Declarative specification of indentation rules

A tooling perspective on parsing and pretty-printing layout-sensitive languages

de Souza Amorim, Eduardo; Erdweg, Sebastian; Steindorfer, Michael; Visser, Eelco

DOI
10.1145/3276604.3276607

Publication date
2018

Document Version
Accepted author manuscript

Published in
SLE 2018 - Proceedings of the 11th ACM SIGPLAN International Conference on Soft ware Language Engineering

Citation (APA)
de Souza Amorim, L. E., Erdweg, S., Steindorfer, M. J., & Visser, E. (2018). Declarative specification of indentation rules: A tooling perspective on parsing and pretty-printing layout-sensitive languages. In D. Pearce , S. Friedrich, & T. Mayerhofer (Eds.), SLE 2018 - Proceedings of the 11th ACM SIGPLAN International Conference on Soft ware Language Engineering (pp. 3-15). New York, NY: Association for Computing Machinery (ACM). https://doi.org/10.1145/3276604.3276607

Declarative Specification of Indentation Rules

A Tooling Perspective on Parsing and Pretty-Printing Layout-Sensitive Languages

Luís Eduardo de Souza Amorim
Delft University of Technology
The Netherlands
l.e.desouzaamorim-1@tudelft.nl

Michael J. Steindorfer
Delft University of Technology
The Netherlands
michael@steindorfer.name

Sebastian Erdweg
Delft University of Technology
The Netherlands
s.t.erdweg@tudelft.nl

Eelco Visser
Delft University of Technology
The Netherlands
e.visser@tudelft.nl

Abstract

In layout-sensitive languages, the indentation of an expression or statement can influence how a program is parsed. While some of these languages (e.g., Haskell and Python) have been widely adopted, there is little support for software language engineers in building tools for layout-sensitive languages. As a result, parsers, pretty-printers, program analyses, and refactoring tools often need to be handwritten, which decreases the maintainability and extensibility of these tools. Even state-of-the-art language workbenches have little support for layout-sensitive languages, restricting the development and prototyping of such languages.

In this paper, we introduce a novel approach to declarative specification of layout-sensitive languages using *layout declarations*. Layout declarations are high-level specifications of indentation rules that abstract from low-level technicalities. We show how to derive an efficient layout-sensitive generalized parser and a corresponding pretty-printer automatically from a language specification with layout declarations. We validate our approach in a case-study using a syntax definition for the Haskell programming language, investigating the performance of the generated parser and the correctness of the generated pretty-printer against 22191 Haskell files.

CCS Concepts • **Software and its engineering → Syntax**; *Parsers*;

Keywords parsing, pretty-printing, layout-sensitivity

SLE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276607

ACM Reference Format:
Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. 2018. Declarative Specification of Indentation Rules: A Tooling Perspective on Parsing and Pretty-Printing Layout-Sensitive Languages. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3276604.3276607

1 Introduction

Layout-sensitive (also known as indentation-sensitive) languages were introduced by Landin [16]. The term characterizes languages that must obey certain *indentation rules*, i.e., languages in which the indentation of the code influences how the program should be parsed. In layout-sensitive languages, alignment and indentation are essential to correctly identify the structures of a program. Many modern programming languages including Haskell [10], Python [21], Markdown [13] and YAML [4] are layout-sensitive. To illustrate how layout can influence parsing programs in such languages, consider the Haskell program in Figure 1, which contains multiple *do*-expressions:

```
1 guessValue x = do
2   putStrLn "Enter your guess:"
3   guess <- getLine
4   case compare (read guess) x of
5     EQ -> putStrLn "You won!"
6     _  -> do putStrLn "Keep guessing."
7              guessValue x
```

Figure 1. *Do*-expressions in Haskell.

In Haskell, all statements inside a *do*-block should be aligned (i.e., should start at the same column). In Figure 1, we know that the statement on line 7 (`guessValue x`) belongs to the inner *do*-block solely because of its indentation. If we modify the indentation of this statement, aligning it with the statements in the outer *do*-block, the program would have a different interpretation, looping indefinitely.

While layout-sensitive languages are widely used in practice, their tools are often handwritten, which prevent their adoption by language workbenches or declarative language frameworks. State-of-the-art solutions for declarative specification of layout-sensitive languages extend context-free grammars to automatically generate layout-sensitive parsers from a language specification, but are limited by their usability, performance and tooling support. For example, Adams [1] proposes a new grammar formalism called *indentation-sensitive context-free grammars* to declaratively specify layout-sensitive languages. However, this technique requires modifying the original symbols of the context-free grammar and, as result, may produce a larger grammar in order to specify certain indentation rules. Erdweg et al. [11] propose a less invasive solution using a generalized parser, requiring only that productions of a context-free grammar are annotated with *layout constraints*. In these constraints, language engineers are required to encode indentation rules, such as alignment or *Landin's offside rule*,[1] at a low-level of abstraction, that is, by comparing lines and columns. In both solutions, parsing may introduce a large performance overhead.

Both approaches ignore an essential tool in a language workbench: *pretty-printers*. Pretty-printers play an important role since they transform trees back into text. This transformation is crucial to developing many of the features provided by a language workbench, such as refactoring tools, code completion, and source-to-source compilers. Deriving a layout-sensitive pretty-printer from a declarative language specification is challenging as the pretty-printer must be *correct*, i.e., the layout used to pretty-print the program must not change the program's meaning.

In this paper, we propose a novel approach to declaratively specifying layout-sensitive languages. We take a holistic approach by considering a domain-specific language to specify common indentation rules of layout-sensitive languages that is (a) general enough to support both parsing and pretty-printing, and (b) lets the user express indentation rules without resorting to low-level constraints in terms of lines and columns.

We make the following contributions.

- We define a domain-specific notation that concisely captures common patterns for indentation rules that occur in layout-sensitive languages (Section 3).
- We discuss our implementation of a layout-sensitive generalized parser with efficient support for parse-time disambiguation of layout constraints (Section 4).
- We present an algorithm for deriving correct layout-sensitive pretty-printers from grammars with layout declarations (Section 5).
- We evaluate the performance and correctness of our solution on a benchmark introduced by Erdweg et al. [11], exercising 22191 Haskell files (Section 6).

We cover related work in Section 7, discussing future work in Section 8, and concluding in Section 9.

[1] Landin introduced the *offside rule*, enforcing that in a program of a layout-sensitive language, all the subsequent lines of certain structures of the language should be "further to the right" than the first line of the corresponding structure. If the tokens of the subsequent lines occur further to the left than the first line, they are *offside*, and the structure is invalid.

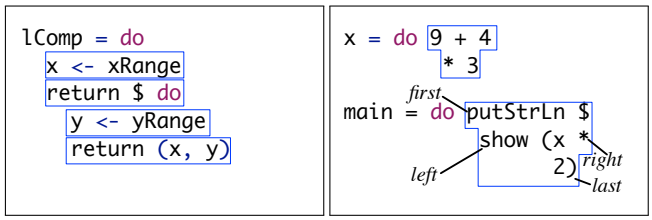
2 Background

In this section, we motivate our work on declarative specification of layout-sensitive languages by providing an overview of *layout constraints* [11], enumerating their shortcomings when used in a language workbench.

