Layout Syntax Support in the BNF Converter

A Declarative Approach to Parsing Indentation-Sensitive Language with Mainstream Parser Generators

Master's thesis in Computer Science and Engineering

Beata Burreau

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Layout Syntax Support in the BNF Converter

A Declarative Approach to Parsing Indentation-Sensitive Language with Mainstream Parser Generators

Beata Burreau

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Layout Syntax Support in the BNF Converter
A Declarative Approach to Parsing Indentation-Sensitive Language with Mainstream Parser Generators
Beata Burreau

Supervisor: Andreas Abel, Computer Science and Engineering
Examiner: Aarne Ranta, Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Division of Computing Science
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Layout Syntax Support in the BNF Converter
A Declarative Approach to Parsing Indentation-Sensitive Language with Mainstream Parser Generators

Beata Burreau
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Many programming languages, such as Haskell and Python, use layout as part of their syntax. We can expect future programming languages to also be layout-sensitive. Therefore, the toolchains for implementing programming languages must support layout-sensitive languages.

This thesis presents a declarative approach to describing layout-sensitive languages and parsing programs written in them. We reserve the terminals `newline`, `indent`, and `dedent` for describing layout syntax in BNF grammar and provide an algorithm for representing the layout of a program with these terminals, before parsing it. By verbalising layout syntax this way, mainstream parser generators, and their parsing algorithms, can be used. This approach is successfully implemented in BNF Converter (BNFC), a tool that generates a compiler front-end from a context-free grammar in Labelled BNF (LBNF) form. With a special kind of LBNF rule, called pragma, it is possible to declare global layout syntax rules, such as the offside rule, which affects the insertion of layout terminals by the aforementioned algorithm. The reserved terminals and the pragmas can together describe popular layout syntax. Furthermore, both purely layout-sensitive languages and those mixing layout-sensitive and insensitive syntax are describable in LBNF.

Keywords: layout syntax, indentation-sensitive language, context-free grammar, Labelled BNF (LBNF), BNF Converter (BNFC), Pfenning safety

Acknowledgements

I give my deepest thanks to my supervisor Andreas Abel for his expertise, guidance, and engagement in this project. It has been a pleasure to work under his supervision. I would also like to thank my friends Matilda Blomquist and Henry Collett for their contributions, Matilda for her insightful questions and good company throughout the project, and Henry for his English skills and writing advice. Lastly, I would like to thank my family and friends for their support during all of my education, which paved the way for this thesis.

Beata Burreau, Gothenburg, 2023-05-10

Contents

1 Introduction

Many programming languages use layout, namely indentation and line breaks, for code structure instead of, or as a complement to, the traditional curly brackets and semicolons. Layout is thus part of their syntax. The idea of structuring code by indentation was introduced by Landin [1] with his language ISWIM; it has inspired a family of layout-sensitive languages, including Python, Haskell, Agda, F# and YAML. Layout syntax is an integral part of many existing programming languages, so we can expect future programming languages to also be layout-sensitive. Therefore, our toolchains for implementing them must support layout syntax. One such tool is the BNF Converter (BNFC), a compiler construction tool that can generate a lexer and a parser from a context-free grammar in Labelled BNF (LBNF) form. LBNF allows describing some standard layout syntax, but it does not offer enough expressivity to, for instance, describe purely layout-sensitive languages like Python.

We set out to enhance BNFC's support for layout syntax. We reserve terminals `newline`, `indent`, and `dedent` for representing layout in LBNF and introduce a set of pragmas, a kind of rule in LBNF, for expressing global layout rules. Together, they allow describing common layout syntax rules and purely layout-sensitive languages.

2 Problem

The BNF Converter already offers some support for layout syntax, but its expressivity could be improved. Grouping of program elements, such as statements, by indentation can be expressed in LBNF. Take, for instance, Python's function definitions. A function definition is composed of a header and a body. The header has the form `def` *name* `(` *parameters* `):` and the body is a group of statements:

```
def f():
    print("hello")
    print("world")
```

The statements must be indented relative to `def` and vertically aligned, expressed in LBNF using a so-called layout pragma. The pragma `layout ":"` in Figure 2.1 below declares that the list of statements following the colon must be indented and vertically aligned *if* curly brackets and semicolons are omitted.

```
layout ":"                                                        ;
CS. CompoundStmt ::= "def" Ident "(" [Ident] ")" ":" "{" [Stmt] "}" ;
separator Stmt ";"                                                ;

SPrint. Stmt     ::= "print" "(" String ")"                       ;
...
```

Figure 2.1: Excerpt of an LBNF grammar describing Python-like function definition with the `layout` pragma

Python's function definitions are valid programs in the language described by the above grammar, but so are the following two definitions.

```
def f(): print("hello")
         print("world")
```

```
def f(): {
    print("hello");
    print("world"); }
```

The left definition is invalid in Python because the function body begins on the same line as the header. In Python, multiline bodies must begin on a line below the header, but such line break requirements are not expressible in LBNF. The right definition is invalid in Python because curly brackets and semicolons structure the body instead of indentation and line breaks. Python offers no alternative to indentation for structuring function definitions [2]. However, LBNF's `layout` pragma requires the terminal, such as the colon, to be followed by a list enclosed in curly braces and separated by semicolons [3]. These are some limitations of the current layout syntax formalism but not the only ones. Another example is that the offside rule is enforced by default. The offside rule allows program elements, such as expressions, to span multiple lines using indentation and, for instance, makes the below three expressions equivalent.

```
1 + 1        1 +        1
               1          + 1
```

Furthermore, terminals marking the beginning and end of a group of program elements, such as Haskell's `let` and `in`, are independently declared. Consequently, an end terminal will mark the end of any group not only those begun by the intended start terminal.

We aim to offer comprehensive support for layout-sensitive languages in BNFC by sensibly extending, and if necessary modifying, BNFC's current layout syntax formalism. In the process, we should answer the following two questions.

- How can layout syntax be described in LBNF in a simple yet expressive manner?
- How can layout syntax, described in LBNF, be verified in compiler front-ends generated by BNFC?

LBNF is a context-free grammar formalism; description of layout syntax in context-free grammar, and other forms of formal grammar, is an ongoing research area [3]–[6]. Thence, there is no industry standard in place. The lack of a standard offers some language design freedom, but in that freedom lies a challenge in designing a simple yet expressive formalism.

BNFC uses mainstream LR parser generators, such as Bison, CUP, and Happy [7], to generate LALR(1) parsers from LBNF grammars. Consequently, the parsing algorithms implemented by these generators dictate the verification of layout syntax in the BNFC-generated compiler front-ends. Because we must rely on existing parsing algorithms, Adams' extension to CFG for describing layout syntax and the derived parsing algorithms [4] cannot be incorporated in BNFC for enhanced layout syntax support.

BNFC can generate compiler front-ends in Haskell, Agda, C, C++, Java, and OCaml. The offer of comprehensive support for layout syntax is limited to the front-ends generated in Haskell, but portability to the other languages is kept in mind since they should offer the same support in the future.

3 Background

A programming language has rules for how elements from its alphabet can be combined to form sentences, just like a natural language does. These rules constitute the language's *syntax* and are described by its *grammar*. Context-free grammar (CFG) is a formal grammar for describing programming languages' syntax. A standard notation for CFG is Backus-Naur form (BNF), and Labelled BNF (LBNF) is a variant of BNF used by the BNF Converter (BNFC). Layout syntax is not describable in BNF, but some layout syntax can be described in LBNF using layout pragmas.

3.1 Context-Free Grammar

A language is a set of sentences derivable from its grammar. Sentences are sequences of letters from the alphabet and have a recursive structure; a sentence consists of phrases, letters or both, the phrases consist of smaller phrases, letters or both, and eventually only letters. Languages can therefore be defined recursively, and context-free grammar is a formal notation for doing just that.

A context-free grammar G is defined by a four-tuple (N, T, P, S) as in [8, p. 173]:

N is a finite set of *nonterminals* such that each represents a language.

T is a finite set of letters, formally *terminals*, and is the defined language's alphabet.

P is a finite relation in $N \times (N \cup T)^*$ of *productions* and constitutes the recursive language definition.

A production is a pair $(A, \alpha) \in P$, often written $A \to \alpha$ and referred to as "the production for A". $A \in N$ is the nonterminal (partially) defined by the production, and $\alpha \in (N \cup T)^*$ a possibly empty sequence of terminals and nonterminals. α defines one way to form sentences in the language represented by A.

There can be more than one production for a particular A. The productions $A \to \alpha_1, A \to \alpha_2, ..., A \to \alpha_n$ can then be written $A \to \alpha_1 \,|\, \alpha_2 \,|\, ... \,|\, \alpha_n$.

S is an element in N called *start symbol.* It represents the language defined by the grammar. The other elements in N represent auxiliary languages that help define the language represented by S.

By convention the productions for the start symbol are listed first in the grammar [9, p. 197].

Let α, β and γ be sequences of terminals and nonterminals in $(N \cup T)^*$.

We say that $\alpha\, A\, \beta$ yields $\alpha\, \gamma\, \beta$, written $\alpha\, A\, \beta \Rightarrow \alpha\, \gamma\, \beta$, if there exists a production $A \to \gamma$ in P.

We say that α derives β, written $\alpha \overset{*}{\Rightarrow} \beta$ if $\alpha = \beta$ or there exists a sequence $\alpha_1, \alpha_2, ..., \alpha_k$, $k \geq 0$ such that $\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_k \Rightarrow \beta$ [10, p. 104].

The *language* of a grammar $G = (N, T, P, S)$ is the set $L(G)$ of sentences derivable from G [8, p. 179]:

$$L(G) = \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$$

A *sentence* is thus a sequence of terminals derivable from S.

