at: key put: result.
```

A.4 New implemented operators and expressions in Petit Parser

IndentParser

```
IndentParser >> parseOn: aStream
| memento position lastIndentLevel |
memento := aStream remember.

aStream indentStack isEmpty
    ifTrue: [ ↑ self fail: memento stream: aStream ].

lastIndentLevel := aStream indentStack top.

(aStream peek == Character cr)
    ifTrue: [ aStream next. ].

(aStream column > lastIndentLevel)
    ifTrue: [ ↑ self fail: memento stream: aStream ].

[((aStream column < lastIndentLevel)
and:
[aStream atEnd not])]
    whileTrue:
        [
        (self isWhitespaceCharacter: aStream next)
            ifFalse:
                [
                position := aStream position.
                aStream restore: memento.
                ↑ Failure message: 'Indent expected'
                          at: position.
                ]
        ].

(self isWhitespaceCharacter: aStream next)
    ifTrue:
        [
        aStream pushIndent.
        ↑ #indent
36          ].
37
38 ↑ self fail: memento stream: aStream
```

line 3: Memoize the stream features, like `position`, `indentStack`. This will be discussed later in subsection 3.4.2.

line 5-9: If the `indentStack` is empty, it fails, otherwise it initializes the begin of the previous line '`lastIndentLevel`' with the top value of the `indentStack`.

line 10-15: If the current parsed character is a new line it continues, and if the begin of the current line is greater than the `lastIndentLevel` it fails.

line 16-30: While the begin of the current line is smaller than the `lastIndentLevel` and it is not the end of our input stream, we look for a tab or space character to have an indent and if it fails we expect an indent.

line 31-38: If we already had an indent, we look for another indent (a tab or space), if it occurs, we increase the stack level by one and return an indent. We fail if none of these cases happens.

DedentParser

```
DedentParser >> parseOn: aStream
| memento lastIndentLevel |
memento := aStream remember.

aStream indentStack isEmpty
    ifTrue: [ ↑ self fail: memento stream: aStream].

lastIndentLevel := aStream indentStack top.

(aStream peek == Character cr)
    ifTrue: [ aStream next. ].

(aStream column >= lastIndentLevel)
    ifTrue: [ ↑ self fail: memento stream: aStream ].

[(aStream column < (lastIndentLevel - 1))
and:
[(aStream atEnd not)
and:
[self isWhitespaceCharacter: aStream peek ] ] ]
     whileTrue: [
                aStream next.
                ].

(self isWhitespaceCharacter: aStream peek) not
    ifTrue:
        [
        aStream popIndent.
        ↑ #dedent
        ].

↑ self fail: memento stream: aStream
```

line 3-12: Same as `IndentParser`.

line 13-15: If the begin of the current line is greater or equal to the `lastIndentLevel` it fails.

line 16-24: While the begin of the of the current line is smaller than the `lastIndentLevel` minus one, it is not the end of our input stream, and the current parsed character is a tab or a space, it continues.

line 25-32: If we already had an indent or a dedent, we look for another dedent (‘no’ tab or space), if it occurs, we decrease the stack level by one and return a dedent. We fail if none of these cases happens.

SetIndentParser

```
SetIndentParser >> parseOn: aStream
    aStream pushIndent: aStream column.
    ↑ nil
```

RemoveIndentParser

```
RemoveIndentParser >> parseOn: aStream
"If indentStack is empty, it fails."
    aStream indentStack isEmpty ifTrue: [
        ↑ Failure message: 'Nothing to left to be popped'
    at: aStream position
    ].