2.1 Layout Constraints

In layout-sensitive languages, indentation and alignment define the *shape* of certain structures of the language and the relationship between these shapes, such that for the structures to be valid, their shape must adhere to certain rules. A shape can be constructed as a box, with boundaries around the non-layout tokens that constitute the structure.

For example, consider the code from Figures 2a and 2b. In Figure 2a, because the list of statements inside a do-expression should be aligned, each shape indicating a single statement of the list must start at the same column. Similarly, if we consider that each statement in the do-expressions from Figure 2b should obey the offside rule, if the statement spans multiple lines, it must have a shape similar to ⌝ (but not □ or ⌜, for example).



(a) (b)

Figure 2. Boxes used to highlight the shape of subtrees in *do*-expressions.

Layout constraints can be used as annotations in productions of context-free grammars to enforce specific shapes into the source code of abstract syntax trees. Each tree exposes its shape given the location of four tokens in its original source code: `first`, `last`, `left`, and `right`—called *token selectors*—as shown in Figure 2b. The token selectors `first` and `last` access the position of the first and last tokens in a tree, respectively. The selector `left` selects the leftmost non-whitespace token that is not on the same line as the first token, whereas the selector `right` selects the rightmost non-whitespace token that is not on the same line as `last`.

Together with token selectors, a layout constraint may also refer to a specific indentation element of the source

code—called *position selectors*—`line` and `col`, which yield the token's line and column offsets, respectively. For example, a layout constraint `layout(x.left.col > x.first.col)` indicates that the subtree at position `x` should follow Landin's offside rule.

Note that constraints may also mention multiple subtrees in an annotated production, defining the relative position of these subtrees. That is the case in the constraint used to indicate that all statements inside a do-expression should be aligned, i.e., `layout(x.first.col == y.first.col)`. Finally, note that constraints may also be combined using the boolean operators *and* (`&&`), and *or* (`||`), and a constraint `ignore-layout` can be used deactivate layout validation locally.

2.2 Tools for Layout-Sensitive Languages

While layout constraints can be used to generate layout-sensitive parsers, there has been little adoption of such specifications by tools such as language workbenches.

Language workbenches enable agile development and prototyping of programming languages by generating an integrated development environment (IDE) from a language specification [12]. Therefore, one of the requirements for language specifications of layout-sensitive languages is related to the *usability* of the specifications, i.e., they must be declarative, concise and easy to use. Furthermore, when using an IDE, language users expect rapid feedback from the editor when editing their programs. Hence, the *performance* of the tools generated from a language specification is another important concern when using a language workbench to develop layout-sensitive programming languages. Finally, language workbenches go beyond parsing and code generation, providing many different features to language users, such as refactorings and code completion. Thus, another concern when developing a layout-sensitive language using a language workbench consists of specifying a *pretty-printer*, which transforms the abstract syntax tree of a program back into source code.

Below, we discuss the shortcomings of layout constraints against these requirements.

Usability. Layout constraints require annotating context-free productions to indicate how the source code corresponding to subtrees should be indented. However, they are rather verbose and low-level, since they involve comparing lines and columns of tokens of different subtrees.

Parsing Performance. Generating tools from a language specification increases maintainability and extensibility, but usually comes with a penalty in performance. For example, Erdweg et al. [11] reported an overhead of about 80% when using a layout-sensitive generalized LR parser that uses layout constraints to disambiguate Haskell programs.

Pretty-printing. Layout constraints can be used to generate parsers, but it is not clear how to use them to automatically derive pretty-printers. One of the challenges when generating a pretty-printer for a layout-sensitive language is that the pretty-printer must be correct, i.e., pretty-printing a program should not change its meaning.

In the remainder of this paper, we show how we tackle each of these concerns, such that language designers can develop layout-sensitive languages using tools such as language workbenches.

3 Layout Declarations

To improve the usability of declarative specifications for layout-sensitive languages, we introduce *layout declarations*: high-level annotations in productions of a context-free grammar that enforce indentation rules on a specific node of the abstract syntax tree. Layout declarations abstract over token and position selectors, and provide a concise specification for most common indentation rules: *alignment* and *indentation* of constructs, and the *offside* rule. We also equip layout declarations with *tree selectors*, allowing them to be more readable than when using the position of the subtree involved in a declaration.

3.1 Tree Selectors

When writing the original layout constraints, one must use the position of the subtree in a production to enforce a constraint over this subtree. However, when reading and writing layout constraints, we want to avoid counting terminals and non-terminals in the production to identify to which tree the constraint applies.

Layout declarations allow the specification of constraints using *tree selectors*. Tree selectors may consist not only of the number of the subtrees, but also *literals* and *labeled non-terminals* that occur in the production. A labeled non-terminal is a non-terminal preceded by a label and a colon. Labels must be unique within a production, and if a literal occurs multiple times in the same production, then they must be referred by its position. For example, consider the following productions, written using SDF3 [25] syntax:[2]

```
Exp.Seq = exp1:Exp ";" exp2:Exp ";"
Exp.Add = exp:Exp "+" exp:Exp {left}
```

In the first production, the first `Exp` subtree might be referred in a layout declaration by its position (1) or by the label `exp1`. Considering the same production, the literals `";"` must be referred by their position, as they occur multiple times in the production. In the second production, the literal `"+"`, can be referred using the literal itself, as it is unique within the production. Finally, note that the underlined label

[2] SDF3 productions have the form: `A.C = X1 X2 ... Xn {annos}`, where the symbol `A` represents a non-terminal, X_i represents either a terminal or a non-terminal, and the constructor `C` indicates the name of the node in the abstract syntax tree when imploding the parse tree. The list of annotations inside brackets `annos` can be used for different purposes, such as operator precedence disambiguation or to specify layout constraints.

in the second production is invalid, because the same label is used on the first Exp non-terminal.

3.2 Alignment

A common rule in layout-sensitive languages requires that certain structures must be aligned in the source code. For instance, as shown previously, all statements in a *do*-block of a Haskell program must be aligned, i.e., they must start at the same column. To express this indentation rule using layout constraints, one may use the following productions:

```
Exp.Do           = "do" StmtList
StmtList.Stmt    = Stmt
StmtList.StmtSeq = Stmt StmtList
  {layout(1.first.col == 2.first.col)}
```

Instead of using low-level concepts such as token and position selectors, we propose using high-level layout declarations **align** or **align-list** to indicate alignment of structures in the source code. A declaration **layout**(**align** ref t) enforces that a tree indicated by the tree selector t should start in the same column as the tree indicated by the ref tree selector, used as reference. Consider the example below, which uses an **align** declaration to indicate that the tail of the list StmtList should be aligned with the head of the list:

```
Exp.Do           = "do" StmtList
StmtList.Stmt    = Stmt
StmtList.StmtSeq = head:Stmt tail:StmtList
  {layout(align head tail)}
```

In SDF, lists may be represented by specific non-terminals (A+ or A*), which instructs the parser to flatten the tree structure corresponding to the list when constructing the abstract syntax tree. However, using layout constraints requires explicitly defining productions for lists, which breaks this abstraction. The layout declaration **align-list** can be applied to list non-terminals to indicate that all elements in a list should start at the same column. Thus, one may write:

```
Exp.Do = "do" stmts:Stmt+
  {layout(align-list stmts)}
```

to indicate that the statements in the list should be aligned.

