Indentation-Sensitive Parsers for Free-Form Languages

Sérgio Queiroz de Medeiros
Escola de Ciência e Tecnologia, UFRN
Natal, Brazil
sergio.medeiros@ufrn.br

Hugo Musso Gualandi
Instituto de Computação, UFRJ
Rio de Janeiro, Brazil
hugomg@ic.ufrj.br

ABSTRACT

Consistent source code indentation is crucial for code readability, even in free-form languages that are oblivious to whitespace. However, as in a free-form language programmers can format the code as they desire, this may lead to mismatched indentation styles. Automatic code formatters address this problem by rewriting code to a standard layout, however they impose a specific style and only work for syntactically valid code.

We describe an extension of Parsing Expression Grammars (PEGs) that can model indentation information and we show that it can be used to specify different indentation styles. A parser based on such extension can check for indentation inconsistencies and inform the developer about them.

To evaluate our approach, we also implemented a full parser for the Lua programming language and used it to parse a well-known Lua codebase. We observed that 98% of source code lines were well-indented according to our specification.

KEYWORDS: Parsing, Indentation, Code Formatting, Syntatic Error Reporting

1 Introduction

Adequate source code indentation is essential for readability and maintainability [7], even for free-form programming languages which ignore whitespace. However, freedom of formatting may lead to clashing layout styles. To address this, code formatters may be employed to automatically rewrite the indentation of the programs. However, the programmer might not like the style enforced by the formatter [10, 11] or may not want to rewrite the source file entirely. Moreover, such formatters usually cannot handle code that is partially written or containing syntactic errors.

We argue that it is possible to develop indentation verifiers for free-form languages by employing parsing techniques originally developed to parse indentation-sensitive languages, such as Python and Haskell. Following the work of Adams and Ağacan [1, 2], we present an extension of Parsing Expression Grammars (PEGs) and use it to specify different indentation styles. The main technical novelty is that when our PEGs parse a line with suspicious indentation, they emit a warning instead of a syntax error.

Because we specify the indentation declaratively, as part of the grammar, our technique is not restricted to a single programming language and can also be adapted to match the indentation conventions of a given project. This contrasts with code formatters, whose ad-hoc algorithms support limited customization.

To evaluate our work, we implemented an indentation-sensitive parser for the Lua programming language and we used it to verify the indentation style of a well-known Lua codebase. By adapting the grammar specification to match the prevailing indentation style we could reach a point where 98% of source-code lines were considered to be well-indented.

The rest of this paper is organized as follows: Section 2 reviews PEGs and exemplifies the new indentation operators. Section 3 describes the semantics of indentation-aware PEGs. Section 4 evaluates the PEG grammars against real Lua code. Section 5 discusses related works and Section 6 presents our conclusions.

2 Describing Indentation Styles with PEGs

In this section we discuss different indentation styles and show how they could be described by a PEG extended with indentation operators.

Briefly, a standard PEG is similar to a Context-Free Grammar (CFG). A PEG consists of a set of rules of the form $A \leftarrow p$, where A is a variable and p is a parsing expression. Differently from CFGs, in PEGs there is only one rule associated with a given variable. Moreover, in PEGs the right-hand side of a rule is a parsing expression and there is a choice operator ($/$), which matches the alternatives in left-to-right order. Section 3 presents the semantics of PEGs in detail.

The abstract syntax of parsing expressions is given below, where a is a terminal, A is a variable, and p is a parsing expression:

$$p = \varepsilon \mid a \mid A \mid p_1 p_2 \mid p_1 / p_2 \mid p* \mid !p \tag{1}$$

Informally, ε successfully matches without consuming any input. The terminal a matches and consumes itself or fails otherwise. A variable A tries to match the grammar rule associated with that variable. The sequence $p_1 p_2$ tries to match p_1 followed by p_2. The ordered choice p_1 / p_2 tries to match p_1 and if that fails then it tries to match p_2. The repetition $p*$ matches p zero or more times, as many times as possible. The negative lookahead $!p$ asserts that the input does not match p; it succeeds without consuming input when the input does not match p and it fails when it does match.

To model indentation-sentitive languages, Adams and Ağacan [2] associate an indentation level with each node of the parse tree and they enrich the input tokens with column

```
if x {
   //commands
}
```

(a) Opening brace (usually at the same line) at a greater indentation level than the **if** keyword and closing brace at the same indentation level of **if**.

```
if x
{
   //commands
}
```

(b) Opening brace (usually at the next line) and closing brace at the same indentation level of the **if** keyword.

```
if x { //commands }
```

(c) Opening and closing braces with an indentation greater or equal than the **if** keyword. (Here both are also on the same line.)