An auxiliary language of a grammar G is the set $L_G(n)$ of sentences derivable from a nonterminal $n \in N \setminus \{S\}$:

$$L_G(n) = \{w \in T^* \mid n \overset{*}{\Rightarrow} w\}$$

A sentence in $L(G)$ comprises sentences from the auxiliary languages. These auxiliary sentences have no established name in programming language theory, in linguistics, however, they are called *constituents* [11, p. 26, 30] and we adopt this term:

A *constituent* is a sequence of terminals derivable from $n \in N \setminus \{S\}$.

To exemplify, Figure 3.1 below shows a context-free grammar for boolean expressions. A boolean variable is a literal, for instance a, with two possible values: `true` or `false`. A boolean expression comprises boolean variables, possibly negated by $\neg$, joined by $\wedge$ and $\vee$. Three nonterminals are needed to capture the precedence relation between $\neg$, $\wedge$ and $\vee$. The grammar is formally stated as

$$G = (\{exp,\ term,\ factor,\ var\},\ \{\vee,\ \wedge,\ \neg,\ \texttt{(},\ \texttt{)}\},\ P,\ exp)$$

P is the set of productions listed in Figure 3.1. The start symbol *exp* represents the language of boolean expressions. Boolean expressions are sentences in the language, and terms, factors, and variables are constituents.

$$
\begin{aligned}
\textit{exp} &\rightarrow \textit{term} \\
&\mid\ \textit{term} \lor \textit{term} \\[1em]
\textit{term} &\rightarrow \textit{factor} \\
&\mid\ \textit{factor} \land \textit{factor} \\[1em]
\textit{factor} &\rightarrow \textit{var} \\
\textit{factor} &\rightarrow \neg\, \textit{factor} \\
\textit{factor} &\rightarrow \texttt{(}\ \textit{exp}\ \texttt{)}
\end{aligned}
$$

Figure 3.1: A context-free grammar for boolean expressions

From the grammar, one can, for instance, deduce that $\neg\ a \land b$ is a valid boolean expression:

1. $\neg\ a$ is a *factor* since a is one.
2. $\neg\ a \land b$ is a *term* since $\neg\ a$ and b are *factors*.
3. $\neg\ a \land b$ is an *exp* since it is a *term*.

Backus-Naur form (BNF) is a notation for context-free grammar and is often used synonymously with it when defining programming languages. Productions are written A `::=` α`;`, terminals enclosed in quotation marks and nonterminals in `<>` [12, p. 281].

3.2 Labelled BNF and the BNF Converter

Labelled BNF (LBNF) is a BNF grammar where every rule is labelled; productions are written on the form l`.` A `::=` α where l is the label. Given an LBNF grammar G, the BNF Converter generates a compiler front-end for the language G describes.

The context-free grammar for boolean expressions can be expressed in LBNF as in Figure 3.2. The productions' unique labels, followed by dots, are listed in the leftmost column.

```
E.       Exp    ::= Term ;
EOr.     Exp    ::= Term "∨" Term ;

T.       Term   ::= Factor ;
TAnd.    Term   ::= Factor "∧" Factor ;

F.       Factor ::= Ident ;
FNot.    Factor ::= "¬" Factor ;
FPar.    Factor ::= "(" Exp ")" ;
```

Figure 3.2: An LBNF grammar for boolean expressions

From an LBNF grammar G, the BNFC tool can generate a compiler-front end for $L(G)$. The compiler front-end, including the lexer and parser, is implemented in an existing programming language called the *target language*. BNFC is a language-agnostic tool that can generate front-ends in Haskell, Agda, C, C++, Java, and OCaml. The tool uses existing lexer and parser generators for the target language, such as Bison and Yacc for C, C++ and Java. As depicted in Figure 3.3, given an LBNF grammar BNFC produces specification files for the lexer and parser generators, which produce a lexer and parser implemented in the target language. Among other things, BNFC also produces an abstract syntax implementation in the target language [13]. The BNFC tool itself is implemented in Haskell.

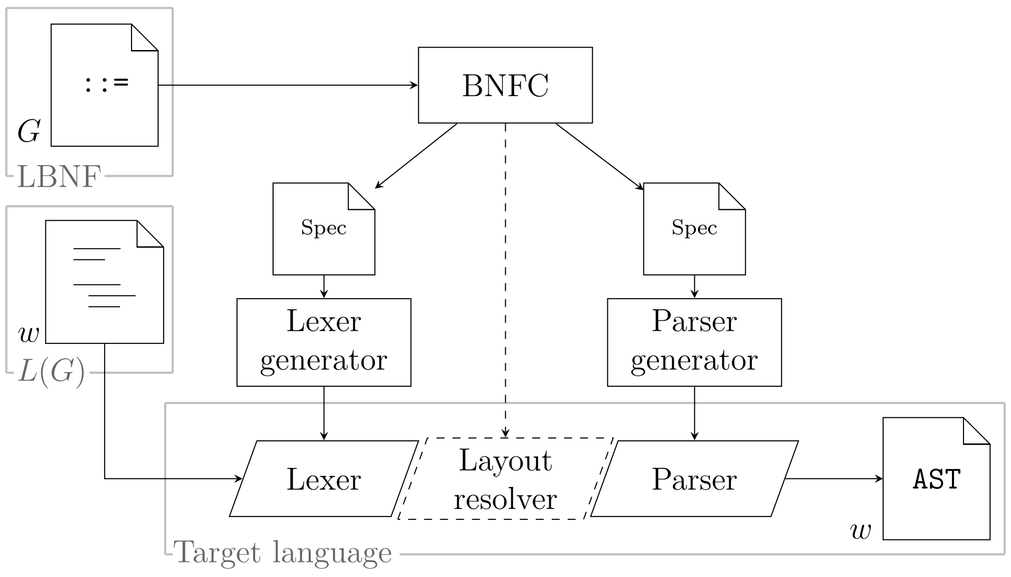


Figure 3.3: The BNFC toolchain

Say BNFC is used to generate a compiler front-end in Haskell from the grammar for boolean expressions in Figure 3.2. Then give the compiler front-end a boolean expression w. If $w \in L(G)$, the compiler front-end will parse the expression and output an abstract syntax tree (AST) representation of

w. Otherwise, it will produce an error. For instance, $\neg\ a \land b$ is parsed to `E (TAnd (FNot (F (Ident "a"))) (F (Ident "b")))`, while $\neg \land b$ yields an error.

Besides regular productions, LBNF has rules called pragmas and macros. A *pragma* instructs BNFC to treat specific terminals or nonterminals differently. For instance, it is possible to specify multiple start symbols using the `entrypoint` pragma. A *macro* is a short-hand notation for a set of productions. For example, rules for translation between precedence levels can be represented with the `coercions` macro. The tool supports some layout syntax [14]; languages with Haskell-like layout syntax can be described using layout pragmas. If a layout pragma is used in an LBNF grammar, BNFC will generate a layout resolver. The resolver translates layout syntax into explicit syntax by inserting `{`, `}` and `;` tokens into the token stream from the lexer before forwarding it to the parser.

4

Layout Syntax

Layout-sensitive languages have rules for line breaks and indentation; they have layout syntax. There are commonalities in existing languages' layout syntax, and we here introduce some common rules and their interplay. A less established notion is the stability, or rather instability, of layout-sensitive languages under the renaming of identifiers, here introduced as Pfenning safety. The layout rules and Pfenning safety are exemplified with the languages Haskell and Python.

4.1 Layout Syntax Rules

Layout syntax rules pose restrictions on the layout of individual sentences and constituents (unary rules) or on multiple constituents (n-ary rules); some standard rules are the single-line rule (unary), the offside rule (unary), line joining (unary), vertical alignment (n-ary), and indentation (n-ary).

4.1.1 The Single-Line Rule

The single-line rule [6] restricts a constituent to one line; it requires that any character in a constituent occurs on the line where the constituent starts. The single-line rule requires constituents to have the following shape:

4.1.2 The Offside Rule

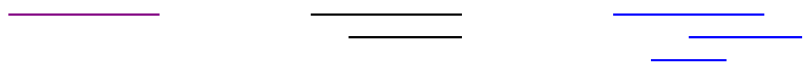
The offside rule allows a constituent to span multiple lines if the succeeding lines are indented relative to the first line. The offside rule, and the concept of using indentation for program structure, was introduced by Landin in his 1966 paper "The Next 700 Programming Languages". Below is Landin's original formulation of the rule followed by a more recent reformulation.

> The southeast quadrant that contains the [constituent]'s first symbol must contain the entire [constituent], except possibly for bracketed subsegments.
>
> Landin, 1966 [1]

[The offside] 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.

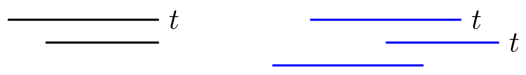
Amorim et al, 2018 [15]

Constituents adhering to the offside rule can, for instance, have the following shapes.



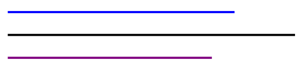
4.1.3 Line Joining

Line joining [16] by a reserved terminal *t* allows a constituent to span multiple lines if *t* is the last terminal on all lines except the last one. The succeeding lines can have any indentation, and *t* is not part of the constituent. Line joining terminals allow constituents to, for instance, have the following shapes.

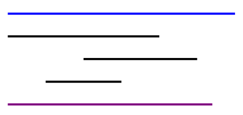


4.1.4 Vertical Alignment

Vertical alignment of a group of constituents restricts them to be aligned at the column. That is, all constituents in the group start in the same column. A vertically aligned constituent group can, for instance, have the following shape where colours distinguish the constituents.

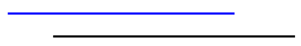


The following is also a vertically aligned group if constituents are allowed to span multiple lines, for instance, by adhering to the offside rule.

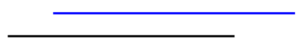


4.1.5 Indentation and Dedentation

Somewhat opposite to alignment, indentation can be enforced between constituents: an indented constituent must have its first terminal at a column to the right of the column of the first terminal of another constituent [15].

