Principled Parsing for Indentation-Sensitive Languages

Revisiting Landin’s Offside Rule

Michael D. Adams
Portland State University
http://michaeldadams.org/

Abstract

Several popular languages, such as Haskell, Python, and F#, use the indentation and layout of code as part of their syntax. Because context-free grammars cannot express the rules of indentation, parsers for these languages currently use ad hoc techniques to handle layout. These techniques tend to be low-level and operational in nature and forgo the advantages of more declarative specifications like context-free grammars. For example, they are often coded by hand instead of being generated by a parser generator.

This paper presents a simple extension to context-free grammars that can express these layout rules, and derives GLR and LR(k) algorithms for parsing these grammars. These grammars are easy to write and can be parsed efficiently. Examples for several languages are presented, as are benchmarks showing the practical efficiency of these algorithms.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory—Syntax; D.3.4 [Programming Languages]: Processors—Parsing; F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems—Parsing

General Terms Algorithms, Languages

Keywords Parsing, Indentation, Offside rule

1. Introduction

Languages such as Haskell [Marlow (ed.) 2010] and Python use the indentation of code to delimit various grammatical forms. In Haskell, the contents of a let, where, do, or case expression can be indented relative to the surrounding code instead of being explicitly delimited by curly braces. In Python, the body of a multi-line function or compound statement must be indented relative to the surrounding code; there is no alternative, explicitly-delimited syntax. For example, in Haskell one may write:

```haskell
mapAccumR f = loop
    where loop acc (x:xs) = (acc', x' : xs')
          where (acc', x') = f acc' x
                  (acc', xs') = loop acc xs
    loop acc [] = (acc, [])
```

The indentation of the bindings after each where keyword determines the parse structure of this code. For example, the indentation of the last line determines that it is part of the bindings introduced by the first where instead of the second where.

Likewise, in Python one may write:

```python
def factorial(x):
    result = 1
    for i in range(1, x + 1):
        result = result * i
    return result
print factorial(5)
```

Here the indentation determines that the for loop ends before the return and the factorial function ends after the return.

While Haskell and Python are well known for being indentation-sensitive languages, quite a few other languages also use indentation. Landin’s ISWIM [Landin 1966] introduced the concept of the offside rule for indentation, which requires that all tokens in an expression be indented at least as far as the first token of the expression. Variations on this rule are used by Haskell, Miranda [Turner 1985], occam [INMOS Limited 1984], Orwell [Wadler 1985], Curry [Hanus (ed.) 2006], and Habit [HASP Project 2010].

F# is indentation sensitive when its lightweight syntax is enabled. Syntax from the language specification. For example, the layout is thus often left to ad hoc, handwritten code.

Whitespace sensitivity may be controversial, but regardless of whether it is a good idea from a language design perspective, it is important that the grammars of layout-sensitive languages be precisely specified. Unfortunately, many language specifications are informal in their description of layout or use formalisms that are not amenable to practical implementation. The task of parsing layout is thus often left to ad hoc, handwritten code.

The lack of a standard formalism for expressing these layout rules and of parser generators for such a formalism increases the complexity of writing parsers for these languages. Often, practical parsers for these languages have significant structural differences from the language specification. For example, the layout rule for Haskell is specified in terms of an extra pass between the lex and the parser that inserts explicit delimiters. This extra pass uses information about whether the parsing that occurs later in the pipeline will succeed or fail on particular inputs. Due to the resulting cyclic dependency, Haskell implementations do not actually structure their parsers this way. As a result, the structural differences between the implementation and the specification make it difficult to determine if one accurately reflects the other.
This paper aims to resolve this situation by proposing a grammar formalism for expressing layout rules. This formalism is both theoretically sound and practical to implement. Indentation-sensitive grammars are easy and convenient to write, and fast and efficient parsers can be implemented for them. The primary contributions of this paper are:

- a grammar formalism for expressing indentation-sensitive languages, which is informally described in Section 2 and formally defined later in Section 4.
- a demonstration in Section 3 of the expressiveness of these grammars by showing how to express the layout rules of ISWIM, Miranda, Haskell, and Python in terms of these grammars;
- a development in Section 5 of GLR and LR(k) parsing algorithms for these grammars, which is possible by a careful factoring of item sets into state and indentation sets; and
- a demonstration in Section 6 of the practical performance of these parsing techniques relative to existing ad hoc techniques.

Section 7 of this paper reviews related work. Section 8 concludes.

Note that Section 5 of this paper assumes a fair amount of familiarity with standard parsing techniques and in particular the work by Knuth [1965] on LR(k) parsing. Other than Section 5, however, this paper assumes only a basic knowledge of context-free grammars.