Listing 1. Different layout styles for the opening and closing brace of a block.

numbers. To specify the indentation restrictions, they also introduced two new form of parsing expressions: $p^{\triangleright}$ and $|p|$.

In $p^{\triangleright}$, the subexpression p is superscripted by some relation $\triangleright$, which indicates how p should be indented regarding its parent node in the parsing tree. For example, we can write $p^{>}$ to say that a tree parsed by p must have a greater indentation than its parent. When not specified, the indentation of a child should be the same of its parent. Although in principle any indentation relation may be used, the most important ones are $=$, $>$, $\geq$, and $\circledast$. The first three have their usual meanings and $\circledast$ denotes the relation $\{(i, j) \mid i, j \in \mathbb{N}\}$, which allows any indentation level, without restriction.

The absolute alignment $|p|$ aligns code vertically. It makes without effect the next usages of the indentation operator $p^{\triangleright}$ until the next token is matched. In practice, this forces the first token of a "line" to be indented with the default $=$, even if the grammar would try to indent it with $>$ or $\geq$.

Let us now discuss how to use these indentation operators to specify indentation layouts in free-form languages. Consider Listing 1, which shows different ways to indent the opening and the closing brace of an if-statement. To allow the indentation style of Listing 1a, we can use the following grammar:

$$\mathit{if\_stmt} \leftarrow \texttt{if}^{=}\ \mathit{exp}^{>}\ \{^{>}\ \mathit{block}^{>}\ \}^{=} \tag{2}$$

This says that the `if` keyword must be at the same indentation level as the parent *if_stmt* node. The condition expression, the opening brace, and the inner block must have a column number greater than the the parent node. Thus, if they appear in another line, they must be indented to the right of the `if` keyword. Conversely, the closing brace must be placed at the same column as the `if`, necessarily in another line.

Note that, because the model only cares about column numbers, the opening brace is not required to be in the same line as the `if` keyword. Furthermore, the expression, opening brace, and block can each be indented at different levels.

To favor an indentation style where the opening brace aligns vertically with the `if` keyword, as shown in Listing 1b, we should use $\{^{=}$ instead of $\{^{>}$ in the grammar rule, as shown below:

$$\mathit{if\_stmt} \leftarrow \texttt{if}^{=}\ \mathit{exp}^{>}\ \{^{=}\ \mathit{block}^{>}\ \}^{=} \tag{3}$$

To deal with a single-line statement as in Listing 1c, one alternative is to weaken the restrictions in the grammar and use $\}^{\geq}$ instead of $\}^{=}$ for the closing brace.

$$\mathit{if\_stmt} \leftarrow \texttt{if}^{=}\ \mathit{exp}^{>}\ \{^{\geq}\ \mathit{block}^{>}\ \}^{\geq} \tag{4}$$

Another option is to merely ignore the indentation warning given by grammars 2 or 3. Because our parser does not reject mis-indented programs outright, the programmer can choose to sparingly deviate from the prescribed indentation rules.

As we mentioned before, the indentation relation does not impose restrictions with regard to the line of a token/terminal. So, considering this last specification of the *if_stmt* rule, Listing 2 respects all indentation restrictions.

```
if x
  {
 print("hello")
    }
```

Listing 2. Because the specification is indifferent to line numbers, it allows some unusual indentation. Here, the opening brace is not in the same line of the keyword **if**, the closing brace is at a greater indentation level than the **if** and and is not aligned with the opening brace.

Perhaps, if the grammar specification was allowed to inspect line numbers, it would be possible to develop a grammar that does not allow the strange code from Listing 2. However, in this paper we want to investigate how far we can get with Adams' technique [2], which does not care about line numbers.

It is not difficult to parse code without emitting indentation warnings. A trivial way to achieve this is by using the $\circledast$ operator. An expression like $\}^{\circledast}$, for example, allows a closing brace at any column. However, by doing this we are imposing no indentation restrictions. As we discuss in Section 4, the indentation rules of the PEG grammar will

depend on the code base and the intended coding style, leading to a parser that performs many or just a few indentation checks.