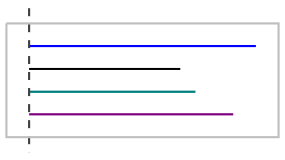


Similarly, a dedented constituent must have its first terminal at a column to the left of the column of the first terminal of another constituent.

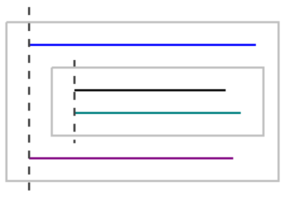


4.1.6 Interplay of Vertical Alignment and Indentation

A *block* of code is a vertically aligned group of zero or more constituents in a layout-sensitive context. The column of the first terminal in a code block is the column at which all constituents must be aligned. We will refer to this column as the block's *offside line* line and here illustrate it with a dashed line.



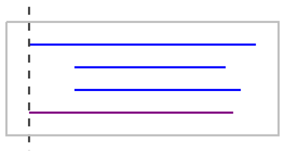
Enforcing indentation between two constituents in a block opens an inner block with an offside line to the right of the outer block's offside line. This is referred to as *block nesting*.



A dedented constituent, the purple one above, marks the end of the preceding block. Here, it starts at the offside line of the outer block and thus continues that block. If it instead were to start at a column in-between the two existing ones, it would still mark the end of the preceding block but also the beginning of a new block with this intermediate column as the offside line.

4.1.7 Interplay of Vertical Alignment, Indentation and the Offside Rule

Indentation plays a different role in a setting where constituents adhere to the offside rule. As illustrated by the second and third blue lines below, lines starting to the right of the current block's offside line are part of a constituent starting at the offside line. Consequently, block nesting can only be achieved by assigning block-introducing properties to specific terminals, such as`where`, `let`, `do`, and `of` in Haskell.



4.2 Pfenning Safety

An interesting property of layout-sensitive languages is whether they are stable under the renaming of an identifier preceding a layout-sensitive code block. Figure 4.1 below shows an example of Haskell's instability under α-renaming; the two print statements in `f` are vertically aligned and thus form a layout block, but renaming `f` to `fun` breaks the vertical alignment and, thereby, the parsing.

```
f :: IO ()
f = do print "hello"
       print "world"
```

```
fun :: IO ()
fun = do print "hello"
       print "world"
```

Figure 4.1: A valid Haskell function `f` and the resulting, invalid function `fun` after α-renaming the function identifier

The property can be referred to as *Pfenning safety* [17] and is defined as follows.

Definition 4.2.1 (Pfenning safety)**.** α-renaming an identifier in a valid program produces an equivalent, valid program

A layout-sensitive language $L(G)$ is called Pfenning-safe if all programs $w \in L(G)$ are so.

4.3 Examples

Python's and Haskell's layout syntax differ at various levels. Python is purely layout-sensitive, while Haskell has both layout-sensitive and insensitive syntax. Constituents can span multiple lines with a line joining terminal in Python and by the offside rule in Haskell. In Python, constituents are grouped by indentation alone, while Haskell has reserved terminals for grouping. Python appears to be Pfenning-safe, whereas Haskell is not.

4.3.1 Python

Python's statements and expressions adhere to the single-line rule but can span multiple lines if joined with a backslash, the line joining terminal. Expressions in round, square or curly brackets can span multiple lines [16]. The indentation of the succeeding lines is irrelevant. Strings within triple quotes, so-called documentation strings, can span multiple lines and are allowed any indentation relative to the first line [18].

Statements can only be grouped by indentation; there is no layout-insensitive alternative. Statements are grouped in compound statements, comprising a header and a statement group. A header includes one or more compound statement keywords, such as `if`, `for` and `in`, `with`, and `def` and ends with a colon which marks the beginning of the statement group. A group of statements can be listed on one or multiple lines. Listed on one line, they are separated by semicolons, and the first statement can begin on the same line as the header. Listed on multiple lines, they must be indented, vertically aligned, and begin on a line below the header [2].

$$\begin{array}{rcll}
\textit{compound\_stmt} & \rightarrow & \texttt{def}\ \textit{var}\ \texttt{(}\ \textit{params}\ \texttt{)}\ \texttt{:}\ \textit{stmts} & \\
 & | & \texttt{for}\ \textit{target}\ \texttt{in}\ \textit{exps}\ \texttt{:}\ \textit{stmts} & \\
 & | & \dots & \\
 & & & \\
\textit{stmts} & \rightarrow & \textit{stmts\_line}\ \texttt{newline} & \\
 & | & \texttt{newline}\ \texttt{indent}\ \textit{stmts\_block}\ \texttt{dedent} & \\
 & & & \\
\textit{stmts\_block} & \rightarrow & \textit{stmts\_line}\ \texttt{newline} & \\
 & | & \textit{stmts\_line}\ \texttt{newline}\ \textit{stmts\_block} & \\
 & | & \textit{compound\_stmt} & \\
 & | & \textit{compound\_stmt}\ \textit{stmts\_block} & \\
 & & & \\
\textit{stmts\_line} & \rightarrow & \textit{stmt}_1\ ;\ \dots\ ;\ \textit{stmt}_n & (n \geq 1) \\
 & | & \textit{stmt}_1\ ;\ \dots\ ;\ \textit{stmt}_n\ ; & (n \geq 1)
\end{array}$$

Figure 4.2: Excerpt of a context-free grammar for Python's compound statements *for* and *def* [2]

Renamable identifiers appear in assignment statements, assignment expressions and some compound statement headers, such as class and function definitions. Assignment statements and assignment expressions are stable under renaming of their identifiers since multiline expressions must be bracketed and thus are indentation insensitive. Compound statements are Pfenning-safe thanks to the forced line break before multiline statement groups. Python is thus Pfenning-safe.

4.3.2 Haskell

Haskell's sentences and constituents, such as declarations and statements, adhere to the offside rule [19]. Constituents can be grouped in layout-sensitive or insensitive blocks using either indentation or curly brackets and semicolons, and the grouping mechanisms can be mixed and nested. Blocks appear in the Haskell constructs *where*, *let*, *do*, and *case*. Figure 4.3 shows their concrete syntax in layout-insensitive form; curly brackets surround the lists of declarations, statements, and case alternatives, and semicolons separate the list elements. Omitting the opening bracket after`where`, `let`, `do` or `of` means that the succeeding constituents form a layout-sensitive block and must be indented and vertically aligned.

$$
\begin{array}{rcll}
\mathit{rhs} & \rightarrow & \mathit{exp} & \\
 & | & \mathit{exp}\ \texttt{where}\ \mathit{decls} & \\
\mathit{exp} & \rightarrow & \texttt{let}\ \mathit{decls}\ \texttt{in}\ \mathit{exp} & \\
\mathit{decls} & \rightarrow & \texttt{\{}\ d_1\ ;\ \ldots\ ;\ d_n\ \texttt{\}} & (\mathrm{n} \geq 0) \\
 & & & \\
\mathit{exp} & \rightarrow & \texttt{do}\ \texttt{\{}\ \mathit{stmts}\ \texttt{\}} & \\
\mathit{stmts} & \rightarrow & \mathit{stmt_1}\ ;\ \ldots\ ;\ \mathit{stmt_n}\ ;\ \mathit{exp} & (\mathrm{n} \geq 0) \\
 & & & \\
\mathit{exp} & \rightarrow & \texttt{case}\ \mathit{exp}\ \texttt{of}\ \texttt{\{}\ \mathit{alts}\ \texttt{\}} & \\
\mathit{alts} & \rightarrow & \mathit{alt}_1\ ;\ \ldots\ ;\ \mathit{alt}_n & (\mathrm{n} \geq 1)
\end{array}
$$

Figure 4.3: Excerpt of a context-free grammar for Haskell's constructs *where*, *let*, *do* and *case* [20]

Haskell is not Pfenning-safe since renaming an identifier can break parsing, as illustrated in Section 4.2. Programming guidelines allude to this fact by telling Haskellers to "make sure renamings [do not] destroy the layout" [21].

5

Enhanced Layout Syntax Support in the BNF Converter

We reserve three terminals in LBNF for representing layout syntax: `newline`, `indent`, and `dedent`. Like before, a layout resolver is generated and plugged in between the lexer and the parser. The layout resolver inserts layout tokens, `newline`, `indent` and `dedent`, into the token stream from the lexer. When the stream reaches the parser, it is checked against the language's layout syntax as specified by its grammar. A set of layout pragmas express standard layout syntax rules, such as associating specific tokens with pre-defined layout rules and enforcing the offside rule globally, and affect the layout resolution process.

5.1 Reserved Terminals Describe Layout Syntax in LBNF

Line breaks and indentation are what constitute layout syntax. We, therefore, reserve `newline`, `indent`, and `dedent` for describing layout syntax in LBNF productions. They are the layout-syntactic equivalents `;`, `{` and `}`.

In a layout-sensitive context, a line break (\n) generally marks the end of a constituent. However, a constituent can span multiple physical lines if some layout rule allows it. The offside rule is one example of such a rule, and line joining terminals is another. For instance, the below expressions have different layout syntax, but other than that, they are syntactically equivalent because of the offside rule.

```
1 + 1       1 +       1
              1         + 1
```

Having one production for each expression in LBNF would bloat the grammar, and the only line break of real interest is that terminating the constituent. Therefore, we adopt Python's idea of logical lines [16]: a constituent can span one or more physical lines and a `newline` terminal marks the end of a constituent. It also marks the end of a physical line, just not *all* physical lines. Blank lines, i.e. physical lines containing only spaces, tabs, possibly a comment, and a line break, are not considered constituents and thus not ended by a `newline` terminal.

The indentation level of a constituent is described relative to the current block's offside line. `indent` represents indentation past the offside line, and `dedent` represents indentation before it, in line with the indentation and dedentation rules described in Section 4.1.5.