Semantics We define translation rules from layout declarations that describe alignment to layout constraints using token and position selectors. Consider the tree selectors x and y, the function *pos*(t), which obtains the position of a subtree indicated by selector t, the function *rename*(X, Y), which locally renames a non-terminal X to a non-terminal Y, and the following equations:

$$\frac{\texttt{align x y} \quad pos(\texttt{x}) = \texttt{x'} \quad pos(\texttt{y}) = \texttt{y'}}{\texttt{x'.first.col == y'.first.col}} \tag{1}$$

$$\frac{\texttt{align-list x} \quad \texttt{x is a tree selector for A+ (or A*)}}{rename(\texttt{A+}, \texttt{A'+})} \tag{2}$$

```
A'+ = A'+ A layout(1.first.col == 2.first.col)
```

Note that in Equation 2, using **align-list** enforces the layout constraint on the list A+ (or A*), which could affect all occurrences of the list in the grammar. Therefore, we first locally rename this non-terminal A+ to a non-terminal A'+, restricting the alignment declaration to the particular list in the production annotated with **align-list**. In Equation 1, on the other hand, the layout declaration can be directly translated to the layout constraint involving token and position selectors.

3.3 Offside Rule

As mentioned before, the offside rule is a common indentation rule applied in layout-sensitive languages. This rule requires that any character in the subsequent lines of a certain structure occur in a column that is further to the right than the column where the structure starts. For example, consider the following productions, which contains a layout constraint that requires that the source code for the OffsideStmt tree obey the offside rule:

```
Exp.Do = "do" Stmt
Stmt.OffsideExp = Exp
   {layout(1.left.col > 1.first.col)}
```

According to this rule, the expression in the following statement is invalid, since the second line starts at a column that is to the left of the column where the statement inside the do-expression starts:

```
do 21 + 7
* 3
```

In fact, any statement in which the multiplication sign is at the left of the digit 1 is invalid. By contrast, a valid program that satisfies the offside rule is:

```
do 21 + 7
    * 3
```

Instead of using layout constraints, one may use the **offside** layout declaration to achieve the same effect:

```
Stmt.OffsideExp = exp:Exp
  {layout(offside exp)}
```

The offside layout declaration can also be used to specify the relationship between the leftmost column of subsequent lines of a tree, and the initial column of another tree. For example, consider the following productions:

```
Exp.Do = "do" stmt:Stmt
  {layout(offside "do" stmt)}
Stmt.ExpStmt = Exp
```

With this declaration, the subsequent lines of Stmt should be in a column to the right of the column where the literal **do** starts. For example, even if we do not consider the offside rule for the inner statement, the following program is still invalid:

```
do 21 + 7
* 3
```

as the symbol * occurs at the same column as the keyword `do`, i.e., it is *offside*.

Semantics We define the semantics of layout declarations `layout(offside t)` and `layout(offside ref t)` by a translation into layout constraints using tokens and position selectors. Consider the following equations with `x` and `y` as tree selectors:

$$\frac{\texttt{offside x} \quad pos(\texttt{x}) = \texttt{x'}}{\texttt{x'.left.col > x'.first.col}} \tag{3}$$

$$\frac{\texttt{offside x y} \quad pos(\texttt{x}) = \texttt{x'} \quad pos(\texttt{y}) = \texttt{y'}}{\texttt{y'.left.col > x'.first.col}} \tag{4}$$

In Equation 3, the declaration `offside x` specifies that the `left` token of the tree `x` should be in a column further to the right than its `first` token. Similarly, in Equation 4, the `offside` declaration between the trees `x` and `y` specifies that the `left` token of `y` should be in a column further to the right than the `first` token of `x`.

3.4 Indentation

Another common pattern in layout-sensitive languages is to enforce indentation between subtrees. That is, a subtree should have its first token at a column to the right of the column of the first token of another subtree. Consider for example, the following productions:

```
Exp.Do = "do" stmt:Stmt
  {layout(indent "do" stmt)}
Stmt.ExpStmt = Exp
```

The declaration in the first production indicates that the statement should start further to the right than the `do` keyword. Thus, this declaration invalidates the following program:

```
do
21 + 7 * 3
```

On the other hand, the following program obeys the declaration, as the expression statement starts further to the right, when compared to the `do` keyword:

```
do 21 + 7 * 3
```

Similar to the `indent` layout declaration, the declaration `newline-indent` allows enforcing that a target subtree should start at a column further to the right than another subtree. Moreover, the latter declaration also enforces that the target subtree starts at a line below the line where the reference subtree ends. Thus, when considering this layout declaration, the program presented previously would also be invalid. A valid program would then be:

```
do
 21 + 7 * 3
```

Semantics The `indent` and `newline-indent` declarations are rewritten into layout constraints involving token and position selectors. Consider `x` and `y` tree selectors and the following equations:

$$\frac{\texttt{indent x y} \quad pos(\texttt{x}) = \texttt{x'} \quad pos(\texttt{y}) = \texttt{y'}}{\texttt{y'.first.col > x'.first.col}} \tag{5}$$

$$\frac{\texttt{newline-indent x y} \quad pos(\texttt{x}) = \texttt{x'} \quad pos(\texttt{y}) = \texttt{y'}}{\texttt{y'.first.col > x'.first.col \&\& y'.first.line > x'.last.line}} \tag{6}$$

Note that the layout declaration `newline-indent` requires a conjunction between two constraints involving the columns of the `first` tokens of both trees referenced by `x` and `y`, and the line of the `last` token of the tree `x` and the line of the `first` token of the tree `y`.

4 Layout-Sensitive Parsing

Parsing layout-sensitive languages is difficult because these languages cannot be straightforwardly described by traditional context-free grammars. Such languages require counting the number of whitespace characters in addition to keeping track of nesting, which requires context-sensitivity. Therefore, most parsers for layout-sensitive languages rely on some ad-hoc modification to a handwritten parser. For example, the Python language specification describes a modified scanner that preprocesses the token stream, generating **newline**, **indent** and **dedent** tokens to keep track of when the indentation changes. Meanwhile, Python's grammar assumes these tokens to enforce the indentation rules of the language. In Haskell, an algorithm that runs between the lexer and parser converts implicit layout into explicit semicolons and curly braces to determine how the structures should be parsed by a traditional context-free grammar.

Because modifications to the parser vary from language to language, they are hard to implement when deriving a parser from a declarative language specification. Therefore, in this section, we propose a solution similar to Erdweg et al.'s, which consists of deriving a scannerless generalized layout-sensitive LR parser (SGLR) from a language specification. Our algorithm improves on Erdweg et al.'s implementation by performing parse-time disambiguation of layout constraints, in contrast to post-parse disambiguation.

4.1 Layout-Sensitive SGLR

In theory, traditional context-free grammars can be used to generate a generalized parser for layout-sensitive languages. Since the parser produces a parse forest containing all possible interpretations of the program, this forest can then be traversed, such that only the trees that obey the indentation rules of the language are produced as result.