Unlike code formatter tools, which impose a very specific layout for a program, the approach followed by our and Adams's work describes a class of acceptable layouts for the program. Nevertheless, the formalism is powerful enough to model the layout restrictions imposed by indentation-sensitive languages such as Python and Haskell [1, 2].

For example, in Figure 1 we have a grammar that uses the previous definition of *if_stmt* and adds rules *block*, *stmt* and *assign_stmt*. Such grammar defines a language where the statements of a *block* should be aligned at the same indentation level and such level should be greater than the one of its surrounding if-statement.

$$
\begin{aligned}
block &\leftarrow (stmt^{=})^{*} \\
stmt &\leftarrow if\_stmt^{=} \;/\; assign\_stmt^{=} \;/\; \cdots \\
if\_stmt &\leftarrow \texttt{if}^{=}\ exp^{>}\ \{^{\geq}\ block^{>}\ \}^{\geq} \\
assign\_stmt &\leftarrow ident^{=}\ =^{>}\ exp^{>}
\end{aligned}
$$

Figure 1. Example of a PEG to match a block of statements at the same indentation level.

In rule *if_stmt*, the expression $block^{>}$ indicates that a *block* should be more indented, where a *block* consists of zero or more statements at the same indentation level ($(stmt^{=})^{*}$).

To the previous grammar work as expected, the first lexical element of a statement must also match the indentation level of its parent node, but this may require some effort. For example, let us consider the following definition of *ident*, where $[a-z]^{+}$ means an expression that matches any lowercase letter one or more times:

$$ident \leftarrow ([a\text{–}z]^{\geq})^{+} \tag{5}$$

The indentation relation ≥ in this definition of *ident* allows all letters of the identifier to appear at a greater indentation level than the statement they are contained in. Because of this, the identifier can appear anywhere in the line and it is possible to end up with misaligned assignment statements, as shown in Listing 3.

```
if x {
   a = 42
 b = 41
    c = 43
}
```

Listing 3. Misalignment caused by ≥ in the *ident* rule.

It is easy to fix this issue by using the absolute alignment operator ||. If we absolutely align the $|stmt|$, we discard the effect of relation ≥ when matching the first letter of an identifier, anchoring it to the indentation of the assignment statement. As a result, the parser now correctly specifies that statements in the block should be vertically aligned.

$$block \leftarrow (|stmt|^{=})^{*} \tag{6}$$

Of course, we could also achieve a similar result by manually changing the definition of *ident* to use = for the first character. However, this manual process can be more arduous when there are many layers of recursion in the affected grammar rules.

$$ident \leftarrow [a\text{–}z]^{=}\ ([a\text{–}z]^{\geq})^{*} \tag{7}$$

3 Semantics of PEGs with Indentation Information

In this section we present a formal semantics of PEGs with indentation restrictions and discuss its differences in comparison with the previous work of Adams [1, 2].

A parsing expression grammar G is a tuple (V, T, P, p_S), where V is a finite set of variables, T is a finite set of terminals, P is a total function from variables to *parsing expressions*, and p_S is the initial parsing expression.

We describe the function P as a set of rules of the form $A \leftarrow p$, where $A \in V$ and p is a parsing expression. A parsing expression, when applied to an input string, either consumes a prefix of it or fails. The notation $G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ means that expression p matches the input xy, consuming the prefix x, while resolving any variables using the rules of G. We use $G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ to express an unsuccessful match.

In the discussion below we assume that G is complete, that is, the matching of p_S always finishes, either consuming an input prefix or failing.

3.1 Extending PEGs to Check Indentation

In order to describe indentation restrictions, we extend the previous definition of PEG grammars. First, we add the set $Sp \in T$ to the tuple that defines a PEG. This defines blank terminals (e.g., space, tab, new line, etc) which should not be considered when checking for indentation. Second, as mentioned in the previous section, we add the parsing expressions $p^{\triangleright}$ and $|p|$, which describe, respectively, an indentation relation and an absolute alignment.

We must also modify the parsing relation $\overset{\text{PEG}}{\rightsquigarrow}$. First, tokens/terminals in the input string are annotated with their corresponding column in the source code. For example, the input token a^{i} represents a a at indentation i. Beware that these column superscripts are for the input string, not the grammar, and should not be confused with indentation operators. Second, the domain of the matching relation $\overset{\text{PEG}}{\rightsquigarrow}$ now includes an indentation set $I \subseteq \mathbb{N}$, a warning set W, and an absolute-alignment flag $f \in \{\ \|,\ \not\|\ \}$. Flag f is $\|$ to indicate

that we are inside an expression $|p|$ that did not consume a token/terminal yet. Otherwise, its value is $\not|$.