The layout terminals `newline`, `indent`, and `dedent` are written without quotation marks for brevity and to distinguish them from other terminals. Recall the Python-like function definition introduced in Chapter 2. This function definition can now be expressed in LBNF with layout terminals describing its layout syntax, as described by below grammar.

```
CSDef.  CompoundStmt ::= "def" Ident "(" [Ident] ")" ":" Stmts ;

Stmts.  Stmts        ::= newline indent [Stmt] dedent         ;
terminator Stmt newline                                         ;

SPrint. Stmt         ::= "print" "(" String ")"               ;
...
```

Figure 5.1: An LBNF grammar for Python-like function definitions with `newline`, `indent` and `dedent` describing layout syntax

The second production, for `Stmts`, describes the body of a function definition. `newline indent [Stmt] dedent` describes how the function body must begin on a line below the header, with the statements indented and vertically aligned. A line break must terminate each statement, as `terminator Stmt newline` describes.

The definition of `f` below is a program in the language described by the grammar.

```
def f():
    print("hello")
    print("world")
```

On the other hand, `g1`, `g2`, and `g3` are not programs in the language. `g1` is not one because the function body begins on the same line as the header, `g2` because the body is not indented, and `g3` because the statements are not vertically aligned.

```
def g1(): print("hello")
          print("world")
```

```
def g2():
print("hello")
print("world")
```

```
def g3():
    print("hello")
        print("world")
```

As often with context-free grammar, different grammars can describe the same language. For instance, the `newline` terminal marking the end of a statement can either be declared using the `terminator` pragma, as in Figure 5.1, or included last in the production for a statement, as below.

```
CSDef.  CompoundStmt ::= "def" Ident "(" [Ident] ")" ":" Stmts ;

Stmts.  Stmts        ::= newline indent [Stmt] dedent         ;
terminator Stmt ""                                            ;

SPrint. Stmt         ::= "print" "(" String ")" newline       ;
...
```

Figure 5.2: Excerpt of an LBNF grammar for Python-like function definitions with `newline`, `indent` and `dedent` describing layout syntax

Whether to include the `newline` terminal in the production or `terminator` pragma depends on the layout syntax of the described language. It is, however, crucial to do only one or the other, *not both*, to avoid requiring two consecutive `newline` terminals at the end of, for instance, a statement. ”`newline newline`” will never occur in an actual program after layout resolution because of the semantics of `newline`.

5.2 Layout Pragmas Express Global Layout Rules in LBNF

The layout terminals `newline`, `indent`, and `dedent` can be used to describe some layout syntax but, for instance, not all rules stated in Section 4.1. To increase expressivity, we provide a set of layout pragmas. For ease of use, we provide a macro for the indented, vertically aligned block. An LBNF grammar that includes any layout terminals, pragmas or macros is a layout-sensitive grammar, and BNFC will generate a layout resolver from it, further described in Section 5.4.

Since LBNF is a flavour of context-free grammar, the layout rule described by a layout pragma holds globally with some fine print: in an LBNF grammar with layout terminals, we consider layout-sensitivity to be the default. Hence a layout rule holds globally in a purely layout-sensitive language. However, some languages blend layout-sensitivity and insensitivity, and most layout rules only apply in a layout-sensitive context. A layout-insensitive context is achieved by assigning layout-escaping properties to a pair of terminals: within them, layout syntax is not a thing. A typical such pair is `{` and `}`. We will refer to a layout-sensitive block as a layout block and an insensitive block as an escaped block.

5.2.1 Pragmas

- `layout offside`

  Sentences and constituents must adhere to the offside rule.

- `layout start` t_1 [`stop` t_2]

- t_1 opens, and t_2 closes a layout block. The block's offside line is determined by the start column of the first non-layout-related terminal following t_1. The second argument is optional. If included, the block can be closed by t_2 or dedentation; if omitted, the block can only be closed by dedentation.

- `layout escape start` t_1 [`stop` t_2]

- t_1 opens, and t_2 closes an escaped block. The second argument is optional. If omitted, t_1 is used as both opening and closing terminal. The productions describing the contents of the block should not include `newline`, `indent`, or `dedent` terminals.

- `layout escape toplevel`

  The top-level block for a parsed sentence is an escaped block.

- `layout linejoin` t

- t is a line joining terminal; when it is the last terminal on a physical line and not part of a comment or string, the terminals on that line are joined with those of the following line to form a single constituent. The line joining terminal is not part of the constituent, and the indentation of the terminals on the second line is irrelevant.

The only layout rule that applies in escaped blocks is the one described by `layout start` t_1 `stop` t_2. t_1 introduces a layout block regardless of context to allow opening a layout block within an escaped block.

5.2.2 Macros

- `layout block` ns n [`nonempty`]

  A shorthand for the pair of rules

  ns. ns `::=` `indent` `[`n`]` `dedent` ;
  `terminator` n [`nonempty`] `newline` ;

  where `nonempty` is optional. n is an existing nonterminal, and the terminating `newline` must not be in the productions for n. ns is a new nonterminal; it represents the block and can be included in other productions. Since the macro is a shorthand for a `terminator` rule, it occupies the nonterminal `[`n`]`, which represents a list of n terminated by `newline`.

5.3 Layout Syntax Rules Expressed in LBNF

With the given layout terminals, pragmas, and macro at hand, the layout syntax rules presented in Section 4.1 can be expressed in LBNF. Some example productions are given in the following sections; other productions expressing the rules also exist.

5.3.1 The Single-Line Rule

Let `M` be a nonterminal representing a set of constituents without `newline`. The single-line rule is then expressed with a terminating `newline`

```
S.  S ::= M newline ;
```

iff no line joining terminals nor the offside rule are declared.

5.3.2 The Offside Rule

The offside rule overrides the single-line rule, and is enforced through the corresponding layout pragma.

```
layout offside ;
```

5.3.3 Line Joining

Line joining overrides the single-line rule, and a line joining terminal `t` is declared using the corresponding layout pragma.

```
layout linejoin t ;
```

t is not part of the constituents and should therefore not be included in the productions describing them.

5.3.4 Vertical Alignment

Take two nonterminals, `M` and `N`, each representing a set of constituents. Let l_1`. M ::=` α_1 and l_2`. N ::=` α_2 be the productions for the nonterminals, such that that the terminating `newline` terminal is not included in α_1 and α_2. Then, vertical alignment is described as

```
B1.  B ::= M newline N newline ;
```

iff none of the constituents represented by `M` and `N` begin nor end with `indent` or `dedent`.

For a single nonterminal, vertical alignment is expressed with a list, where the list items are terminated by `newline`.

```
B2.  B ::= [M]        ;
terminator M newline ;
```

Alternatively, the `block` macro is used.

```
layout block B M ;
```

5.3.5 Indentation and Dedentation

Indentation of a constituent represented by `N` relative to one represented by `M` is, quite trivially, expressed as

```
I.  I ::= M indent N ;
```

assuming none of the constituents represented by `M` end with `indent` and none of the constituents represented by `N` begin with `indent`.

Dedentation is expressed correspondingly with `dedent`.

5.3.6 Interplay of Vertical Alignment and Indentation

Let `B` represent a vertically aligned group, like those described by the productions in Section 5.3.4. An indented, vertically aligned group is then expressed as

```
IB. IB ::= indent B dedent ;
```

`IB` represents an indented block and is useful when describing a nested block structure. For instance, the structure of a single-line constituent, an indented block, and another single-line constituent that is vertically aligned with the first one, shown below.

```
NB1. NB ::= S IB S ;
```

5.3.7 Interplay of Vertical Alignment, Indentation and the Offside Rule

When the offside rule applies, the `layout start` pragma is needed to achieve block nesting.

```
layout offside       ;
layout start "let"   ;

NB2. NB ::= "let" IB ;
```

Note how this allows the indented block to start on the same line as `let`. Requiring the block to start on a new line is expressed as

```
NB3. NB ::= "let" newline IB ;
```

Unlike blocks opened by a start terminal like `let`, blocks opened by indentation alone can only begin on a new line which must be reflected in the grammar by a `newline` immediately preceding the `indent`. In the production `NB1` above, the `newline` is included in the production for `S`.

Correspondingly, declaring a stop terminal, like `in`, allows the stop terminal and the constituent following it to be on the same line as the last line of the block.

```
layout offside                    ;
layout start "let" stop "in"      ;

NB4. NB ::= "let" IB "in" S       ;
NB5. NB ::= "let" newline IB "in" S ;
```

Stop terminals give rise to a situation where `newline` does not necessarily represent a physical line break (\n) and `dedent` not necessarily dedentation. If this feels unintuitive, remember how `newline`, `indent`, and `dedent` are the layout-syntactic equivalents of `;`, `{` and `}`. Consider a sentence matching productions `NB4` or `NB5`; the stop terminal, `in`, may well be on the same line as the last non-layout-related terminal in the block. However, a `newline` should be used to mark the end of the last constituent in the block and a `dedent` to mark the end of the block.