In practice this approach does not scale, since ambiguities caused by layout can grow exponentially [11], making it

infeasible to traverse all trees in a parse forest produced when parsing a program of a layout-sensitive language. Thus, disambiguation of layout constraints at parse time should be preferred over post-parse disambiguation [15, 22].

We propose an implementation of a scannerless generalized LR parser (SGLR), that rejects trees that violate layout constraints at parse time. Our implementation calculates position information (line and column offsets for starting and ending positions) for token selectors (**first**, **last**, **left**, and **right**), propagating this information when building the trees bottom up, and using this information to evaluate layout constraints. The main difference between our implementation and the one proposed by Erdweg et al. [11] is that we evaluate all layout constraints at parse time, when building the subtrees, whereas in Erdweg et al.'s implementation, disambiguation using **left** and **right** constructs is performed after parsing (we discuss their implementation in more detail in Section 7).

Position Information The first modification we propose to add layout-sensitivity to the original SGLR algorithm [23] is to add position information to every tree node. That is, each node of the parse tree should contain the line and column at which it begins, and the line and column at which it ends. This information can be obtained from the parser, since it keeps track of the position in the source code when it starts and finishes parsing a structure. Besides that, our algorithm also calculates the position information for the **left** and **right** tokens of every tree node. We present the algorithm that constructs parse tree nodes in Figure 3.

The algorithm propagates position information about token selectors based on the subtrees of the tree node being constructed. The method CREATE-TREE-NODE takes as arguments the production being applied, the list of trees that represent the subtrees of the node being created, and two `Position` variables `beginPos` and `curPos`, indicating the line and column where the tree starts and the line and column where the parser is currently at, respectively. The algorithm first constructs a tree node `t` given its list of subtrees, as shown in line 2. In lines 3 and 4, the information about the **first** and **last** tokens of `t` are assigned to the current node given the arguments `beginPos` and `endPos`.

The remainder of the algorithm computes the information about **left** and **right**. The algorithm calculates the position information about **left** by processing the list of subtrees, as its value should be the leftmost value (the one in the lowest line, and lowest column), when considering the **left** tokens of all subtrees that do not represent layout (line 14). However, if any subtree starts in a line that is below the line where `t` starts, the algorithm updates the **left** token of `t` accordingly (line 16). A similar strategy is applied to calculate the information about the **right** token.

Enforcing Layout Constraints The layout-sensitive SGLR algorithm works by rejecting trees that violate the layout

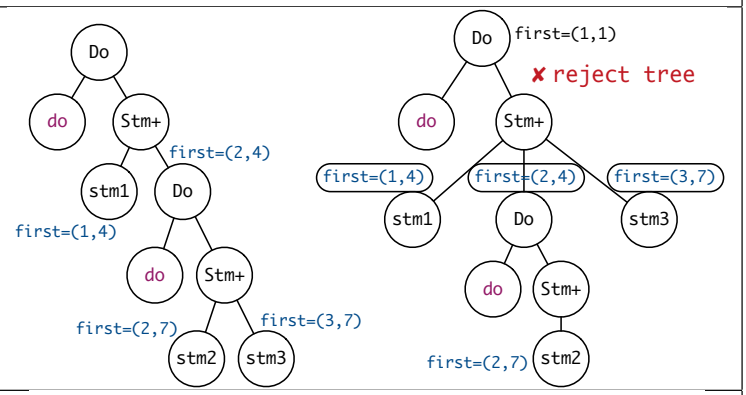
```
1  function CREATE-TREE-NODE(Production A.C = X1 ... Xn,
       List<Tree> [t1, ..., tn], Position beginPos,
       Position curPos)
2    Tree t = [A.C = t1, ..., tn]
3    t.first = beginPos
4    t.last  = curPos
5    t.left  = null
6    t.right = null
7
8    // calculate left and right
9    foreach(ti in t) {
10     // should not consider layout
11     if (isLayout(ti))
12       continue
13     if (ti.left != null)
14       t.left = leftMost(t.left, ti.left)
15     if (ti.first.line > t.first.line)
16       t.left = leftMost(t.left, ti.first)
17
18     if (ti.right != null)
19       t.right = rightMost(t.right, ti.right)
20     if (ti.last.line < t.last.line)
21       t.right = rightMost(t.right, ti.last)
22   }
23   return t
24 end function
25
26 function leftMost(p1, p2) {
27   if (p1 == null || p1.col > p2.col)
28     return p2
29   else return p1
30 end function
31
32 function rightMost(p1, p2) {
33   if (p1 == null || p1.col < p2.col)
34     return p2
35   else return p1
36 end function
```

Figure 3. Pseudocode for the modified *CREATE-TREE-NODE* method from the original SGLR and the auxiliary functions *leftMost* and *rightMost*, in the implementation of the layout-sensitive SGLR.

constraints defined in a production using the information collected in the algorithm of Figure 3. A layout constraint is enforced at parse time when executing reduce actions in the parser, i.e., in the function DO-REDUCTIONS [23]. In layout-sensitive SGLR, a reduction is performed only when a production does not define a layout constraint, or when the layout constraint it defines is satisfied.

For example, the trees in Figure 4 indicate how the parser constructs tree nodes and verifies layout constraints. For the first program, the layout constraint states that the statements must be aligned. Therefore, since the second tree for this program does not satisfy this constraint, the tree is rejected as

Source Code	Layout Constraints	Trees

`do stm1`
`  do stm2`
`  stm3`
	`Exp.Do = "do" stmts:Stmt+`
	`  {layout(align-list stmts)}`
		

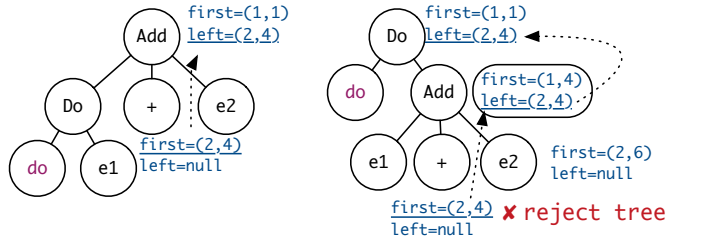
`do e1` `  + e2`	`Stmt.OffsideExp = exp:Exp` `  {layout(offside exp)}`	

Figure 4. Example of how our algorithm for a layout-sensitive SGLR constructs trees and applies layout constraints.

the parser does not perform the reduce action to construct it. In the second program, we can see how the information about the `left` is propagated. Similarly to the first example, the first tree constructed when parsing this program is the only one produced by the parser, since the second tree violates the offside rule.

4.2 Propagation of `left` and `right` at Parse Time

In the algorithm presented in Section 4.1, we propagate position information about `left` and `right` while building the parse tree. However, this approach may not produce the correct result in all scenarios. For example, consider a parse forest containing two different parse trees. Suppose that the source code for each tree in the parse forest is indicated by the programs below, where the symbols `*` represent actual characters in the program, and `-` represents a comment:

```
*****      *****
****       ----
***         ***
```

Considering that both programs start at column 1, in the first tree, the `left` token is at column 2, whereas in the second tree, `left` is actually at column 3, because part of its source code is a comment. Thus, it is unclear what is the actual value for `left` when considering the parse forest, i.e., both trees simultaneously.