Figure 2 presents the complete semantics of PEGs with the new parsing expressions that allow to describe indentation information. Symbol X represents the result of a matching that can either succeed or fail.

Rule **empty.1** deals with the case of an empty parsing expression. It always succeeds and does not change the current input, indentation set, warning set, and absolute-alignment flag.

Rule **var.1** handles the case of a variable A. It basically forwards the result X of matching the rule associated with A in grammar G.

Rules **term.1**, **term.2** and **term.3** deal with the successful matching of a terminal a, whereas rules **term.4** and **term.5** handle the corresponding failure cases.

When matching a space terminal, the indentation of the actual input is not relevant (rule **term.1**), and neither the indentation set I, nor the warning set W, not the absolute-alignment flag change. However, when matching a terminal $a \notin Sp$ we will check the current indentation i against the indentation set I. In case i fits the expected indentation (rule **term.2**), the matching succeeds, the currently column becomes the new indentation set, and the absolute-alignment flag is disabled. Otherwise, the matching still succeeds and the absolute-alignment flag is disabled too, but we do not update the indentation set and we add an indentation warning to the set W.

A concatenation $p_1\ p_2$ is handled by rules **seq.1** and **seq.2**. In case p_1 succeeds, the result of the concatenation is given by the matching of p_2 against a suffix of input, where the indentation set, the warning set and the absolute-alignment flag may have be updated during the matching of p_1. When p_1 fails, the whole concatenation fails and set W may still be updated (rule **seq.2**).

Rules **rep.1** and **rep.2** deal with the matching of p^*. When the matching of p fails (rule **rep.1**), the repetition succeeds and gives the current input, indentation set and absolute-alignment flag as result, plus a possibly updated warning set. When p succeeds, we keep matching p^* against an input suffix.

In case of a predicate $!p$, it succeeds when the matching of p fails (rule **not.1**), and it fails when the matching of p succeeds (rule **not.2**). Notice that the warning set W stays unchanged; any indentation warnings that occur inside p are ignored.

In case of an ordered choice $p_1\ /\ p_2$, if p_1 succeeds this is the matching result (rule **ord.1**), otherwise the result of the choice is given by the matching of p_2 (rule **ord.2**).

Rules **ind.1**, **ind.2** and **ind.3** handle the new expression $p^{\triangleright}$. When the absolute-alignment flag is disabled ($\not|$), we perform the matching of p considering the indentation set J computed from I using the relation $\triangleright$ (rules **ind.1** and **ind.2**). Otherwise, the absolute-alignment flag is equal to $||$, we ignore the indentation relation when matching p and just use the current indentation set I (rule **ind.3**).

Lastly, rule **abs.1** turns on the absolute alignment flag when we encounter $|p|$.

3.2 Comparison with the Semanatics Proposed by Adams and Ağacan

Our semantics presented in Figure 2 has much in common with the semantics presented by Adams and Ağacan [2], but has also some important differences.

First, the work of Adams and Ağacan assumes that the input stream provides tokens for the grammar, so the PEG does not describe lexical elements. Although this is the usual in case of CFGs, it is not the case for PEGs, which seldom use a separate lexer. To allow the description of lexical elements in PEGs, we introduced the set Sp in our formalization and used it to determine if we would check the indentation when matching a terminal.

Another difference of our work is that failing to respect the alignment restrictions produces a warning instead of a failure. This was not contemplated by Adams because his original work focused on indentation-sensitive languages, but as we are interested in parsing free-form languages, failing a matching because of a misalignment seems too strict.

Finally, the formalisms used to describe the semantics of PEGs are fairly different. Adams and Ağacan give a semantics based on a step counter, heavily based on the original PEG semantics presented by Ford [4]. Our work, instead, presents a natural semantics, where the matching of a parsing expression gives us a proof tree.

4 Evaluation

In this section we evaluate the usage of a PEG parser that checks indentation to parse programs of a free-form language. To perform such evaluation, we defined a PEG for the Lua language and we use it to parse the code of a well-known Lua library. Section 4.1 presents and overview of the Lua grammar, and Section 4.2 discusses its practical usage and a few modifications that we performed. Finally, Section 4.3 discusses the behavior of our Lua parser in case of invalid input.