5.4 Layout Resolution

Plugged in-between lexer and parser, the layout resolver operates on a stream of tokens, ts, and inserts `newline`, `indent`, and `dedent` tokens into the stream, representing the parsed sentence's layout syntax. The layout resolution algorithm is defined recursively and visits each token in ts once. The algorithm takes two parameters, t and ts, and has a global state φ, further described in sections 5.4.1 and 5.4.3, respectively. The set of terminals, or tokens, is denoted by T, the set of blocks by B, and the set of n-tuples, or lists, is denoted A^n, $n \in \mathbb{N}$.

t	$\in T$		Inspected token in token stream
ts	$\in T^m$	$m \in \mathbb{N}$	Remaining token stream
φ	$\in B^n$	$n \in \mathbb{N}$	Block stack

The layout resolution process is demonstrated with a Python example, namely the definition of a ”hello world” function. The grammar in Figure 5.3 describes Python's function definitions and layout syntax and is similar to that in Figure 4.2. Python's line joining terminal, the backslash, is described with the `layout linejoin` pragma. Expressions in round brackets can span multiple lines, which is expressed using the `layout escape` pragma.

```
layout linejoin "\\"                                                  ;
layout escape start "(" stop ")"                                      ;

CSDef.        CompoundStmt ::= "def" Ident "(" [Ident] ")" ":" Stmts ;
separator Ident ","                                                   ;

StmtsInl.     Stmts        ::= StmtsLine1 newline                     ;
StmtsMultil.  Stmts        ::= newline indent StmtsBlock dedent       ;

SBl1.         StmtsBlock   ::= StmtsLine1 newline                     ;
SBl2.         StmtsBlock   ::= StmtsLine1 newline StmtsBlock          ;
SBl3.         StmtsBlock   ::= CompoundStmt                           ;
SBl4.         StmtsBlock   ::= CompoundStmt StmtsBlock                ;

-- snoc list to allow the optional, trailing semicolon
SL1Nil.       StmtsLine1   ::= Stmt                                   ;
SL2Nil.       StmtsLine2   ::= Stmt ";"                               ;
SL2Snoc.      StmtsLine2   ::= StmtsLine2 Stmt                        ;
SL2CSnoc.     StmtsLine2   ::= StmtsLine2 Stmt ";"                    ;
coercions StmtsLine 2                                                 ;

SCall.        Stmt         ::= Ident "(" [Exp] ")"                    ;
separator Exp ","                                                     ;

EStr.         Exp          ::= String                                 ;
```

Figure 5.3: An LBNF grammar for Python's function definitions and function calls with layout syntax

Below is the subject of demonstration: the definition of a function `f` that prints "hello world".

```python
def f():
    print\
    ("hello world")
```

Figure 5.4: A "hello world" function in Python with a line-joined print statement

5.4.1 Token Stream

The treatment of a token may depend on the token following it why tokens are inspected pairwise; t is inspected relative to t', the first token in ts. The treatment also depends on the tokens' positions; luckily, the tokens carry position information. The line and column of a token $t \in ts$ is denoted by t_{line} and t_{col} respectively. The undefined token is represented by ε.

For n-tuples $ts \in T^n$ indexed from 0, such as the token stream, $ts[i]$ represents the i^{th} element; and $ts[i:]$ represents the elements at index i and above. The cons operator : is right-associative, so $x : y : z$ is interpreted as $x : (y : z)$.

5.4.2 Layout Pragma Variables

Layout pragmas affect layout resolution; they and the tokens reserved using them are variables of the layout resolution. The boolean variables *offside* and *toplevel* are set to `true` if corresponding layout pragmas are used. The sets S_{layout}, $S_{esc} \subset T \times T$ contain ordered pairs of start and stop tokens for layout blocks and escaped blocks, respectively. Let $T_{l_{start}}$, $T_{l_{stop}} \subset T$ be the sets of start and stop tokens for layout blocks and $T_{e_{start}}$, $T_{e_{stop}} \subset T$ be the ones for escaped blocks, all pairwise disjoint except for $T_{e_{start}}$ and $T_{e_{stop}}$. Then, $S_{layout} \subset T_{l_{start}} \times T_{l_{stop}}$ and $S_{esc} \subset T_{e_{start}} \times T_{e_{stop}}$. The set $T_{linejoin} \subset T$ contains line joining tokens. A token, or pair of tokens, reserved using a layout pragma is a member of the corresponding set, as illustrated below.

$$\begin{array}{llllll}
\texttt{layout start } t & \rightarrow & \langle t, \varepsilon \rangle \in S_{layout} & \wedge & t \in T_{l_{start}} & \\
\texttt{layout start } t_1 \texttt{ stop } t_2 & \rightarrow & \langle t_1, t_2 \rangle \in S_{layout} & \wedge & t_1 \in T_{l_{start}} & \wedge \ t_2 \in T_{l_{stop}} \\
\texttt{layout escape start } t & \rightarrow & \langle t, t \rangle \in S_{esc} & \wedge & t \in T_{e_{start}} & \wedge \ t \in T_{e_{stop}} \\
\texttt{layout escape start } t_1 \texttt{ stop } t_2 & \rightarrow & \langle t_1, t_2 \rangle \in S_{esc} & \wedge & t_1 \in T_{e_{start}} & \wedge \ t_2 \in T_{e_{stop}} \\
\texttt{layout linejoin } t & \rightarrow & t \in T_{linejoin} & & &
\end{array}$$

The Python grammar in Figure 5.3 yields the following variables for layout resolution.

$$\begin{array}{lcl}
\textit{offside} & = & \texttt{false} \\
\textit{toplevel} & = & \texttt{false} \\
S_{layout} & = & \emptyset \\
S_{esc} & = & \{ \langle \texttt{"("}, \texttt{")"} \rangle \} \\
T_{linejoin} & = & \{ \texttt{"\\\\"} \}
\end{array}$$

5.4.3 Block Stack

The insertion of `indent` and `dedent` tokens depends on the block context, why the layout resolver maintains a stack of blocks. Type B denotes the set of blocks, where a block $b \in B$ is either a layout block or an escaped block: $b \in (\mathbb{N} \times S_{layout}) \mid S_{esc}$. A layout block opened by a start token t_1 is denoted $\langle c, t_1, t_2 \rangle$, with t_2 being the stop token for t_1, if one exists, and c the offside line of the block. A layout block opened by indentation alone has no start or stop tokens and is therefore denoted $\langle c, \varepsilon, \varepsilon \rangle$. On the contrary, an escaped block must have a start and stop token but no offside line and is denoted $\langle t_1, t_2 \rangle$.

We introduce some shorthand notation for a block b:

b	$\langle c, t_1, t_2 \rangle$	$\langle t_1, t_2 \rangle$	
b_{col}	c	ε	Offside line
b_{start}	t_1	t_1	Start token
b_{stop}	t_2	t_2	Stop token

The block stack, $\varphi \in B^n$, is modified through PUSH and POP operations and can be observed using PEEK. With only one stack, a reference to it is omitted in PUSH, POP and PEEK calls, so PUSH is a unary operation whilst POP and PEEK are nullary operations.

In connection with φ, we define $T_{\varphi_{stop}} \subseteq T_{l_{stop}} \cup T_{e_{stop}}$ as the set of tokens that close blocks if encountered in the token stream. A layout block *can* be closed by a stop token, while an escaped block *must* be closed by a stop token. Therefore, $T_{\varphi_{stop}}$ contains the stop tokens for the topmost layout blocks on the stack until and including the topmost escaped block. If the stack contains only layout blocks, $T_{\varphi_{stop}}$ is the set of stop tokens for all blocks on the stack. Note that $\varepsilon \notin T_{\varphi_{stop}}$, so $T_{\varphi_{stop}}$ may be empty. The top-level block can never be closed, so the bottom block in φ remains throughout layout resolution and is unaffected by block closing operations.

5.4.4 Initialisation

On initialisation, t is undefined, and ts is the complete token stream of the parsed sentence. The block stack, φ, is initialised with a layout block $\langle 1, \varepsilon, \varepsilon \rangle$ unless *toplevel* is `true`, in which case it is initialised with an escaped block $\langle \varepsilon, \varepsilon \rangle$.

In the Python example, ts represents the program in Figure 5.4, and the block stack is initialised with a layout block since *toplevel* = `false`.

$t = \varepsilon$

$ts =$ [`"def"`, `"f"`, `"("`, `")"`, `":"`, `"print"`, `"\\"`, `"("`, `"\"hello world\""`, `")"`]

$\varphi = [\ \langle 1, \varepsilon, \varepsilon \rangle\ ]$

5.4.5 Algorithm

The layout resolution process is first demonstrated with a Python example, then described in full by Algorithm 1 and its auxiliary functions, Algorithms 2 - 8. Algorithm 1 describes layout resolution independent of block context, whereas algorithm 2 describes layout resolution specific to layout blocks. Algorithms 3 - 8 describe block stack operations.

The layout resolver visits each token in the token stream once but only acts on two things: line breaks and reserved terminals from layout pragmas. Therefore, the following demonstration only includes the iterations of the algorithm where the layout resolver takes action. Resolving the layout of the Python program, the currently inspected token, t, and the token stream ts, are illustrated like below, together with the state of the block stack, φ. The layout-resolved token stream from previous iterations is included in light grey.

`"def" "f"`

$$\underbrace{\texttt{"("}}_{t}\ \underbrace{\texttt{")" ":" "print" "}\backslash\backslash\texttt{" "(" "}\backslash\texttt{"hello world}\backslash\texttt{"" ")"}}_{ts}$$

$$\varphi \ = \ [\ \langle 1, \varepsilon, \varepsilon \rangle\ ]$$

With the above input, the layout resolver acts since `"("` $\in T_{e_{start}}$. It opens an escaped block, as in the case on line 29 in Algorithm 1, which affects the block stack and the set of tokens that can close blocks, $T_{\varphi_{stop}}$, as shown below. The set $T_{\varphi_{stop}}$ is only shown when non-empty.

`"def" "f" "("`

$$\underbrace{\texttt{")"}}_{t}\ \underbrace{\texttt{":" "print" "}\backslash\backslash\texttt{" "(" "}\backslash\texttt{"hello world}\backslash\texttt{"" ")"}}_{ts}$$

$$\begin{aligned} \varphi &= [\ \langle \texttt{"("}, \texttt{")"} \rangle,\ \langle 1, \varepsilon, \varepsilon \rangle\ ] \\ T_{\varphi_{stop}} &= \{\ \texttt{")"}\ \} \end{aligned}$$

Now, $t =$ `")"`, which also calls for action since `")"` $\in T_{\varphi_{stop}}$. The resolver closes all blocks on the stack until and including the block with `")"` as stop token, by the case on line 25 in Algorithm 1. Observe the effect on φ, where $\langle \texttt{"("}, \texttt{")"} \rangle$ has been popped off the stack. Consequently, $T_{\varphi_{stop}}$ is empty.