While this could be a problem when propagating position information about `left` and `right` tokens, and applying layout constraints at parse time, we believe that this scenario does not occur often in practice. As an alternative solution, we could adapt our SGLR algorithm to fall back to post-parse disambiguation in such cases.

5 Layout-Sensitive Pretty-Printing

A pretty-printer is a tool that transforms an abstract syntax tree back into text. Pretty-printers are key components of language workbenches. For example, they can be used by other tools such as refactoring tools and code completion, or when defining source-to-source compilers. A lack of pretty-printing support effectively prevents the adoption of language workbenches for layout-sensitive languages.

Pretty-printing programs in a layout-sensitive language is not an easy task. Because the layout in the source code identifies how the code should be parsed, the pretty-printer needs to be designed such that the meaning of the original program does not change after it is pretty-printed. Thus, in general, a pretty-printer is *correct* if the same abstract syntax tree is produced when parsing both the original and the pretty-printed programs. More formally, if we consider a program p and parsing and pretty-printing as two functions `parse` and `prettyPrint`, the following equation must hold:

$$\texttt{parse}(p) = \texttt{parse}(\texttt{prettyPrint}(\texttt{parse}(p)))$$

In this section we propose a technique to derive a correct pretty-printer based on a language specification containing layout declarations. We use strategies to apply modifications

to the pretty-printed program, such that each layout declaration is considered while performing a top-down traversal in an intermediate representation of the abstract syntax tree.

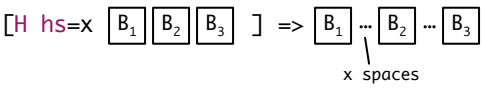
5.1 From Trees to Boxes

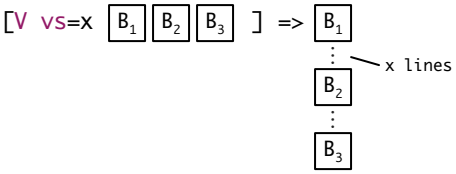
A naive implementation of a pretty-printer consists of printing the program separating each token by a single whitespace. However, it is easy to see that for a layout-sensitive language that enforces alignment, our naive pretty-printer would produce an invalid result as the pretty-printed program would not contain any newlines.

Manipulating this string directly to fix the layout according to the indentation rules of the language is also not ideal, as we lose the information about the structure of the program and the layout declarations encoded in the abstract syntax tree. Therefore, in order to produce an abstract representation of a program that takes into account the program structure and its layout, we use the Box language [19**?** , 20] as an intermediate representation.

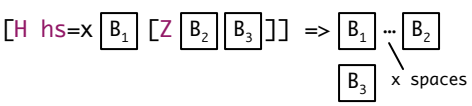
Boxes provide a structured representation of the pretty-printed text. Each node in the abstract syntax tree can be translated into a box, with its subtrees recursively translated into sub-boxes. The most basic boxes are string boxes, which can be composed (nested) using composition operators. Our approach considers three different composition operators in the Box language: vertical composition (**V**), horizontal composition (**H**) and z-composition (**Z**) [24].

The horizontal composition operator concatenates a list of boxes into a single line, whereas the vertical composition operator concatenates a list of boxes putting each box into a different line, starting at the same column. Each operator optionally takes an integer **hs** or **vs** as parameter to determine the number of spaces or empty lines separating each box, respectively. To illustrate, consider the examples below:





The z-composition operator places its boxes vertically on separate lines resetting the indentation of all boxes after the first to 0. Thus, for those boxes, the indentation from surrounding boxes is ignored and they start at the left margin. For example, if we combine the horizontal operator and the z-composition operator, we obtain the following output:



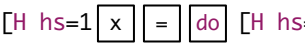
Boxes can be easily converted into text by recursively applying the box operators, as shown by the examples. Therefore, instead of manipulating the string produced by the pretty-printer, we manipulate boxes to enforce the layout declarations from the language specification.

5.2 Applying Layout Declarations to Boxes

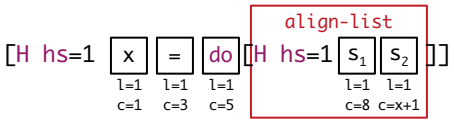
Boxes provide information about the layout of the program, retaining the structure of the abstract syntax tree. In order to apply layout declarations to the boxes generated from pretty-printing a tree, each box should also contain its relative line and column positions in the pretty-printed program. For example, consider the following Haskell program, pretty-printed from a naive pretty-printer, as discussed previously:

```
x = do s1 s2
```

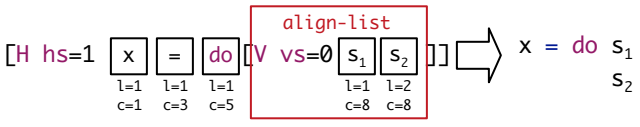
One possible box representation for this program is:



To apply layout declarations to this program, we attach the relative line and column positions in the source code to the box (indicated by `l` and `c` in the diagram below). Furthermore, since boxes are created from the nodes in the abstract syntax tree, we also attach to the boxes the layout declarations from the corresponding node in the abstract syntax tree. Assuming that s_1 ends at column x, our pretty-printer produces the following boxes:

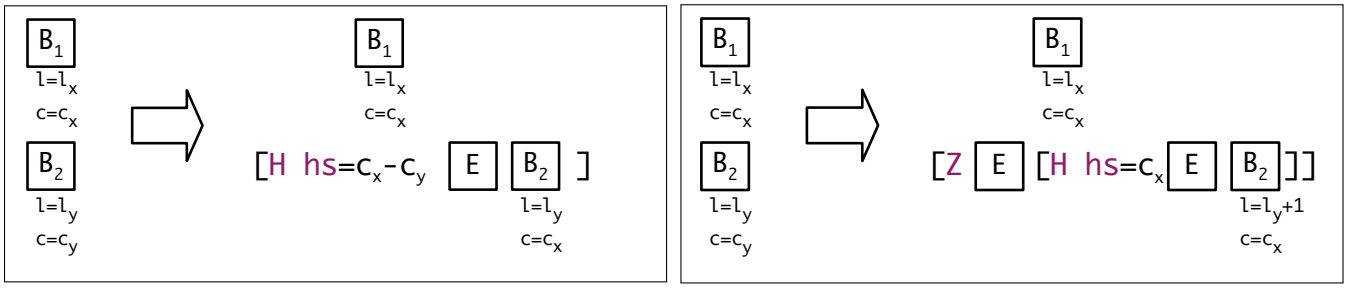


Transforming this box into a string and parsing that string results in a syntax error, since the statements inside the do-expression do not start at the same column. To ensure correct use of layout in the pretty-printed string, we apply a *layout fixer* that traverses the boxes and fixes the indentation where necessary. In this case, when considering an **align-list** layout declaration, the layout fixer replaces the inner horizontal operator by a vertical operator producing the following boxes and pretty-printed program:



which satisfies the layout declaration.