4.1 A PEG for Lua

Lua is a scripting language with a verbose (Pascal-like) syntax that supports several programming styles, e.g., procedural, object-oriented, functional, etc [5]. The complete syntax for Lua 5.4 is available on its manual [1].

Figure 3 shows an excerpt of our Lua grammar using PEGs, where . is an expression that matches any terminal, and p? means an optional matching. When naming rules, we use lowercase letters when defining syntactical elements and uppercase ones to define lexical elements. Of particular note

[1] https://lua.org/manual/5.4/manual.html#9

Empty $\dfrac{}{G[\varepsilon]\ x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (x, I, f, W)}$ (**empty.1**) **Non-terminal** $\dfrac{G[P(A)]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} X}{G[A]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} X}$ (**var.1**)

Terminal $\dfrac{a \in Sp}{G[a]\ a^i x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (x, I, f, W)}$ (**term.1**)

$\dfrac{i \in I \qquad a \notin Sp}{G[a]\ a^i x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (x, \{i\}, \nparallel, W)}$ (**term.2**) $\qquad$ $\dfrac{i \notin I \qquad a \notin Sp}{G[a]\ a^i x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (x, I, \nparallel, W \cup \{i\})}$ (**term.3**)

$\dfrac{a \neq b}{G[a]\ b^i x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W)}$ (**term.4**) $\qquad$ $\dfrac{}{G[a]\ \varepsilon\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W)}$ (**term.5**)

Sequence $\dfrac{G[p_1]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (y, J, g, W') \quad G[p_2]\ y\ J\ g\ W' \overset{\text{PEG}}{\rightsquigarrow} X}{G[p_1\ p_2]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} X}$ (**seq.1**) $\quad$ $\dfrac{G[p_1]\ x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W')}{G[p_1\ p_2]\ x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W')}$ (**seq.2**)

Repetition $\dfrac{G[p]\ x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W')}{G[p^*]\ x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (x, I, f, W')}$ (**rep.1**) $\quad$ $\dfrac{G[p]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (y, J, g, W') \quad G[p^*]\ J\ g\ W' \overset{\text{PEG}}{\rightsquigarrow} X}{G[p^*]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} X}$ (**rep.2**)

Negative Predicate $\dfrac{G[p]\ x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W)}{G[!p]\ x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (x, I, f, W)}$ (**not.1**) $\quad$ $\dfrac{G[p]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (y, J, g, W')}{G[!p]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W)}$ (**not.2**)

Ordered Choice $\dfrac{G[p_1]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (y, J, g, W')}{G[p_1\ /\ p_2]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (y, J, g, W')}$ (**ord.1**)

$\dfrac{G[p_1]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W') \quad G[p_2]\ xy\ I\ f\ W' \overset{\text{PEG}}{\rightsquigarrow} X}{G[p_1\ /\ p_2]\ xy\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} X}$ (**ord.2**)

Indentation $\dfrac{G[p]\ xy\ J\ \nparallel\ W \overset{\text{PEG}}{\rightsquigarrow} (y, J', \nparallel, W')}{G[p^{\triangleright}]\ xy\ I\ \nparallel\ W \overset{\text{PEG}}{\rightsquigarrow} (y, I', \nparallel, W')}$ (**ind.1**), where $J = \{j \mid j \in \mathbf{N},\ \exists i \in I,\ j \triangleright i\}$ and $I' = \{i \mid i \in I,\ \exists j \in J',\ j \triangleright i\}$

$\dfrac{G[p]\ xy\ J\ \nparallel\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W')}{G[p^{\triangleright}]\ xy\ I\ \nparallel\ W \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, W')}$ (**ind.2**), where $J = \{j \mid j \in \mathbf{N},\ \exists i \in I,\ j \triangleright i\}$ $\qquad$ $\dfrac{G[p]\ x\ I\ \|\ W \overset{\text{PEG}}{\rightsquigarrow} X}{G[p^{\triangleright}]\ x\ I\ \|\ W \overset{\text{PEG}}{\rightsquigarrow} X}$ (**ind.3**)

Absolute Alignment $\dfrac{G[p]\ x\ I\ \|\ W \overset{\text{PEG}}{\rightsquigarrow} X}{G[|p|]\ x\ I\ f\ W \overset{\text{PEG}}{\rightsquigarrow} X}$ (**abs.1**)

Figure 2. Semantics of PEGs with Indentation.

is the lexical rule SP. Following idiomatic PEG design, this rule matches spaces and it is used inside other lexical rules to match all spaces following a lexical element itself.