2. The Basic Idea

In order to support indentation-sensitive parsing, we use a modification of traditional, context-free grammars. We parse over a sequence of terminals where every terminal is annotated with the column at which it occurs in the source code. We call this its indentation. During parsing, we also annotate each non-terminal with an indentation. The grammar specifies a numerical relation that the indentation of each non-terminal must have with the indentation of its immediate children. These relations are usually chosen so the indentation of a non-terminal is the minimum column at which any token in the non-terminal is allowed to occur. Thus, the indentation of a non-terminal usually coincides with the intuitive notion of how far a block of code is indented. Formally, however, the indentation of a non-terminal has no meaning other than that it must appropriately relate to the indentation of the non-terminal’s children.

We call these grammars indentation-sensitive context-free grammars (IS-CFG) to contrast them with traditional indentation-insensitive context-free grammars (II-CFG). Section 3 gives examples of IS-CFGs for real world languages, and Section 4 formally defines this class of grammars.

As a simple example, we may write \( A \rightarrow \left( ( m \cdot A^n \cdot )^n \right)^n \) to mean that ( and ) must be at the same indentation as the A on the left of the production arrow but the A on the right must be at a greater indentation. We may also write \( A \rightarrow \left( [ \geq A^n \cdot ]^n \right)^n \) to mean the same except that [ and ] must be at a greater or equal indentation than the A on the left of the production arrow. In addition, we may write \( A 

### Figure 1. Example IS-CFG parse trees for \( (X^i)^2 \) and \( X^i \).
3.1 ISWIM and Miranda

With ISWIM, Landin [1966] introduced several innovations still used in programming languages today. Among these is the use of indentation to indicate program structure. Landin defines an "offside rule" that specifies that:

The southeast quadrant that just contains the phrase’s first symbol must contain the entire phrase, except possibly for {

This rule requires that all the tokens in a particular non-terminal be at a column that is at least as far right (i.e., in the southeast quadrant) as the column of the first token.

The IS-CFG for ISWIM is similar to the II-CFG for ISWIM that ignores the offside rule except that we annotate each production of the IS-CFG with appropriate indentation relations. As an example, consider the productions in Figure 2 where we have applied the offside rule to only the right-hand side of where clauses. Note that the productions for expr are created automatically from the productions for expr as specified in Section 3. We do likewise for the terminals. Using this grammar to parse the expression

\[
\begin{align*}
x + v \ &\text{where} \\
y + z \ &+ \ w
\end{align*}
\]

results in the parse tree in Figure 3.

For productions that do not involve the offside rule, such as the productions for addition and negation, we annotate the non-terminals with = and the terminals with ≥. This means that these productions do not change the current indentation and terminals are allowed at any column greater than or equal to the current indentation. For example, the expressions for \( x + v, (y + z), \) and all the variable references are all at the same indentation as their respective parents. For those expressions, no special indentation rule is in effect and we simply use = to propagate the indentation from parent to child. The terminals use ≥ so they can be at that or any greater indentation. If a non-terminal is too far left (e.g., if \( w \) was at column 5), then the ≥ constraint is violated and the code would be rejected. This is a common pattern that we will see in other grammars as well.

Next, consider forms that involve the offside rule. Wherever a non-terminal \( A \) should trigger the offside rule, we simply use \( |A| ≥ \) instead of \( A \). An example is the right-hand side of a where clause where we use \( |expr| ≥ \) instead of expr. Because we use ≥ instead of =, the right-hand side is allowed to be at a greater indentation than its parent. We see this in the example parse tree where the right-hand side of the where clause has an indentation of 6 while its parent has an indentation of 1. If the top-level expr were at indentation 7, however, such a parse with a right-hand side at indentation 6 would be rejected. We also use expr instead of expr so that the first token of the right-hand side of the where clause has the same indentation as the entire right-hand side. As a consequence, in the example parse tree, the indentations on the path from the right-hand side of the where to the ‘;’ are all exactly 6.