We adopt a similar strategy for adapting the boxes for the remaining layout declarations. For a layout declaration **align** `x y`, the left-most column of a box B_2 corresponding to the tree indicated by `y` should be equal to the left-most column of a box B_1 from the reference tree `x`. To satisfy this layout declaration, if B_2 starts at a column to the left of the



(a) $c_x > c_y$ **(b)** $c_y > c_x$

Figure 5. Manipulating boxes to apply a layout declaration that enforces alignment between the boxes B_1 and B_2.

starting column of B_1, our layout fixer wraps B_2 in a horizontal operator, using an empty box (a box E containing the empty string), setting hs as the number of spaces necessary to align the two boxes. For the case where B_2 starts at a column further to the right than the left-most column where B_1 starts, the layout fixer uses a combination of a z-operator and a horizontal operator to skip to the next line, adding the indentation necessary to align both boxes. Both scenarios are illustrated in Figure 5. Note that empty boxes allow indenting other boxes (using the horizontal operator) or moving them to a new line (using the z-operator).

The same strategy can be used for the layout declarations `indent x y`, and `newline-indent x y`, setting the horizontal box such that the boxes are not aligned, but that the left-most column of B_2 is to the right of the left-most column of B_1, enforcing a z-operator whenever it is necessary to print the text into another line.

For offside declarations, we apply a slightly different approach. Because an offside declaration requires that the boxes in the subsequent lines should be further to the right than the column where the structure starts, we verify the operands of the z-operator. That is, for all boxes that move to a newline due to a z-operator and violate the offside rule, we use horizontal composition with an empty box to indent them such that the offside rule is satisfied.

We apply these strategies in a top-down traversal of the boxes that represent the original program. This approach produced satisfactory results when considering the Haskell programs in our benchmark as we will discuss in Section 6.2.

5.3 Layout Declarations for Pretty-printing

In this paper, we focus primarily on the correctness of a generated pretty-printer, but pretty-printing the program in a single line, adding newlines only to enforce layout declarations may not produce a *pretty*-printer. In layout-sensitive languages, concepts such as alignment, indentation and even the offside rule contribute to make the pretty-printed code prettier, i.e., more readable. However, these are *not sufficient* to determine a pretty layout. For example, consider the following production defining an if-else construct, with layout declarations to enforce the alignment of the *then* (T) and *else* (E) branches:

```
S.IfElse = "if" E "then" T:S "else" E:S
  {layout(align T E)}
```

A pretty-printed program using this production and the layout fixing algorithm looks like:

```
if e1 then s1 else
          if e2 then s2 else
                     s3
```

While this program is correct according to the layout declaration, one may say it is not pretty, as its layout may not make the program more readable, specially if we would consider writing programs with nested if-else statements.

The declarations from Section 3 are always enforced when parsing the program. However, for constructs that are not layout-sensitive, we could use a more flexible approach, using declarations only to produce better pretty-printers. Thus, we introduce *pretty-printing layout declarations*, which are similar to the previous ones, but are used *only* for pretty-printing. Layout declarations for pretty-printing start with the prefix `pp-`, and are ignored by the parser.

With pretty-printing layout declarations, the language designer can generate prettier pretty-printers, but still allow flexible layout when parsing the program. For example, consider the same production as the one shown previously, with additional pretty-printing layout declarations:

```
S.IfElse = "if" E "then" T:S "else" E:S
  {layout(pp-newline-indent "if" T && pp-align
  "if" "else" && align T E)}
```

Applying the pretty-printer generated from this production into the same program, produces:

```
if e1 then
 s1
else
 if e2 then
  s2
 else
  s3
```

Note that the pretty-printed program using only the `align` declaration would also be accepted by the same parser defined by the production above, since the additional layout declarations are used only for pretty-printing.

To provide more flexibility to language designers regarding indentation sizes and newlines, we also introduce the layout declaration `pp-newline-indent-by(x)` and `pp-newline(x)`. The declaration `pp-newline-indent-by(x)` is a variation of the declaration `pp-newline-indent`, such that it is possible to specify the number of spaces (using the integer `x`) that pretty-printer must consider when indenting the program. The declaration `pp-newline(x) t`, on the other hand, enforces that the tree `t` starts on a newline, indented by `x` spaces from the enclosing indentation.[3]

For instance, if instead we use the layout declaration `layout(pp-newline(1) T && pp-newline "else")` on the same production, it is possible to construct a pretty-printer that produces the following program:

```
if e1 then
 s1
else if e2 then
 s2
else s3
```

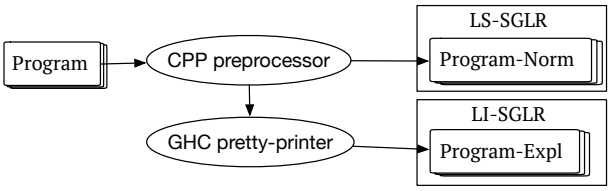
6 Evaluation

In this section we evaluate our approach for generating a parser and a pretty-printer from a grammar containing layout declarations. We are interested in answering the following research questions.

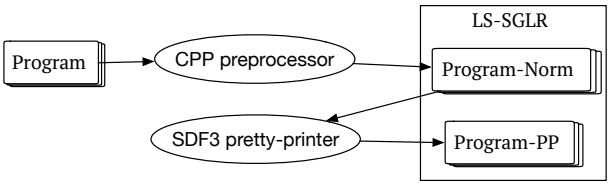
- RQ1 How parse-time disambiguation of ambiguities due to layout affects the performance of a generalized parser?
- RQ2 What is the accuracy of our layout fixer when pretty printing files of a layout-sensitive language?
- RQ3 How easy is it to specify a layout-sensitive language?

In order to answer the these research questions, we generate a parser and a pretty-printer derived from a declarative specification for Haskell containing layout declarations. We apply both generated parser and pretty-printer to 22191 Haskell programs from the Hackage[4] repository, using the benchmark described in [11]. We used the files in the same benchmark to provide a fair comparison between the performance of our parser and their implementation.

In order to measure the performance overhead of the layout-sensitive parser, we use a pretty-printer tool, part of the `haskell-src-exts` package[5], which has an option to pretty-print programs using only explicit grouping (brackets and semicolons). We also preprocess files using the C preprocessor part of the Glasgow Haskell Compiler (GHC) supporting additional extensions to increase the coverage



(a) Evaluating the performance of the parser.



(b) Evaluating the correctness of the pretty-printer.

Figure 6. Evaluation setup.

of files. The diagram in Figure 6a illustrates the process we adopted.

To measure the performance of our layout-sensitive parser (`LS-SGLR`) on the original program, we first apply the C preprocessor, applying the parser to the `Program-Norm` file. Similarly, we measure the performance of an implementation of SGLR without support for layout-sensitive disambiguation (`LI-SGLR`) on a program that contains brackets and semicolons to explicitly delimit structures (`Program-Expl`), using the pretty-printer from the `haskell-src-exts` package. We then compare the performance of both parsers to verify the overhead of using the layout-sensitive features of our implementation.

To measure the correctness of the pretty-printer generated using our approach, we use the process described in Figure 6b. First, we preprocess the file using the C preprocessor, generating the file `Program-Norm`. Next, we parse this file and pretty-print its abstract syntax tree using our pretty-printer to generate a new program `Program-PP`. Finally, we parse this file comparing its tree with the tree originated from the file `Program-Norm`.