`"def" "f" "(" ")"`

$$\underbrace{\texttt{":"}}_{t}\ \underbrace{\texttt{"print" "}\backslash\backslash\texttt{" "(" "}\backslash\texttt{"hello world}\backslash\texttt{"" ")"}}_{ts}$$

$$\varphi \ = \ [\ \langle 1, \varepsilon, \varepsilon \rangle\ ]$$

Being in a layout-sensitive block, the layout resolver acts on line breaks. It detects a line break by comparing the lines of `":"` and `"print"`, done on line 4 in Algorithm 2. Furthermore, it detects that `"print"` is indented relative to the current offside line by comparing columns on line 7 in the same algorithm. Since *offside* = `false`, the layout resolver opens a layout block by indentation with the column of `"print"`, 5, as offside line. It inserts the tokens `newline` and `indent` into the layout-resolved token stream to reflect the layout.

`"def" "f" "(" ")" ":" "newline" "indent" "print"`

$$\underbrace{\texttt{"}\backslash\backslash\texttt{"}}_{t}\ \underbrace{\texttt{"(" "}\backslash\texttt{"hello world}\backslash\texttt{"" ")"}}_{ts}$$

$$\varphi \ = \ [\ \langle 5, \varepsilon, \varepsilon \rangle,\ \langle 1, \varepsilon, \varepsilon \rangle\ ]$$

Again, the layout resolver detects a line break by comparing the lines of `"\\"` and `"("`. Though this time, `"\\"` $\in T_{linejoin}$, which is caught in the case on line 5 in Algorithm 2. The layout resolver simply omits the backslash from the token stream and does *not* insert a `newline`, allowing the print statement to continue on the next line.

`"def" "f" "(" ")" ":" "newline" "indent" "print"`

$$\underbrace{\texttt{"("}}_{t}\ \underbrace{\texttt{"\textbackslash"hello world\textbackslash"" ")"}}_{ts}$$

$$\varphi \ = \ [\ \langle 5,\varepsilon,\varepsilon\rangle,\ \langle 1,\varepsilon,\varepsilon\rangle\ ]$$

As known by now, `"("` $\in T_{e_{start}}$ so the layout resolver opens an escaped block, which affects φ and $T_{\varphi_{stop}}$.

`"def" "f" "(" ")" ":" "newline" "indent" "print" "(" "\"hello world\""`

$$\underbrace{\texttt{")"}}_{t}\ \underbrace{\texttt{[]}}_{ts}$$

$$\begin{aligned} \varphi &= [\ \langle \texttt{"("}, \texttt{")"}\rangle,\ \langle 5,\varepsilon,\varepsilon\rangle,\ \langle 1,\varepsilon,\varepsilon\rangle\ ] \\ T_{\varphi_{stop}} &= \{\ \texttt{")"}\ \} \end{aligned}$$

And again, `")"` $\in T_{\varphi_{stop}}$, closing the escaped block. Though this time, since *ts* is empty, the stop token is caught by the case on line 9 in Algorithm 1. The layout resolver performs another iteration with $t =$ `")"` to also resolve the layout.

`"def" "f" "(" ")" ":" "newline" "indent" "print" "(" "\"hello world\""`

$$\underbrace{\texttt{")"}}_{t}\ \underbrace{\texttt{[]}}_{ts}$$

$$\varphi \ = \ [\ \langle 5,\varepsilon,\varepsilon\rangle,\ \langle 1,\varepsilon,\varepsilon\rangle\ ]$$

With *ts* empty, one can imagine a last, invisible token on a line below t and at column 1. Being in a layout block, the layout resolver inserts a `newline` into the token stream to represent the line break between `")"` and this imagined last token. Furthermore, the imagined token is dedented relative to the offside line of any block on the stack except the bottom block, resulting in a total block closing as per Algorithm 8. For each closed layout block, the layout resolver inserts a `dedent`.

`"def" "f" "(" ")" ":" "newline" "indent" "print" "(" "\"hello world\""`
`")" "newline" "dedent"`

$$\underbrace{\varepsilon}_{t}\ \underbrace{\texttt{[]}}_{ts}$$

$$\varphi \ = \ [\ \langle 1,\varepsilon,\varepsilon\rangle\ ]$$

With t undefined and *ts* empty, the layout resolution is finished. The resulting, layout-resolved token stream is displayed below, as code for readability. Note how there is no trace of the line joining token or the line break following it in the resolved token stream.

```
def f(): newline
indent print
       ("hello world") newline
dedent
```

Terminals `newline`, `indent`, and `dedent` now represent the program's layout, allowing the parser to check it against the Python grammar with layout syntax in Figure 5.3. The relevant productions are repeated below.

```
CSdef.        CompoundStmt ::= "def" Ident "(" [Ident] ")" ":" Stmts ;

StmtsMultil. Stmts        ::= newline indent StmtsBlock dedent      ;

SBl1.         StmtsBlock   ::= StmtsLine1 newline                    ;
SL1Nil.       StmtsLine1   ::= Stmt                                  ;
SCall.        Stmt         ::= Ident "(" [Exp] ")"                   ;
EStr.         Exp          ::= String                                ;
...
```

The print statement ended by `newline` constitutes a `StmtsBlock`. Terminals `newline indent` preceeds the `StmtsBlock` and `dedent` succeeds it, which matches the production for `Stmts`. Finally, the header and `Stmts` constitute a `CompoundStmt` and, thereby, a valid Python program.

The layout resolution process is described in full by the following algorithms. As mentioned, Algorithm 1 describes layout resolution independent of block context, Algorithm 2 describes layout resolution specific to layout blocks, and algorithms 3 - 8 describe block stack operations. In algorithms 1 and 2, **return** is omitted in favour of brevity; the last expression of a clause is returned.

Algorithm 1 Layout Resolution

```
1: function RESOLVE($t$, $ts$)
2:     $b$ := PEEK $\varphi$
3:     $t'$ := $ts[0]$
4:     if $ts$ empty then                                          ▷ End of token stream
5:         if $t$ undefined then                                   ▷ Empty token stream
6:             [ ]
7:         else if $t \in T_{l_{start}}$ then                      ▷ Layout start token
8:             $t$ : newline : indent : dedent : CLOSEALL
9:         else if $t \in T_{\varphi_{stop}}$ then                 ▷ Stop token
10:            if $b$ is layout block then
11:                CLOSEUNTIL($t$) ++ $t$ : newline : CLOSEALL
12:            else
13:                CLOSEUNTIL($t$) ++ $t$ : CLOSEALL
14:        else if $b$ is layout block then
15:            $t$ : newline : CLOSEALL
16:        else
17:            $t$ : CLOSEALL
18:    else if $t$ undefined then                                  ▷ Beginning of token stream
19:        if $b$ is layout block $\wedge$ $b_{col} < t'_{col}$ then
20:            OPENL($t'_{col}$)
21:            indent : RESOLVE($ts[0]$, $ts[1:]$)
22:        else
23:            RESOLVE($ts[0]$, $ts[1:]$)
24:    else                                                        ▷ Arbitrary token in token stream
25:        if $t \in T_{\varphi_{stop}}$ then                      ▷ Stop token
26:            CLOSEUNTIL($t$) ++ RESOLVEL($t$, $ts$)
27:        else if $t \in T_{l_{start}} \wedge t' \in T_{l_{start}} \cup T_{e_{start}}$ then   ▷ Stacked start tokens
28:            $t$ : RESOLVE($t'$, $ts[1:]$)
29:        else if $t \in T_{e_{start}}$ then                      ▷ Escape start token
30:            OPENESC($t$)
31:            $t$ : RESOLVE($t'$, $ts[1:]$)
32:        else if $t \in T_{l_{start}}$ then                      ▷ Layout start token
33:            OPENL($t'_{col}$, $t$)
34:            if $t_{line} = t'_{line}$ then
35:                $t$ : indent : RESOLVE($t'$, $ts[1:]$)
36:            else if $b$ is escaped block $\vee$ $b_{col} < t'_{col}$ then
37:                $t$ : newline : indent : RESOLVE($t'$, $ts[1:]$)
38:            else
39:                $t$ : newline : indent : dedent : CLOSEUNTIL($t'_{col}$)
                       ++ RESOLVE($t'$, $ts[1:]$)
40:        else if $b$ is layout block then
41:            RESOLVEL($t$, $ts$)
42:        else
43:            $t$ : RESOLVE($t'$, $ts[1:]$)
```

Algorithm 2 Layout Resolution in Layout Block

```
1:  function resolveL($t$, $ts$)
2:      $b$ := peek $\varphi$
3:      $t'$ := $ts[0]$
4:      if $t_{line} < t'_{line}$ then                                  ▷ Last token on line
5:          if $t \in T_{linejoin}$ then                                ▷ Line joining token
6:              resolve($t'$, $ts[1:]$)
7:          else if $b_{col} < t'_{col}$ then                           ▷ Right of offside line
8:              if $offside$ then
9:                  $t$ : resolve($t'$, $ts[1:]$)
10:             else
11:                 openL($t'_{col}$)
12:                 $t$ : newline : indent : resolve($t'$, $ts[1:]$)
13:         else if $b_{col} = t'_{col}$ then                           ▷ On offside line
14:             $t$ : newline : resolve($t'$, $ts[1:]$)
15:         else                                                        ▷ Left of offside line
16:             $dedents$ = closeUntil($t'_{col}$)
17:             $t$ : newline : $dedents$ ++ resolve($t'$, $ts[1:]$)
18:     else if $t' \in T_{\varphi_{stop}}$ then                        ▷ Upcoming stop token
19:         $t$ : newline : resolve($t'$, $ts[1:]$)
20:     else
21:         $t$ : resolve($t'$, $ts[1:]$)
```