Finally, since parenthesized expressions are exempt from the indentation constraints of the surrounding code, the production for parenthesized expressions uses expr instead of expr. This frees

\[
\text{Productions written by the user:}
\]

\[
\begin{align*}
\text{expr} &\rightarrow \text{expr} \ where \ \geq \ ID \ \geq \ v \ \geq \ |expr| \ \geq \\
\text{expr} &\rightarrow \text{expr} \ where \ ID \ \geq \ expr \ \geq \\
\text{expr} &\rightarrow \ expr \ \geq \ expr \ \geq \\
\text{expr} &\rightarrow \ ID \ \geq
\end{align*}
\]

\[
\text{Productions added by desugaring:}
\]

\[
\begin{align*}
\text{expr} &\rightarrow \text{expr} \ where \ ID \ \geq \ expr \ \geq \\
\text{expr} &\rightarrow \text{expr} \ where \ expr \ \geq \\
\text{expr} &\rightarrow \ expr \ \geq \\
\text{expr} &\rightarrow \ ID
\end{align*}
\]

the indentation of the expression inside the parentheses from any indentation constraints coming from the context of the expression. This is used in the example parse tree where the \( y + z \) expression has an indentation of 0 even though the \( (y + z) \) expression has an indentation of 6. Uses of the offside rule inside the parentheses still have effect, of course.

The Miranda language uses the same offside rule as ISWIM except for two differences. The first is that expressions inside parentheses are subject to indentation constraints imposed by the context outside the parentheses. The production for parenthesized expressions thus uses = instead of ≥ and is simply:

\[
\text{expr} \rightarrow \ expr \ where \ \geq \ expr \ \geq
\]

The second difference is that the language specification adopts the notational convention that for any non-terminal \( x \),\n
\[
x(\); means that \( x \) is followed by an optional semicolon and is subject to the offside rule..., so that every token of \( x \) must lie below or to the right of the first. Provided the layout makes it clear where \( x \) terminates, the trailing semicolon may be omitted. \[Turner1989, \S25\]

This notation is easily handled by introducing, for each non-terminal \( A \), the non-terminal \( A(\;) \) and the two productions \( A(\;) \rightarrow |A| ≥ \) and \( A(\;) \rightarrow |A| ≥ \ where \).
3.2 Haskell

3.2.1 Language

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

While the indentation rules of Haskell are intuitive to use in practice, the way that they are formally expressed in the Haskell language specification [Marlow (ed.) 2010 §10.3] is not nearly so intuitive. The indentation rules are specified in terms of both the lexer and an extra pass between the lexer and the parser. Roughly speaking, the lexer inserts special `\{n\}` tokens where a new block might start and special `<n>` tokens where a new clause within a block might start. The extra pass then translates these tokens into explicit semicolons and curly braces.

The special tokens are inserted according to the following rules:

- If a `let`, `where`, `do`, or `of` keyword is not followed by the lexeme `\{`, the token `\{n\}` is inserted after the keyword, where `n` is the indentation of the next lexeme if there is one, or 0 if the end of file has been reached.
- If the first lexeme of a module is not `\{` or `module`, then it is preceded by `\{n\}` where `n` is the indentation of the lexeme.
- Where the start of a lexeme is preceded only by white space on the same line, this lexeme is preceded by `<n>`, where `n` is the indentation of the lexeme, provided that it is not, as a consequence of the first two rules, preceded by `\{n\}`. [Marlow (ed.) 2010 §10.3]

Between the lexer and the parser, an indentation resolution pass converts the lexeme stream into a stream that uses explicit semicolons and curly braces to delimit clauses and blocks. The stream of tokens from this pass is defined to be `L tokens [\]` where `tokens` is the stream of tokens from the lexer and `L` is the function in Figure 4. Thus the context-free grammar has to deal with only semicolons and curly braces. It does not deal with layout.

This `L` function is fairly intricate, but the key clauses are the ones dealing with `<n>` and `{n}`. After a `let`, `where`, `do`, or `of` keyword, the lexer inserts a `{n}` token. If `n` is a greater indentation than the current indentation, then the first clause for `{n}` executes, an open brace (`{`) is inserted, and the indentation `n` is pushed on the second argument to `L` (i.e., the stack of indentations). If a line starts at the same indentation as the top of the stack, then the first clause for `<n>` executes and a semicolon (`;`) is inserted to start a new clause. If it starts at a smaller indentation, then the second clause for `<n>` executes and a close brace (`}`) is inserted to close the block started by the inserted open brace. Finally, if the line is at a greater indentation, then the third clause executes, no extra token is inserted, and the line is a continuation of the current clause. The effect of all this is that `\{`, `;`, and `}` tokens are inserted wherever layout indicates that blocks start, new clauses begin, or blocks end, respectively. The other clauses in `L` handle a variety of other edge cases and scenarios.