We measure how easy it is to specify a layout-sensitive language by counting the total number of layout declarations used in the grammar.

6.1 Experimental Setup

We executed the benchmarks on a computer with 16GB RAM, and an Intel Core i7 CPU with a base frequency of 2.7GHz and a 6MB Last-Level Cache. The software consisted of Apple's macOS version 10.13.5 (17F77) and Oracle's Java Virtual Machine version 1.8.0_102.

We measured the execution time of batch-parsing the corpus of Haskell programs using the Java Microbenchmarking Harness (JMH), which is a framework to overcome the

[3] If `x = 0`, the declaration `pp-newline` can be used instead.
[4] http://hackage.haskell.org
[5] http://hackage.haskell.org/package/haskell-src-exts

Table 1. Benchmark results when executing our LS-SGLR parser on programs containing their original layout, and the LI-SGLR parser on programs containing explicit layout.

Parser	Data Set	Time (seconds)	Overhead
`LS-SGLR`	`Program-Norm`	638.05 ± 1.96	1, 72x
`LI-SGLR`	`Program-Expl`	370.26 ± 0.68	—

Table 2. Benchmark results when considering a subset of 14830 programs that do not have longest-match ambiguities.

Parser	Data Set	Time (seconds)	Overhead
`LS-SGLR`	`Program-Norm`	239.79 ± 0.90	1.53x
`LI-SGLR`	`Program-Expl`	156.37 ± 0.56	—

pitfalls of (micro-)benchmarking. When executing the benchmarks, we disabled background processes as much as possible, fixing the virtual machine heap size to 8 GB. We configured JMH to perform 5 warmup iterations, and 10 measurements, calculating the average time of each execution. We use the same settings to test the correctness of the pretty-printer, however, instead of using JMH, we simply compare Java objects corresponding to the abstract syntax tree of the programs `Program-Norm` and `Program-PP`.

6.2 Experiment Results

Performance of the Parser Table 1 shows the results of the *parse-time* of the LS-SGLR parser on programs with original layout, and the original SGLR parser on programs with explicit layout. Overall, we measured the overhead of our layout-sensitive parser to be 1.72x. This compares to 1.80x for Erdweg et al.'s implementation. Because Haskell programs may still require an additional post-parse disambiguation to disambiguate longest-match constructs [10, 11], we suspect that part of this overhead is caused by this additional disambiguation step, since programs with explicit layout do not present such ambiguities. For this reason, we also ran the same experiment on programs that *do not* contain longest-match ambiguities (14830 programs), measuring the overhead of disambiguating only ambiguities due to layout. As shown in Table 2, for such programs our parser presented an overhead of 1.5x.

Correctness of the pretty-printer When executing our pretty-printer, we verified that only 5 out of 22191 programs produced incorrect results (0.02 %). Because of the low number of cases, we investigated these programs manually and verified that because we apply our layout-fixer using a top-down traversal, a ripple effect when fixing a declaration may disrupt parts of the program that have been previously fixed.

Language specification The SDF3 grammar for Haskell used in our experiments contains 473 productions. It was necessary to annotate 34 productions to specify the indentation rules for Haskell. In total, we added 43 layout declarations, being 10 `offside`, 1 `align`, 5 `align-list`, 19 `indent`, and 8 `ignore-layout` declarations. Note that some productions required multiple declarations.

6.3 Threats to Validity

A threat to external validity, with respect to the generality of our results, is that we used only Haskell in our benchmarks. Despite not being able to generalize our results beyond Haskell programs, we believe that Haskell has indentation rules that are similar to other layout-sensitive languages. We have also tried our approach on a syntax definition for a subset of Python. However, because we do not cover the entire language, we could not parse many real-world programs, and decided to not include it in our benchmarks.

Another threat to the validity of our results concerns the correctness of our parser. To tackle this issue, we verified that the abstract syntax trees we obtained from our parser and the trees from the implementation done by Erdweg et al. were equal. Erdweg et al. checked the correctness of their parser by comparing it with to the parser from GHC. Since they obtained positive results from that comparison, we believe that our parser is also correct.

7 Related Work

In this section, we highlight previous work on layout-sensitive parsers and generating pretty-printers from a declarative specification, discussing how prior work inspired us.

7.1 Layout-Sensitive Parsing

As we mentioned previously, our approach to derive a layout-sensitive parser from a declarative specification was inspired by the work by Erdweg et al. [11]. Their parser performs post-parse disambiguation to avoid splitting parse states that were already merged when finding an ambiguity, which would degrade the performance of the parser. Our parser prevents such ambiguities to be constructed by filtering trees at parse time using the propagated information about token selectors. This change improves the performance of the parser by avoiding the post-parse disambiguation step.

Indentation-sensitive context-free grammars (IS-CFGs) [1], can be used to generate LR(k) or GLR layout-sensitive parsers. In IS-CFGs each terminal is annotated with the column at which it occurs in the source code, i.e., its indentation, and each non-terminal is annotated with the minimum column at which it can occur. To express alignment of constructs, an IS-CFG requires additional productions, which are generated automatically from certain non-terminals. We opted for not modifying the original grammar, only requiring that productions are annotated with layout declarations. While our approach is based on a scannerless generalized parser, we obtained similar performance results to a layout-sensitive LR(k) parser generated from an IS-CFG when considering Haskell programs with longest-match ambiguities. Finally, it is not clear how to automatically derive a pretty-printer from

an IS-CFGs, whereas we provided a mechanism to derive a pretty-printer from a specification with layout declarations.

Afroozeh and Izmaylova [2] use data-dependent grammars [14] to generate a layout-sensitive parser. They propose high-level declarations such as `align` and `offside` that are translated into equations, which are evaluated during the execution of a generalized LL parser. In our work, we opted to leave the grammar intact and have layout declarations as annotations on productions. In contrast, their declarations are intermingled with the non-terminals in productions, which decreases readability. Finally, their approach also requires propagating data "upwards" and "downwards" when building tree nodes, whereas we propagate data only upwards.

Brunauer and Mühlbacher [6] propose another approach to declaratively specify layout-sensitive languages using a scannerless parser. They modify the non-terminals of the grammar to include integers as parameters, which are mixed with the grammar productions to indicate the number of spaces that must occur within certain productions. However, these changes have a detrimental effect on the readability and on the size of the resulting grammar. We opted to abstract over details such as number of spaces, columns, and lines, by using high-level layout declarations.

7.2 Pretty-printing

Many solutions have been proposed to integrate the specification of a parser and a pretty-printer [5, 18, 24**?**]. However, none of these solutions is aimed at generating layout-sensitive parsers *and* pretty-printers using the same specification. For instance, the syntax definition formalism SDF3 [24] allows the specification of a parser and a default pretty-printer to be combined by using *template productions*. Template productions are similar to regular productions, but the indentation inside the template is considered only when pretty-printing the program. Thus, they are similar to our layout declarations for pretty-printing as they do not enforce any restriction with respect to layout while parsing. However, when using templates in combination with layout declarations to generate layout-sensitive parsers, any inconsistency between the templates and the declarations might result in an incorrect pretty-printer.