Algorithm 3 Layout Block Opening

```
1: function openL($c$, $t_1$)
2:     if $t_1$ undefined then
3:         push $\langle c, \varepsilon, \varepsilon \rangle$ $\varphi$
4:     else
5:         $t_2$ := lookup $t_1$ in $S_{layout}$
6:         push $\langle c, t_1, t_2 \rangle$ $\varphi$
```

Algorithm 4 Escaped Block Opening

```
1: function openEsc($t_1$)
2:     $t_2$ := lookup $t_1$ in $S_{esc}$
3:     push $\langle t_1, t_2 \rangle$ $\varphi$
```

Algorithm 5 Single Block Closing

```
1: function close
2:     pop $\varphi$
```

Algorithm 6 Multiple Block Closing

```
 1: function closeUntil($t_{stop}$)    ▷ Closes all blocks until and including
 2:     $b$ := peek $\varphi$                   the block with stop token $t_{stop}$
 3:     $dedents$ = [ ]
 4:     while $b_{stop}$ undefined $\vee$ $b_{stop} \neq t_{stop}$ do
 5:         close
 6:         $dedents$ = dedent : $dedents$
 7:     if $b$ is layout block then
 8:         $dedents$ = dedent : $dedents$
 9:     close
10:     return $dedents$
```

Algorithm 7 Multiple Block Closing

```
 1: function closeUntil($c$)    ▷ Closes all blocks with offside lines
 2:     $b$ := peek $\varphi$                to the right of column $c$
 3:     $dedents$ = [ ]
 4:     while $b$ is layout block $\wedge$ $c < b_{col}$ do
 5:         close
 6:         $dedents$ = dedent : $dedents$
 7:     return $dedents$
```

Algorithm 8 Total Block Closing

```
 1: function closeAll
 2:     $b$ := peek $\varphi$
 3:     $dedents$ = [ ]
 4:     while $1 < |\varphi|$ do
 5:         if $b$ is layout block then
 6:             $dedents$ = dedent : $dedents$
 7:         close
 8:     return $dedents$
```

5.5 Implementation

The enhanced layout syntax support, as described in sections 5.1, 5.2, and 5.4, is incorporated in BNFC version 3.0. The third version is a reimplementation of BNFC not yet available to the public.

6

Pfenning Safety

Recall the Pfenning principle: "α-renaming an identifier in a valid program produces an equivalent, valid program". The statement holds for all programs in a Pfenning-safe language; it often holds for some, but not all, programs in a language that is not Pfenning-safe.

6.1 Safe Programs

A program w is Pfenning-safe if multiline layout blocks begin on lines where renamable identifiers do not precede them. In other words, a layout-resolved program w is Pfenning-safe if all occurrences of a renamable identifier, eventually followed by `indent`, are first followed by a `newline`.

The Pfenning safety of a program w can be decided by controlling the token stream after layout resolution; the below automaton describes this process. In the automaton, *flexible* represents renamable identifiers. They are tokens of the basic lexical types `String` and `Ident`, and lexical types defined by `token` rules. Furthermore, *fixed* represents fixed-size tokens other than `newline` and `indent`. If a renamable identifier is immediately followed by `indent`, the automaton ends in the rejecting state, s_2, and w is not Pfenning-safe. Otherwise, the automaton ends in one of the accepting states, s_0 and s_1, and the program is Pfenning-safe.

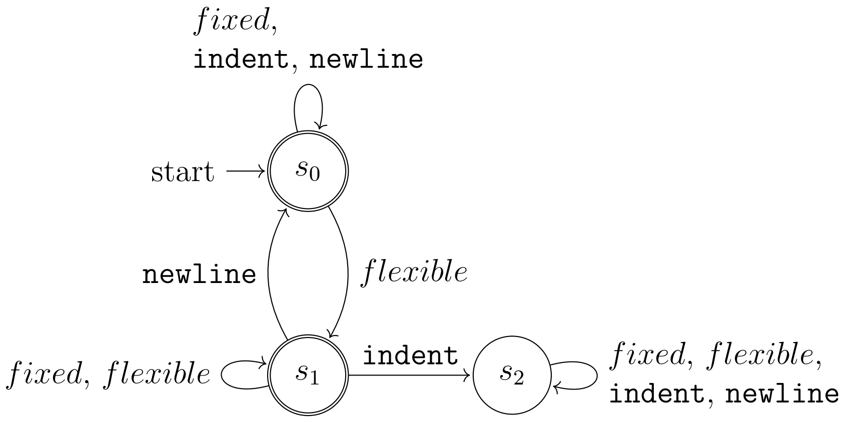


Figure 6.1: Pfenning-safety control of a layout-resolved program

6.2 Safe Languages

The layout-sensitive language described by a grammar $G = (N, T, P, S)$ is Pfenning-safe if all programs $w \in L(G)$ are so. In LBNF, renamable identifiers are of the basic lexical types `String` and `Ident`, or a lexical type defined by the user with a `token` rule. Hence it should be possible to determine the Pfenning safety of a language by analysing its LBNF grammar, for instance, by analysing the follow sets of the nonterminals. We leave such an analysis for future work.

7 Discussion

LBNF has reserved terminals, pragmas, and a macro for describing layout syntax. Together with a layout resolver, plugged in-between lexer and parser in the generated compiler front-ends, they allow describing and verifying standard layout syntax. Because the layout resolver is self-contained, it is relatively straightforward to extend the support to other target languages than Haskell. On the downside, a parser-independent layout resolver cannot utilise parsing information to avoid parse errors.

Standard layout syntax is describable in LBNF; both purely layout-sensitive languages and those mixing layout-sensitive and insensitive syntax can be described. Nonetheless, LBNF's expressivity has its limits. We here discuss two examples of layout syntax that is not expressible: Haskell and Agda-like module declarations and requirements on the amount of indentation. Lastly, we call attention to the global effects of reserving terminals with layout pragmas.

7.1 BNFC's Layout Formalism

BNFC's layout formalism relies on verbalising layout syntax. Reserve a set of terminals for representing layout, and use them to express the layout rules in the LBNF grammar. Then, insert layout terminals in the token stream from the lexer before feeding it to the parser, and have the parser verify that the program adheres to the layout rules specified by the grammar. This approach is not new. Python uses `newline`, `indent`, and `dedent` together with a layout resolver between lexer and parser. Haskell, essentially, does the same thing with `;`, `{`, and `}`, and lets the layout resolver translate layout into explicit syntax since the language has both kinds of syntax. Where BNFC differs is that the tool can generate a layout resolver for any layout-sensitive language described by an LBNF grammar. Therefore, the layout resolver generated by BNFC is adaptable to standard layout rules specified by layout pragmas in the grammar.

BNFC is a tool relying on existing lexer and parser generators and the parsing algorithms that the latter support. Given these circumstances, we deemed it the best solution to generate a layout resolver plugged in-between lexer and parser. The layout resolver is relatively self-contained and only depends on the lexer's positional data associated with the tokens in the token stream. Because the layout resolver is self-contained, extending BNFC's layout syntax support to other target languages should be straightforward.

7.1.1 Portability to Target Languages Other than Haskell

LBNF's layout terminals, pragmas and macro are the same for all target languages, and so is the internal representation of an LBNF grammar in BNFC version 3.0. Thus layout syntax in other target languages is supported by LBNF as-is.

The layout resolution algorithm is described in pseudocode in Section 5.4 and does not depend on Haskell-specific constructs. It should therefore be possible, and relatively straightforward, to implement the generation of a layout resolver in other target languages such as Java.

7.1.2 Stand-Alone Layout Resolvers Cannot Use Parsing Information

A drawback of a separate layout resolver is that it cannot utilise parsing information. Specifically, the layout resolver cannot consult the parser for parse errors avoidable by inserting a `dedent` terminal, similar to Haskell's parse-error(t) [22]. The consultation is this: let t be an arbitrary token in the token stream and ts the layout-resolved token stream before t. Then, if ts t is not a valid prefix of the language's grammar, but ts `dedent` t is, insert `dedent` into the token stream. Agda performs similar parser consultation for closing a `let` – `in` block at `in`. BNFC's layout resolver can handle this if `in` is declared a layout stop terminal, but cannot handle more complex cases where parser consultation could avoid parse errors.

7.2 LBNF's Expressivity Limits

It is possible to express the standard layout rules in LBNF with the layout terminals and pragmas, as shown in Section 5.3. However, expressivity is limited by the chosen layout terminals and pragmas and the fact that LBNF is a form of context-free grammar. We give and discuss two examples of layout syntax that lie beyond LBNF's current expressivity limits.

7.2.1 Haskell and Agda-like Module Declaration

In Haskell and Agda, all occurrences of `where` require the layout block following it to be indented, with one exception: the block of declarations following `where` in the top-level module declaration. There, the layout block is not required to be indented, and the languages allow both layouts below.

```
module M where                  module M where
    f :...                      f :...
```

Such context-sensitive requirements are not expressible in LBNF. A layout block must be opened to enforce vertical alignment of the declarations following `where`. `where` must be declared a layout start terminal to open the layout block since the languages adhere to the offside rule. A layout block is, by definition, indented relative to the offside line of the enclosing block, which is why the layout of the right module declaration above cannot be described in LBNF.

A nested layout block could have the same offside line as its enclosing block by tweaking the layout resolver; change the strict inequality to $\leq$ on line 36 in Algorithm 1. This change would allow both layouts above but has a global effect. It is thus similar to Haskell's language extension NondecreasingIndentation [23] and does not express the special layout treatment of `where` in the module declaration.