Note that `L` uses `parse-error` to signal a parse error, but uses `parse-error(t)` as an oracle that predicts the future behavior of the parser that runs after `L`. Specifically,

```
if the tokens generated so far by L together with the next token \t represent an invalid prefix of the Haskell grammar, and the tokens generated so far by L followed by the token “\t” represent a valid prefix of the Haskell grammar, then parse-error(t) is true. [Marlow (ed.) 2010 §10.3]
```

This handles code such as

```
let x = do f; g in x
```

where the block starting after the do needs to be terminated before the `in`. This requires knowledge about the parse structure in order to be handled properly, and thus `parse-error(t)` is used to query the parser for this information.

In addition to the operational nature of this definition, the use of the `parse-error(t)` predicate means that `L` cannot run as an independent pass; its execution must interact with the parser. In fact, the Haskell implementations GHC [GHC 2011] and Hugs [Jones 1994] do not use a separate pass for `L`. Instead, the lexer and parser share state consisting of a stack of indentations. The parser accounts for the behavior of `parse-error(t)` by making close braces optional in the grammar and appropriately adjusting the indentation stack when braces are omitted. The protocol relies on “some mildly complicated interactions between the lexer and parser” [Jones 1994] and is tricky to use. While preparing the parser in Section 4, we found that even minor changes to the error propagation of the parser affected whether syntactically correct programs were accepted by this style of parser.

While we may believe the correctness of these parsers based on their many years of use and testing, the significant and fundamental structural differences between their implementation and the language specification are troubling.
3.2.2 Grammar

Haskell’s layout rule is more complicated than those of ISWIM and Miranda, but is also easily specified as an IS-CFG. By using an IS-CFG there is no need for an intermediate \( L \) function, and the lexer and parser can be cleanly separated into self-contained passes. The functionality of \( \text{parse-error}(t) \) is simply implicit in the structure of the grammar.

Figure 5 shows example productions for case expressions. For productions that do not change the indentation, we annotate non-terminals with a default indentation relation of = and terminals with a default indentation relation of \( \geq \). We use \( > \) instead of \( \geq \) because Haskell distinguishes tokens that are at an indentation equal to the current indentation from tokens that are at a strictly greater indentation. The former start a new clause while the latter continue the current clause.

In Haskell, a block can be delimited by either explicit curly braces or use of the layout rule. In Figure 5 this is reflected by the two different productions for \( \text{altBlock} \). If \( \text{altBlock} \) expands to \( '{' \, \text{alts} \, '}' \), then the \( \oplus \) relation allows \( \text{alts} \) to respect the indentation constraints from the surrounding code. Since Haskell’s layout rule allows closing braces to occur at any column, we use \( \oplus \) instead of the usual \( \geq \) on \( '}' \). On the other hand, if \( \text{altBlock} \) expands to \( \text{altLayout}^* \), then the \( > \) relation increases the indentation. In the productions for \( \text{altLayout} \), the use of \( \text{alts} \) instead of \( \text{alts}^\oplus \) ensures that the first tokens of the \( \text{alts} \) all align to the same column. Note that within an \( \text{alts} \), each \( \text{alt} \) must be separated by a semicolon (\( ; \)). Thus, because \( \text{altLayout} \) refers to \( \text{alts} \) instead of \( \text{alts}^\oplus \), each instance of \( \text{alt} \) can be separated using either layout or a semicolon. When using curly braces to explicitly delimit a block, semicolons must always be used.

Other grammatical forms that use the layout rule follow the same general pattern as case with only minor variation to account for differing base cases (e.g., let uses \( \text{decl} \) in place of \( \text{alt} \)) and structures (e.g., a do block is a sequence of \( \text{stmt} \) ending in an \( \text{exp} \)).

A subtlety of Haskell’s layout rule is that tokens on the same line as, but after, a closing brace may not have to respect the current indentation. This is because the \( L \) function considers the indentation of only the first token of a line (i.e., where \( \langle n \rangle \) is inserted) and tokens after a \( \text{let} \), \( \text{where} \), \( \text{do} \) or of keyword (i.e., where \( \langle n \rangle \) is inserted). One might view this as an artifact of how the language specification uses \( L \) to define layout, but this aspect of Haskell’s layout rule is still expressible by having the lexer annotate tokens whose indentation is to be ignored with an indentation of infinity \( \ominus \). Since terminals have an indentation relation of \( > \), the infinite indentation of these tokens will always match. We have the lexer handle this instead of the parser because it is the linear order of tokens instead of the grammatical structure of the syntax that controls what tokens are indentation sensitive. For example, the token after the \( \text{do} \) keyword is indentation sensitive regardless of the structure of the expression following the \( \text{do} \). This requires the lexer to maintain a bit of extra state indicating whether we are at the start of a line or after a \( \text{let} \), \( \text{where} \), \( \text{do} \) or of keyword, but this is a fairly light requirement as the lexer is presumably already tracking state to determine the column of each token.