Overall, we followed the grammar specification provided in the Lua manual. As we mentioned before, the default indentation relation is =, so we omit it in the Lua grammar presented in Figure 3.

In comparison with the if-statements discussed previously, the defined one in Figure 3 has also an else-part, which should be at the same indentation level of the **if** keyword.

$$
\begin{aligned}
\mathit{chunk} &\leftarrow \texttt{SP}\ \mathit{block}\ \texttt{!.}\\
\mathit{block} &\leftarrow \mathit{cmd}^{*}\ \mathit{return\_cmd}?\\
\mathit{cmd} &\leftarrow \texttt{SEMICOLON}^{\geq}\ /\ \mathit{varlist}\ \texttt{EQ}^{>}\ \mathit{explist}^{\geq}\\
&\quad /\ \texttt{GOTO IDENT}^{>}\ /\ \texttt{DO}\ \mathit{block}^{>}\ \texttt{END}\\
&\quad /\ \texttt{FUNCTION}\ \mathit{funcname}^{>}\ \mathit{funcbody}\\
&\quad /\ \texttt{IF}\ \mathit{exp}^{\geq}\ \texttt{THEN}^{\geq}\ \mathit{block}^{>}\\
&\qquad (\texttt{ELSEIF}\ \mathit{exp}^{\geq}\ \texttt{THEN}^{>}\ \mathit{block}^{>})*\\
&\qquad (\texttt{ELSE}\ \mathit{block}^{>})?\ \texttt{END}^{\geq}\\
&\quad /\ \texttt{FUNCTION}\ \mathit{funcname}^{>}\ \mathit{funcbody}\\
\mathit{funcbody} &\leftarrow \texttt{LPAR}^{>}\ (\mathit{parlist}?)^{>}\ \texttt{LPAR}^{>}\ \mathit{block}^{>}\ \texttt{END}
\end{aligned}
$$

Figure 3. Excerpt of PEG for Lua 5.4 with Indentation Information.

4.2 Parsing Lua libraries

To evaluate our Lua parser, we used it to parse the code of `awesome` [2], a highly configurable window manager for the X window system.

Below, we discuss some interesting cases, where we rewrote parts of the grammar or relaxed indentation restrictions because of the programming style used in `awesome`.

In Lua, functions are first-class values and lambda expressions often appear as function parameters, as seen in this except from Awesome's `autofocus.lua`:

```
signal("property::selected", function (t)
    delayed_call(check_focus, t)
end)
```

In the previous call, the body of the lambda expression is less indented than the keyword **function**. In case our Lua parser complains about this indentation style, we will get several warnings. On the other hand, by allowing this style we would also not check the indentation of a function body in an ordinary definition of a non-anonymous function.

As an alternative solution, we created a new version of rule *funcbody*, called *funcbody_exp*. While rule *funcbody* keeps checking the indentation of the function body and the closing keyword **end**, as shown in Figure 3, rule *funcbody_exp* does not, it uses the ⊛ indentation relation.

Another indentation check that caused several warnings when parsing files from the `awesome` library was the one related to the arguments of a function, as shown below:

$$\mathit{func\_args} \leftarrow \texttt{LPAREN}\ \mathit{explist}^{>}?\ \texttt{RPAREN}^{>}\ /\ \cdots$$

In case of a function call with a long prefix or with several arguments, it is usual to break the call in several lines in a way that the arguments are not more indented than the left parentheses that indicates the call, as in the following excerpt from file `keygrabber.lua` of `awesome`:

```
self.stop_callback(
    self.current_instance, other_arguments
)
```

In this call, neither argument `self.current_instance` nor the right parentheses have the indentation expected by the grammar. To allow such style, we changed the previous definition of *func_args* to use ⊛ instead of >.

After performing similar adjustments in our Lua grammar, where we replaced other indentation relations with ⊛, but without adding new grammar rules, we ran again our corresponding PEG parser for the 468 Lua files in the `awesome` library. Some of these files are related to documentation and tests.

Overall, we could parse most Lua files (279 ≈ 59.6%) without any indentation warnings. Considering all 468 files, which have 36420 Lines of Code (LOC), we reported 707 warnings. Thus, our rate of indentation warning per LOC was 1.94%, that is, around one indentation warning for every 50 lines of code. Below, we discuss some of the indentation warnings reported.