Different approaches have been proposed to derive prettier [26], and correct-by-construction [7] pretty-printers. However, these approaches are aimed at traditional programming languages, and might require further adaptations to be applied to layout-sensitive languages. Finally, none of these approaches allow a specification of the pretty-printer that can be derived from the context-free grammar. We use layout declarations as annotations to context-free productions to indicate how structures should be pretty-printed such that the pretty-printed program obeys the indentation rules of the language. Furthermore, our pretty-printing layout declarations enable customizing the generated pretty-printer such that it also produces prettier results.

8 Future Work

As future work we plan to apply our techniques to more layout-sensitive languages, examining their indentation rules to observe how our generated parser and pretty-printer behave in other scenarios. We also would like to investigate different strategies to apply our layout fixer, finding alternatives that do not cause a ripple effect when applying (pretty-printing) layout declarations, as it may produce incorrect results. Furthermore, we would like to study the integration between our pretty-printing layout declarations and other syntax definition formalisms that enable declarative specification of both parser and pretty-printer, such as SDF3.

Another aspect to consider is preservation of comments when pretty-printing layout-sensitive programs. Currently, our pretty-printer discards comments altogether, but ideally, comments should be preserved while maintaining the correctness of the pretty-printer. Preservation of comments in transformations is challenging even for traditional languages, and most approaches rely on heuristics [8, 17].

Finally, we propose a more in-depth analysis of SGLR mechanisms to disambiguate longest-match constructs. As shown by our experiment, such ambiguities are responsible for a considerable fraction of the overhead of our parser for Haskell. It would also be interesting to study how layout-sensitive and longest-match disambiguation are related to operator precedence disambiguation [3, 9].

9 Conclusion

In this paper, we presented an approach to support declarative specifications of layout-sensitive languages. We tackled the main issues that prevent the adoption of these languages in tools such as language workbenches: usability, performance and tool support. We introduced layout declarations, providing language designers with a high-level specification language to declare indentation rules of layout-sensitive languages. Furthermore, we described a more efficient implementation of a scannerless layout-sensitive generalized LR parser based on layout declarations. Finally, we presented strategies to derive a correct pretty-printer, which produced the correct result for almost all of the programs in our benchmark. Overall, we believe that our work can be used to facilitate the development and prototyping of layout-sensitive languages using tools such as language workbenches.

Acknowledgments

The work presented in this paper was partially funded by CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil) and by the NWO VICI Language Designer's Workbench project (639.023.206). We would also like to thank the anonymous reviewers for their feedback.

References

[1] Michael D. Adams. 2013. Principled parsing for indentation-sensitive languages: revisiting landin's offside rule. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 511–522. https://doi.org/10.1145/2429069.2429129

[2] Ali Afroozeh and Anastasia Izmaylova. 2015. One parser to rule them all. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Gail C. Murphy and Guy L. Steele Jr. (Eds.). ACM, 151–170. https://doi.org/10.1145/2814228.2814242

[3] Ali Afroozeh and Anastasia Izmaylova. 2016. Operator precedence for data-dependent grammars. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 13–24. https://doi.org/10.1145/2847538.2847540

[4] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. 2009. YAML Ain't Markup Language, Version 1.2. Available on: http://yaml.org/spec/1.2/spec.html.

[5] R. Boulton. 1996. *Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing.* Number 390. University of Cambridge, Computer Laboratory.

[6] Leonhard Brunauer and Bernhard Mühlbacher. July, 2006. Indentation Sensitive Languages. (July, 2006). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.2933&rep=rep1&type=pdf Unpublished Manuscript.

[7] Nils Anders Danielsson. 2013. Correct-by-construction pretty-printing. In *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013, Boston, Massachusetts, USA, September 24, 2013*, Stephanie Weirich (Ed.). ACM, 1–12. https://doi.org/10.1145/2502409.2502410

[8] Maartje de Jonge and Eelco Visser. 2011. An Algorithm for Layout Preservation in Refactoring Transformations. In *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers (Lecture Notes in Computer Science)*, Anthony M. Sloane and Uwe Aßmann (Eds.), Vol. 6940. Springer, 40–59. https://doi.org/10.1007/978-3-642-28830-2_3

[9] Luís Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. 2018. Towards Zero-Overhead Disambiguation of Deep Priority Conflicts. *Programming Journal* 2 (2018), 13.

[10] Simon Marlow (editor). 2010. Haskell 2010 Language Report. Available on: https://www.haskell.org/onlinereport/haskell2010.

[11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2012. Layout-Sensitive Generalized Parsing. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Krzysztof Czarnecki and Görel Hedin (Eds.), Vol. 7745. Springer, 244–263. https://doi.org/10.1007/978-3-642-36089-3_14

[12] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? https://doi.org/articles/languageWorkbench.html

[13] John Gruber. 2004. Markdown: Syntax. Available on: https://daringfireball.net/projects/markdown/syntax.

[14] Trevor Jim, Yitzhak Mandelbaum, and David Walker. 2010. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 417–430. https://doi.org/10.1145/1706299.1706347

[15] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2010. Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 918–932. https://doi.org/10.1145/1869459.1869535

[16] Peter J. Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (1966), 157–166. https://doi.org/10.1145/365230.365257

[17] Huiqing Li, Simon Thompson, and Claus Reinke. 2005. The Haskell Refactorer, HaRe, and its API. *Electronic Notes in Theoretical Computer Science* 141, 4 (2005), 29–34. https://doi.org/10.1016/j.entcs.2005.02.053

[18] Lisa F. Rubin. 1983. Syntax-Directed Pretty Printing - A First Step Towards a Syntax-Directed Editor. *IEEE Trans. Software Eng.* 9, 2 (1983), 119–127.

[19] M.G.J. van den Brand. July, 1993. *Generation of Language Independent Modular Prettyprinters.* Technical Report P9315. University of Amsterdam.

[20] M.G.J. van den Brand. October, 1993. *Prettyprinting Without Losing Comments.* Technical Report P9327. University of Amsterdam.

[21] Guido van Rossum and Fred L. Drake. 2011. *The Python Language Reference Manual.* Network Theory Ltd.

[22] Eelco Visser. 1997. A Case Study in Optimizing Parsing Schemata by Disambiguation Filters. In *International Workshop on Parsing Technology (IWPT 1997).* Massachusetts Institute of Technology, Boston, USA, 210–224.

[23] Eelco Visser. 1997. *Scannerless Generalized-LR Parsing.* Technical Report P9707. Programming Research Group, University of Amsterdam.

[24] Tobi Vollebregt. 2012. *Declarative Specification of Template-Based Textual Editors.* Master's thesis. Delft University of Technology, Delft, The Netherlands. Advisor(s) Eelco Visser and Lennart C. L. Kats. https://doi.org/uuid:8907468c-b102-4a35-aa84-d49bb2110541

[25] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. 2012. Declarative specification of template-based textual editors. In *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, Anthony Sloane and Suzana Andova (Eds.). ACM, 1–7. https://doi.org/10.1145/2427048.2427056

[26] Philip Wadler. 1998. A Prettier Printer. In *Journal of Functional Programming*. Palgrave Macmillan, 223–244.