Finally, GHC also supports an alternative indentation rule that is enabled by the \texttt{RelaxedLayout} extension. It allows opening braces to be at any column regardless of the current indentation [\texttt{GHC} [2011] §15.2]. This is easily implemented by changing the first production for \( \text{altBlock} \) to be:

\[
\text{altBlock} \rightarrow '{' \, \text{alts}^\oplus \, '} \, \ominus
\]

This assumes no tokens are at column 0, which we reserve for this purpose.

3.3 Python

3.3.1 Language

Python represents a different approach to specifying indentation sensitivity. It is explicitly line oriented and features \texttt{NEWLINE} in its grammar as a terminal that separates statements. The grammar uses \texttt{INDENT} and \texttt{DEDENT} tokens to delimit indentation-sensitive forms. An \texttt{INDENT} token is emitted by the lexer whenever the start of a line is at a strictly greater indentation than the previous line. Matching \texttt{DEDENT} tokens are emitted when a line starts at a lesser indentation.

In Python, indentation is used only to delimit statements, and there are no indentation-sensitive forms for expressions. This, combined with the simple layout rules, would seem to make parsing Python much simpler than for Haskell, but Python has line joining rules that complicate matters.

Normally, each new line of Python code starts a new statement. If, however, the preceding line ends in a backslash (\( \backslash \)), then the current line is “joined” with the preceding line and is a continuation of the preceding line. In addition, tokens on this line are treated as if they had the same indentation as the backslash itself.

Python’s \texttt{explicit} line joining rule is simple enough to implement directly in the lexer, but Python also has an \texttt{implicit} line joining rule. Specifically, expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes.

The indentation of the continuation lines is not important.

\texttt{Python} [§2.1.6]

This means that \texttt{INDENT} and \texttt{DEDENT} tokens must not be emitted by the lexer between paired delimiters. For example, the second line of the following code should not emit an \texttt{INDENT} and the indentation of the third line should be compared to the indentation of the first line instead of the second line.

\begin{verbatim}
x = [
    y ]
z = 3
\end{verbatim}

Thus, while the simplicity of Python’s indentation rules is attractive, they contain hidden complexity that requires interleaving the execution of the lexer and parser.

3.3.2 Grammar

Though Python’s specification presents its indentation rules quite differently from Haskell’s specification, once we translate it to an IS-CFG, it shares many similarities with that of Haskell. The lexer still needs to produce \texttt{NEWLINE} tokens, but it does not produce \texttt{INDENT} or \texttt{DEDENT} tokens. As with Haskell, we start with a grammar where the non-terminals and terminals are annotated with indentation relations of \( = \) and \( \geq \), respectively.

In Python, the only form that changes indentation is the \texttt{suite} non-terminal, which represents a block of statements contained inside a compound statement. For example, one of the productions for while is:

\begin{verbatim}
while_stmt → 'while' \texttt{test}' ':'\texttt{suite}'
\end{verbatim}

A \texttt{suite} has two forms. The first is for a single-line statement and is the same as with the standard Python grammar. The second is for multi-line statements. The following productions handle both of these two cases:

\begin{verbatim}
suite → stmt_list\texttt{NEWLINE} 
suite → \texttt{NEWLINE} block 
block → block\texttt{NEWLINE} \texttt{statement} 
block → \texttt{statement}
\end{verbatim}
When a suite is of the multi-line form (i.e., using the second production), the initial NEWLINE token ensures that the suite is on a separate line from the preceding header. The block inside a suite must then be at some indentation greater than the current indentation. Such a block is a sequence of statement forms that all start with their first token at the same column. In Python’s grammar, the productions for statement already include a terminating NEWLINE, so NEWLINE is not needed in the productions for block.

For implicit line joining, we employ the same trick as for parenthesized expressions in ISWIM and braces in Haskell. For any production that contains parentheses, square brackets or curly braces, we annotate the part contained in the delimiters with the \(\circ\) indentation relation. Since the final delimiter is also allowed to appear at any column, we annotate it with \(\circ\). For example, one of the productions for list construction becomes:

\[
\text{atom} \rightarrow \text{'}[^{*}\text{listmaker*}]\text{'}
\]

There remain a few subtleties with Python’s line joining rules that we must address. First, as with Haskell, tokens after a closing delimiter can appear at any column. For example, the following code is validly indented according to Python’s rules:

```python
while True:
x = 1 + (2 * 3)
```

To handle this we use the same trick as for Haskell and annotate tokens that are not at the start of a line with an infinite indentation.