In some files from `awesome` a Lua table is initialized as follows, where the opening curly bracket is not more indented than keyword **local**:

```
local capi =
{
    field = value
}
```

This causes an indentation warning similar to the one below, where the valid indentation range goes from 2, assuming the indentation of **local** is 1, to the maximum available integer:

```
Warning (line 2): Suspicious indentation for '{'.
Expecting [2,2147483647], but got 1.
```

Another indentation warning related to the table constructor is the following one, where the parser was expecting fields `down`, `back` and `enter` to have an indentation greater or equal than field `up`:

```
menu.menu_keys = { up = { "Up", "k" },
                   down = { "Down", "j" },
                   back = { "Left", "h" },
                   enter = { "Right", "l" } }
```

Overall, our parser reported only one warning related to the indentation of the if-statement. The reason was an one line if-statement with an *elseif* clause like the one below:

```
if x then s = 1 elseif y then s = 2 end
```

In such statement, kewywords **if** and **elseif** are not at the same indentation level. As this coding style is rare in Lua, as indicated by the few associated warnings, we did not

[2] https://github.com/awesomeWM/awesome

modify the indentation restrictions of the grammar in order to accept it.

To avoid reporting too many indentation warnings, our parser reports at most one indentation warning per line. However, a single layout mistake may lead to multiple warnings if many different lines are involved. The following example comes from Awesome's `menu.lua`:

```
 child[num]:hide()
if active_child == child[num] then
  active_child = nil
end
table.remove(child, num)
```

The first statement of the block was mistakenly indented with an extra space, causing the alignment of the block to be set more to the right than it should. The parser then deduces that the following lines (except the statement inside the `if`) are not indented enough and issues an indentation warning for each of them.

4.3 Parsing Syntatically Invalid Lua Files

Given that our semantics from Figure 2 propagates the indentation warning set also in case of a failed matching, it allows to report indentation warnings even in case of syntactically invalid files. As a drawback, it is in theory possible to obtain spurious indentation warnings originating from grammar rules that backtracked, although in the case of our Lua parser that never happened.

The capacity to report indentation warnings for syntactically invalid files may help to identify an unclosed block of statements, as in the following example:

```
while x < 100 do
  if y then
    z = z + 42

  x = x + 1
end
```

Here, the if statement is missing its `end` token. The parser shipped with the reference Lua interpreter gives a poor error diagnostic. It assumes that the `x=x+1` assignment and the `end` belong to the if statement, and it reports an error only at the end of the file, when it can't find another `end` to close the `while`. Our PEG parser, on the other hand, indicates two indentation warnings: a warning related to the assignment `x=x+1`, as it is not more indented than the if-statement; a warning related to the keyword `end`, which is not aligned with the if-statement. As these warnings refer to specific lines where there is an indentation issue, probably they will help the developer to fix the error faster.

Moreover, keeping the set of indentation warnings may be valuable in case we use some error recovery mechanism for PEGs [6, 12].

5 Related Work

The main inspiration for our work were the indentation-sensitive grammars of Adams. His first model was for context-free grammars [1] and is more suited for bottom-up-parsers. Adams' second model, developed together with Ağacan, uses parsing expression grammars [2] and is more suited for top-down recursive descent parsing.

Code formatters also must specify valid indentation layouts, however they do so as part of the pretty printers rather than the parser. That is, in a typical code formatter the program is first parsed, discarding any indentation from the source file, and then pretty-printed back into well-formatted code. Because the original layout information is discarded, code formatters are usually geared for converting code to a standard layout, rather than verifying the layout of existing code. Because the parser and pretty printer are decoupled, there is the advantage that any parser algorithm can be used, but the disadvantage that the printer can only operate on valid syntax trees and that changes in the grammar may also require some rewriting in the corresponding code formatter. Among the many algorithms used for pretty printing, we can highlight the Coutaz's box model [3] and Wadler's algebra of documents [13], which build a set of layout constraints from the AST and then use a constraint solver to decide where to insert linebreaks. There are also pretty printer algorithms that focus on speed, such as the linear time heuristics of Oppen [9].