    aStream popIndent.
    ↑ nil
```

PreserveIndent

```
1 Parser >>preserveIndent
2     ↑  #blank asParser star,
3        #indent asParser not,
4        #dedent asParser not,
5       (#blank asParser star,
6        #newline asParser,
7        #blank asParser star) star
8        ,self ==> [:tokens | tokens fifth ].
```

line 2-4: `Blank`, which is our defined whitespace (a tab or a space) can occur many times or not at all, followed by no indent and dedent.

line 5-7: A repeating `blank` followed by a new line followed by another repeating `blank` are as a group and can occur many times or not at all.

line 8: The expression, which is the fifth element ‘self’ will be returned by the method (a group ‘(and)’ counts as one element, therefore the ‘self’ is getting the fifth element in the whole sequence).

Aligned

```
1 PossessiveRepeatingParser>> aligned
2     ↑ SetIndentParser new,
3      (self copy setParser: parser preserveIndent),
4       RemoveIndentParser new
5         ==> #second
```

line 2: We set an indent.

line 3: It ensures that all expressions from a sequence will start at the same column (see subsection 3.3.4).

line 4: We set a dedent.

line 5: The second element, which is the grouped sequence in line 3 will be returned.

AlignedWith

```
1 Parser>> alignWith: anotherParser
2     ↑ SetIndentParser new,
3      (self, anotherParser preserveIndent),
4       RemoveIndentParser new
5         ==> #second
```

line 2: We set an indent.

line 3: If we consider ‘self’ as $< e_1 >$ and ‘anotherParser’ as $< e_2 >$, this grouped sequence succeeds when $< e_2 >$ starts at the same column as $< e_1 >$ (see subsection 3.3.4).

line 4: We set a dedent.

line 5: The second element, which is the grouped sequence in line 3 will be returned.

AlignedWithOptional

```
1 Parser>> alignWithOptional: anotherParser
2     ↑ SetIndentParser new,
3       (self, (anotherParser preserveIndent optional)),
4       RemoveIndentParser new
5         ==> #second
```

line 3: The other lines are the same as `alignwith:`, but the grouped sequence is optional, which can occur or not at all.

ArbitraryIndentDedent

```
1  Parser>> indentedDedentedArbitrary
2      | indent dedent |
3      indent := UnresolvedParser new.
4      dedent := UnresolvedParser new.
5
6      indent def: #indent asParser,
7                  (self / indent),
8                  #dedent asParser.
9
10     dedent def: #dedent asParser,
11                 (self / dedent),
12                 #indent asParser.
13
14     ↑ self / indent / dedent
```

line 6-8: It allows the expression block to be placed on the same line or a new line with the same or higher indent level (it makes sure that all indents will be consumed by appropriate number of dedents).

line 10-12: It allows the expression block to be placed on the same line or a new line with the same or lower indent level (it makes sure that all dedents will be consumed by appropriate number of indents).

ArbitraryIndent

```
1  Parser>> indentedArbitrary
2      | indent |
3      indent := UnresolvedParser new.
4
5      indent def: #indent asParser,
6                  (self / indent),
7                  #dedent asParser.
8      ↑ self / indent
```

line 5-8: It allows the expression block to be placed on the same line or a new line with the same or higher indent level (it makes sure that all indents will be consumed by appropriate number of dedents).

TrimWithoutIndents

```
1  Parser>> trimWithoutIndents
2      ↑ (#indent asParser not,
3         #dedent asParser not,
4         #space asParser) star,
5         self,
6        (#indent asParser not,
7         #dedent asParser not,
8         #space asParser) star
9          ==>  #second
```

line 2-4: It is a repeating sequence group, which starts with no indent followed by no dedent and ended with a space.

line 5: It is the expression itself and will be returned from this method as shown in line 9.

line 6-8: It is the same as in the first 3 lines.

TrimWithIndents

```
1 Parser>> trimWithIndents
2     ↑ (#indent asParser
3        / #dedent asParser
4        / #space asParser) star,
5        self,
6        (#indent asParser
7        / #dedent asParser
8        / #space asParser) star
9          ==>  #second
```

line 2-4: The only difference to the previous implementation is that, instead of a sequence group, we have a choice group, which parses (1) an indent or (2) a dedent or (3) a space. This group can be repeated zero or more times.