Second, while a lexer based on regular expressions can detect the start of a line and thus produce finite indentations for the first token of a line but infinite indentations for other tokens, it cannot detect matching parentheses to determine that NEWLINE tokens should be omitted inside delimited forms. Thus non-terminals that occur inside delimited forms need to allow the insertion of NEWLINE tokens at arbitrary locations. This may mean there have to be two forms of a non-terminal (i.e., for expressions inside versus outside a delimited form), but this is a fairly mechanical transformation that can be automated by the use of syntactic sugar similar to the syntactic sugar for \(\backslash a\). Alternatively, it may be possible to use a grammar that does not use NEWLINE tokens at all and instead, like for Haskell, uses vertical alignment to delimit statements.

Finally, as with the standard Python parser, the lexer still handles the explicit line joining that is triggered by a line ending in a backslash (\(\backslash\)). It gives the tokens of an explicitly joined line the same indentation as the backslash itself, and the backslash is not emitted as a token.

### 3.4 Conventions and syntactic sugar

In an IS-CFG, every symbol in every production must be annotated with an indentation relation. In many indentation-sensitive languages, however, productions often allow terminals to appear at any indentation greater than the current indentation but do not themselves change the current indentation. Thus we can simplify the job of writing an IS-CFG by adopting the convention that if a symbol on the right-hand side of a production is not explicitly annotated with an indentation relation, then it implicitly defaults to \(\circ\) if it is a non-terminal and \(\triangleright\) if it is a terminal. For example, with this convention, the only productions in Figure 5 that need explicit annotations are those for \(\backslash a\backslash t\backslash B\backslash l\o\backslash c\k\). All other productions simply use the defaults. Using this convention most productions in a grammar do not have to be annotated with indentation relations. They thus look like ordinary II- CFG productions, and only the forms that explicitly deal with indentation must be explicitly annotated.

In addition, just as II- CFGs often allow the use of alternation bars (\(\mid\)) or Kleene stars (\(*\)) to simplify writing grammars, it is often convenient to allow symbols on the right-hand side of a production to be annotated with a composition of indentation relations. Thus we might write \(A \rightarrow B\circ\) instead of the more verbose

\[
A \rightarrow B\circ
\]

These conventions are merely notational conveniences and do not affect the fundamental theory.

### 4. Indentation-Sensitive Grammars

The formalism for IS-CFGs that this paper proposes is an extension of II- CFGs. Thus to review the standard definition of II-CFGs, recall that a grammar is a four-tuple \(G = (N, \Sigma, \delta, S)\) where \(N\) is a finite set of non-terminal symbols, \(\Sigma\) is a finite set of terminal symbols, \(\delta\) is a finite production relation, and \(S \in N\) is the start symbol. The relation \(\delta\) is a subset of \(N \times (N \cup \Sigma)^*\), and we write \(A \rightarrow X_1X_2\cdots X_n\) for a tuple \((A, X_1, X_2\cdots X_n)\) that is an element of \(\delta\). As a notational convention let \(A, B, C\) be elements of \(N\), let \(a, b, c\) be elements of \(\Sigma\), and let \(X, Y, Z\) be elements of \(N\cup\Sigma\). Let \(U, V, W\) be elements of \((N \cup \Sigma)^*\), and \(u, v, w\) be elements of \(\Sigma^*\).

We define a rewrite relation \(\Rightarrow \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*\) such that \(UAV \Rightarrow U(X_1X_2\cdots X_n)V\) iff \(A \rightarrow X_1X_2\cdots X_n\). We define \(\Rightarrow^*\) as the reflexive, transitive closure of \(\Rightarrow\).

The language recognized by a grammar is then defined as \(L(G) = \{w \in \Sigma^* | S \Rightarrow^* w\}\) and is the set of words reachable by the rewrite relation from the start symbol.

An IS- CFG is also a four-tuple, \(G = (N, \Sigma, \delta, S)\), except that \(\delta\) and \(S\) account for indentations. \(S\) is an element of \(N \times N\) and records the indentation of the initial non-terminal. The production relation, \(\delta\), is an element of \(N \times ((N \cup \Sigma) \times \mathbb{N})\) where \(\mathbb{N}\) is the domain of indentation relations and each indentation relation is a subset of \(N \times \mathbb{N}\). In principle, these indentation relations can be any subset of \(N \times \mathbb{N}\), but for our purposes we restrict \(I\) to the relations \(=, >, \geq\) and \(\circ\).

Here and in the remainder of this paper, we restrict ourselves to finite indentations, but everything generalizes straightforwardly to languages with infinite indentations.