Parr and Vinju [11] propose a machine learning-based approach to format programs. In their approach, we need to provide a corpus D of programs for a given language L, where such corpus should have a reasonable formatting consistency. The training phase uses D besides a lexer and a parser for L derived from a grammar G. Changing grammar G does not affect much the resulting formatting style. As in other code formatters approaches, it is not possible to rewrite only specific parts of a file. In case a user wants to enforce a different coding style, in our approach she needs to edit the grammar and change some indentation relations, while in case of Parr and Vinju it is necessary to get a different corpus D' with the desired sytle or to edit the files in corpus D.

Nilsson-Nyman et al. [8] describe a parsing algorithm that uses indentation to help identify matching pairs of delimiter tokens and thus improve syntax error recovery. They specify indentation by means of a separate *bridge grammar*, which makes their technique compatible with an external parser generator. However, because the bridge model is focused on delimiter tokens, it cannot specify indentation layouts in general.

6 Conclusion

In this work, we presented an extension of parsing expression grammars that can specify indentation layouts for free-form

programming languages. We presented several examples describing how to use these grammars to model different indentation styles.

We also evaluated the effectiveness of the compilation warnings by specifying an indentation style for Lua and testing how many indentation errors it identified in a widely-used Lua codebase. Our analysis concluded that Adams' indentation model was capable of specifying useful indentation rules, despite ignoring line numbers and caring only about column numbers. However, there were several situations where we had to work around this limitation by introducing $\geq$ or $\circledast$ relations, which weaken the grammar. For future work, it might be worth investigating indentation models that care about both line and column numbers, which would allow different indentation behavior when two tokens are on the same line.

ARTIFACT AVAILABILITY

This paper is accompanied by an artifact[3] that contains the indentation-sensitive Lua parser, as well as the Lua source files from the `awesome` library, which were used as inputs to our experiments.

ACKNOWLEDGMENTS

Hugo Gualandi received financial support from Conselho Nacional de Desenvolvimento Científico e Tecnológico — Brasil (CNPq) — project code 403601/2023-1.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) – Finance Code 001.

REFERENCES

[1] Michael D. Adams. 2013. 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* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 511–522. doi:10.1145/2429069.2429129

[2] Michael D. Adams and Ömer S. Ağacan. 2014. Indentation-sensitive parsing for Parsec. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 121–132. doi:10.1145/2633357.2633369

[3] Joëlle Coutaz. 1985. A layout abstraction for user-system interface. *SIGCHI Bull.* 16, 3 (Jan. 1985), 18–24. doi:10.1145/1044201.1044202

[4] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. ACM, New York, NY, USA, 111–122.

[5] Roberto Ierusalimschy. 2016. *Programming in lua* (4 ed.). Lua.org.

[6] Sérgio Medeiros and Fabio Mascarenhas. 2018. Syntax error recovery in parsing expression grammars. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 1195–1202. doi:10.1145/3167132.3167261

[7] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. 1983. Program indentation and comprehensibility. *Commun. ACM* 26, 11 (Nov. 1983), 861–867. doi:10.1145/182.358437

[8] Emma Nilsson-Nyman, Torbjörn Ekman, and Görel Hedin. 2009. Practical Scope Recovery Using Bridge Parsing. In *Software Language Engineering*, Dragan Gašević, Ralf Lämmel, and Eric Van Wyk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 95–113.

[9] Dereck C. Oppen. 1980. Prettyprinting. *ACM Trans. Program. Lang. Syst.* 2, 4 (Oct. 1980), 465–483. doi:10.1145/357114.357115

[10] Jan Ouwens. 2024. Why are there no decent code formatters for Java? https://jqno.nl/post/2024/08/24/why-are-there-no-decent-code-formatters-for-java/. https://jqno.nl/post/2024/08/24/why-are-there-no-decent-code-formatters-for-java/

[11] Terence Parr and Jurgen Vinju. 2016. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (Amsterdam, Netherlands) *(SLE 2016)*. Association for Computing Machinery, New York, NY, USA, 137–151. doi:10.1145/2997364.2997383

[12] Sérgio Queiroz de Medeiros, Gilney de Azevedo Alvez Junior, and Fabio Mascarenhas. 2020. Automatic syntax error reporting and recovery in parsing expression grammars. *Science of Computer Programming* 187 (2020), 102373. doi:10.1016/j.scico.2019.102373

[13] Philip Wadler. 2003. *A prettier printer.* Palgrave Macmillan. 223–243 pages.

[3] https://dx.doi.org/10.5281/zenodo.16592251