7.2.2 Requirements on the Amount of Indentation

The `indent` terminal represents any indentation past the offside line and poses no requirements on the amount of indentation. Its definition limits the expressivity of LBNF. Whether this limit can, and if so should, be breached is discussable. Requirements on the amount of indentation can be broken down into two kinds: exact requirements; and lower and upper bounds.

Exact requirements are expressible in LBNF by reserving a set of terminals and altering the layout resolution algorithm. The terminal `indent`n could represent indentation of n blank space characters relative to the current offside line, where $n \in \mathbb{N}$. The `indent` terminal would still represent any indentation past the offside line, that is, of $n > 0$ blank spaces. The layout resolver would insert `indent`n, instead of `indent`, terminals into the token stream. With a built-in type, `Indent`, defined as `("indent" [1 - 9] digit*)`, BNFC would substitute any occurrences of `indent` in the grammar with the nonterminal `Indent`. An `indent`n terminal in the layout-resolved token stream would then match the terminal `indent`n and the nonterminal `Indent`. This solution would allow expressing requirements of any indentation, with `indent`, and exact requirements, with `indent`n.

An indentation of zero blank spaces is, in fact, *no* indentation, and therefore `indent0` would not be of the type `Indent`. However, the layout-resolver would insert an `indent0` terminal in-between a layout start terminal and the terminal following it *if* the latter starts at the current offside line. Interestingly, this would, together with a slight alteration of the layout-resolution algorithm, allow expressing Haskell and Agda-like module declarations:

```
layout start "where"                                          ;
Module. Module ::= "module" Ident "where" indent0 [Decl] dedent ;
```

Lower and upper bound requirements would be less trivial to express, if possible, and are left for future work.

From a language design perspective, whether BNFC should support requirements on the amount of indentation is debatable. For instance, YAML's documentation [24] advises against using the exact amount of indentation to convey information. Even the most straightforward requirements, such as indentation of exactly two blank spaces, introduce additional concerns for someone writing a program in the language and do not add much value except perfectly indented code.

7.3 Terminals Reserved with a Layout Pragma Must be Handled with Care

LBNF is a form of CFG, so reserving a terminal using a layout pragma has a global effect, the implication of which is best illustrated with an example using the `layout start` pragma. Layout start terminals allow layout blocks to start on the same line as the start terminal, as explained in Section 5.3.7. Returning to Python-like function definitions, we can describe a language where the function body can begin on the same line as the header by making the colon a layout start terminal.

```
layout start ":"                                                  ;

CSInl.  CompoundStmt ::= "def" Ident "(" [Ident] ")" ":" Stmts         ;
CSNewl. CompoundStmt ::= "def" Ident "(" [Ident] ")" ":" newline Stmts ;

Stmts.  Stmts        ::= indent [Stmt] dedent                          ;
terminator Stmt newline                                                ;

SPrint. Stmt         ::= "print" "(" String ")"                        ;
...
```

Figure 7.1: An LBNF grammar for Python-like function definitions with layout syntax

Both definitions of `f` and `g1` below are valid programs in the language described by the grammar in Figure 7.1, in contrast to the language described in Figure 5.1 where `g1` is not valid.

```
def f():                        def g1(): print("hello")
    print("hello")                        print("world")
    print("world")
```

Now, if one were to describe a Python-like lambda expression for the same language, described in Figure 7.1, it could be described by the following production without a second thought.

```
Lambda.    Lambda ::= "lambda" [Param] ":" Exp ;
```

Only, and this is the caveat, no layout-resolved programs will match this production. Because the colon is a layout start terminal, it opens a layout block, and the layout resolver will have inserted an `indent` terminal in-between the colon and the expression. The lambda expression must instead be described by the following production with `indent` and `dedent`.

```
Lambda.    Lambda ::= "lambda" [Param] ":" indent Exp dedent ;
```

Hence, a note of caution on layout start terminals: they must be treated as such in all productions where they occur. Layout stop terminals have a lower risk for this kind of pitfall, both because they are associated with a specific layout start terminal and because they are less common than start terminals.

8

Conclusion

The BNF Converter now offers comprehensive support for layout syntax in compiler front-ends generated in Haskell. Standard layout syntax can be described in LBNF with the reserved terminals `newline`, `indent`, and `dedent`, a collection of layout pragmas, and a macro. Both purely layout-sensitive languages and those mixing layout-sensitive and insensitive syntax can be described. Given an LBNF grammar describing a layout-sensitive language, BNFC generates a compiler front-end with a custom layout resolver plugged in-between lexer and parser. The compiler front-end verifies the regular syntax and layout syntax of a program written in the layout-sensitive language. Extending BNFC's layout syntax support to other target languages than Haskell is straightforward since we provide a target language-independent algorithm for layout resolution.

Layout syntax can introduce instability under α-renaming of identifiers in a program; we call a program Pfenning-safe if α-renaming an identifier produces an equivalent, valid program. An automaton can determine the Pfenning safety of an individual, layout-resolved program. Determining the Pfenning-safety of an entire language should be possible by analysing its LBNF grammar; we encourage further research on such an analysis.

Bibliography

[1] P. J. Landin, "The next 700 Programming Languages," *Communications of the ACM*, vol. 9, no. 3, pp. 157–166, Mar. 1966, ISSN: 0001-0782. [Online]. Available: `https://doi.org/10.1145/365230.365257`.

[2] Python Software Foundation, *Compound Statements*, version 3.10.7, Nov. 18, 2022. [Online]. Available: `https://docs.python.org/3.10/reference/compound_stmts.html`.

[3] L. Brunauer and B. Mühlbacher, "Indentation Sensitive Languages," *Unpublished manuscript, July*, 2006.

[4] M. D. Adams, "Principled Parsing for Indentation-Sensitive Languages: Revisiting Landin's Offside Rule," *SIGPLAN Not.*, vol. 48, no. 1, pp. 511–522, 2013, ISSN: 0362-1340. DOI: `10.1145/2480359.2429129`. [Online]. Available: `https://doi.org/10.1145/2480359.2429129`.

[5] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann, "Layout-sensitive Generalized Parsing," in *Software Language Engineering*, K. Czarnecki and G. Hedin, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 244–263. [Online]. Available: `https://doi.org/10.1007/978-3-642-36089-3_14`.

[6] F. Zhu, J. Liu, and F. He, "Lay-it-out: Interactive Design of Layout-Sensitive Grammars," 2022, Preliminary draft. [Online]. Available: `https://arxiv.org/pdf/2203.16211.pdf`.

[7] A. Ranta and M. Forsberg, *Implementing Programming Languages.* London: College Publications, 2012, ISBN: 978-1-84890-064-6.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation, 3rd Edition* (Pearson international edition). Addison-Wesley, 2007, ISBN: 978-0-321-47617-3.

[9] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley series in computer science / World student series edition). Addison-Wesley, 1986, ISBN: 0-201-10088-6.

[10] M. Sipser, *Introduction to the theory of computation.* PWS Publishing Company, 1997, ISBN: 978-0-534-94728-6.

[11] P. R. Kroeger, *Analyzing Grammar: An Introduction.* Cambridge University Press, 2005, ISBN: 9781139443517. [Online]. Available: `https://books.google.se/books?id=rSglHbBaNyAC`.

[12] N. Wirth, *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976, ISBN: 978-0-13-022418-7.

[13] A. Abel, J. Almström Duregård, K. Angelov, *et al.* "The BNF Converter." (Sep. 11, 2022), [Online]. Available: `https://bnfc.digitalgrammars.com`.

[14] BNFC Developers, *Layout Syntax*, version 2.9.4. [Online]. Available: `https://bnfc.readthedocs.io/en/v2.9.4/lbnf.html#layout-syntax`.

[15] L. E. de Souza Amorim, M. J. Steindorfer, S. Erdweg, and E. Visser, "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 2018, Boston, MA, USA, November 05-06, 2018*, D. J. Pearce, T. Mayerhofer, and F. Steimann, Eds., ACM, 2018, pp. 3–15. DOI: `10.1145/3276604.3276607`. [Online]. Available: `https://doi.org/10.1145/3276604.3276607`.

[16] Python Software Foundation, *Line Structure*, version 3.10.7, Nov. 3, 2022. [Online]. Available: `https://docs.python.org/3/reference/lexical_analysis.html#line-structure`.

[17] N. A. Danielsson, *Make Agda "Pfenning-safe"?* Nov. 2, 2022. [Online]. Available: `https://github.com/agda/agda/issues/2465`.

[18] Python Software Foundation, *Documentation Strings*, version 3.10.7, Nov. 21, 2022. [Online]. Available: `https://docs.python.org/3.10/tutorial/controlflow.html#documentation-strings`.

[19] S. L. P. Jones, "Haskell 98: Lexical structure," *J. Funct. Program.*, vol. 13, no. 1, pp. 7–16, 2003. DOI: `10.1017/S0956796803000418`. [Online]. Available: `https://doi.org/10.1017/S0956796803000418`.

[20] S. L. P. Jones, "Haskell 98: Syntax reference," *J. Funct. Program.*, vol. 13, no. 1, pp. 125–138, 2003. DOI: `10.1017/S0956796803001114`. [Online]. Available: `https://doi.org/10.1017/S0956796803001114`.

[21] "Programming guidelines." (Apr. 25, 2023), [Online]. Available: `https://wiki.haskell.org/Programming_guidelines`.

[22] S. L. P. Jones, "Haskell 98: Syntax reference," *J. Funct. Program.*, vol. 13, no. 1, pp. 125–138, 2003. DOI: `10.1017/S0956796803001114`. [Online]. Available: `https://doi.org/10.1017/S0956796803001114`.

[23] G. Team, *Overview of all language extensions*, Apr. 5, 2023. [Online]. Available: `https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/table.html?highlight=nondecreasingindentation`.

[24] I. Spaces, *Yaml language development team*, Apr. 6, 2023. [Online]. Available: `https://yaml.org/spec/1.2.2/#61-indentation-spaces`.