As a notational convention, let \(i, j, k\) and \(l\) be indentations and \(\triangleright\) be an indentation relation. For the sake of compactness, we adopt the notations \(X^i\) and \(X^*\), respectively, for a pair of \(X\) and either an indentation \(i\) or an indentation relation \(\triangleright\). Thus we write \(A \rightarrow X_1^iX_2^i\cdots X_n^i\) for a tuple \((A, (X_1, \triangleright_1) (X_2, \triangleright_2)\cdots (X_n, \triangleright_n))\) that is an element of \(\delta\).

As with II- CFGs, we define a rewrite relation

\[
\Rightarrow^* \subseteq ((N \cup \Sigma) \times \mathbb{N})^* \times ((N \cup \Sigma) \times \mathbb{N})^*
\]

where \(UA^iV \Rightarrow UX_1^{i_1}X_2^{i_2}\cdots X_n^{i_n}V\) iff \(A \rightarrow X_1^{i_1}X_2^{i_2}\cdots X_n^{i_n}\) and \(j_1 \triangleright_1 i, j_2 \triangleright_2 i, \ldots, j_n \triangleright_n i\). The \(\Rightarrow^*\) relation, the language \(L(G)\), derivations, and parse trees are all defined as with II- CFGs except that they are in terms of this new rewrite relation.

Note that every II-CFG is encodable as an IS-CFG by translating every production \(A \rightarrow X_1X_2\cdots X_n\) to \(A \rightarrow X_1^\circ X_2^\circ\cdots X_n^\circ\) and every word \(a_1a_2\cdots a_m\) to \(a_1^\circ a_2^\circ\cdots a_m^\circ\) with arbitrary \(1, 2, \ldots, m \in \mathbb{N}\). Conversely, erasing the indentations and indentation relations in an IS-CFG results in an II-CFG. Note that translating from an II-CFG to an IS-CFG will not introduce ambiguities, but translating from an IS-CFG to an II-CFG might.

### 5. Parsing

Of course, a grammar is not practically useful if we cannot effectively parse with it. In this section, we show how to modify traditional parsing techniques for II-CFGs to handle IS-CFGs. We show this for both GLR and LR(k) parsing. This can also be done for
The first step of our approach is to model the IS-CFG by an infinite II-CFG. Of course we do not actually compute with this infinite grammar, but it provides a mathematical model from which we derive a computable parsing algorithm. Given an IS-CFG \( G = (N, \Sigma, \delta, S) \), this II-CFG is \( G' = (N', \Sigma', \delta', S') \) where:

\[
N' = N \times \mathbb{N} \\
\Sigma' = \Sigma \times \mathbb{N} \\
\delta' = \left\{ \begin{array}{l}
A_i \rightarrow X_1^i; X_2^i; \ldots; X_n^i \\
|A| \rightarrow X_1^i; X_2^i; \ldots; X_n^i \in \delta, \\
i, j_1, j_2, \ldots, j_n \in \mathbb{N}, \\
j_1 \vartriangleleft j_2 \vartriangleleft \cdots \vartriangleleft j_n \vartriangleleft i \end{array} \right\}
\]

\( S' = S \)

This grammar has an infinite number of non-terminals, terminals, and productions per non-terminal, but we still limit derivations to finite lengths.

Note that traditional parsing algorithms on this grammar may not terminate due to the infinite size of \( G' \), so we formally model \( G' \) as the limit of successive approximations where each approximation bounds the maximum indentation in any non-terminal, terminal or production to successively greater values. We gloss over this detail in the remainder of this paper.

**Lemma 6 (Equivalence).** \( S \Rightarrow^* W \) for \( G \) iff \( S' \Rightarrow^* W' \) for \( G' \).

**Proof.** By induction on the number of reductions and the fact that for all \( W \) and \( W' \), \( W \Rightarrow W' \) in \( G \) iff \( W \Rightarrow W' \) in \( G' \).

### 5.3 Rewrite system

In this subsection, we consider LR(k) parsing in terms of a rewrite system that concisely specifies what it means for a parser to be LR(k). In later subsections, we derive more conventional stack-based parsing algorithms. In this development, we closely follow the original presentation of LR(k) parsing by Knuth [1965] with only minor changes to use current notational conventions.

Recall that an LR(k) parser is one that always produces a rightmost derivation, and a rightmost derivation is one in which the rightmost non-terminal is always expanded before any other non-terminals. The symbols resulting from such an expansion step are called the handle. For example, if

\[
S \Rightarrow^* UXV \Rightarrow UX_1^i \cdots X_p^j V \Rightarrow^* W
\]

is a rightmost derivation, then a handle of \( UX_1^i \cdots X_p^j V \) is \( Y_1^i \cdots Y_p^j \). Note that in order for this to be a rightmost derivation, \( V \) necessarily contains only elements of \( \Sigma \), though \( U \) and \( X_1^i \cdots X_p^j \) may contain elements of both \( \Sigma \) and \( N \).

Since an LR(k) parser works from the result of a rightmost derivation back to the start symbol, LR(k) parsing can be accomplished by iteratively searching for the handle of a string and performing the appropriate reduction. Again following Knuth [1965], given the infinite II-CFG \( G' = (N', \Sigma', \delta', S') \) for recognizing prefixes that end in a handle. Here the non-terminals are

\[
N'' = \left\{ \left[ A_i \atop a_1^i a_2^i \cdots a_k^i \right] \left| A_i \in N', a_1^i, a_2^i, \ldots, a_k^i \in \Sigma' \right. \right\}
\]

and represent the part of the string that contains the handle. The \( a_1^i a_2^i \cdots a_k^i \) track the \( k \) terminals expected after the handle and
are the lookahead. For each $A^i \rightarrow X_1^j \cdots X_m^j \cdots X_n^j \in \delta'$ and each $u = a_1^i a_2^i \cdots a_k^i$, we include the following in $\delta''$:

$$
\begin{align*}
A^i, u & \rightarrow X_1^j \cdots X_n^j u \\
A^i, u & \rightarrow X_1^j \cdots X_m^j X_{m+1}^{j+1}, v \\
& \text{for each } v \in H_k \left(X_{m+1}^{j+1} \cdots X_n^j u\right)
\end{align*}
$$

Here, $u$ and $v$ are the lookahead expected by the parser. $H_k(W)$ computes such lookaheads by computing the $k$-length prefixes of $\mathcal{L}(W)$ and is defined as

$$
H_k(W) = \left\{ a_1^1 a_2^1 \cdots a_k^1 | W \Rightarrow^* a_1^1 a_2^1 \cdots a_k^1 U \right\}
$$

where the reduction relation $\Rightarrow^*$ is for $G'$.

The intuition here is that the first production expands to the handle along with a lookahead string and the second production expands to an intermediate non-terminal that in turn eventually expands to the handle.

Since this grammar is regular, it can be implemented by a state machine [BZ], which leads to the following rewrite based algorithm for parsing.

**Algorithm 7.** Given input string $W$, if $S = W$ then stop and accept the string. Otherwise, find all prefixes of $W$ that match the regular language $\mathcal{L}(G')$. If there are no such matches, then reject the string. Otherwise, non-deterministically choose one of the matches, and let the last production of the match be

$$
A^i, u \rightarrow X_1^{j_1} X_2^{j_2} \cdots X_n^{j_n} u
$$

Replace this occurrence of $X_1^{j_1} X_2^{j_2} \cdots X_n^{j_n}$ in $W$ with $A^i$ as it is a handle of $W$, and repeat the algorithm with the new value of $W$.

Note that this algorithm is non-deterministic and accepts the word if any path through the algorithm accepts the word. We allow this because productions in $G$ (e.g., $A \rightarrow B^i$ and $A \rightarrow C^i$) may produce multiple productions when translated to $G'$ (e.g., all $A^i \rightarrow B^j$ and $A^i \rightarrow C^k$ such that $k > i$). These may introduce ambiguities in $G'$ even when there are no ambiguities in $G$. Once we convert to a finite version of the parsing algorithm, we will eliminate this non-determinism.

### 5.4 Parsing with stacks

Of course, rewriting the entire string and restarting the automaton from the start as done in Algorithm 7 is inefficient. Instead, we can save a trace of the states visited. When a handle is reduced, we rewind to the state just before the first symbol of the handle and proceed from there. This is the essential idea behind the traditional LR(k) parser development by Knuth [1963]. We apply this idea to our infinite II-CFG to obtain the following construction.

We begin with the notion of an item. We denote an item by

$$
A^i \rightarrow X_1^j \cdots X_m^j \cdot X_{m+1}^{j+1} \cdots X_n^j \Rightarrow u
$$

where $A^i \rightarrow X_1^j \cdots X_n^j$ is a production in $G'$ and $u \in (\Sigma^*)^k$ is the lookahead. The algorithm maintains a stack of sets of items $S_0 S_1 \cdots S_n$, where $S_k$ is the top element of the stack. We use the notation $S_0 S_1 \cdots S_n \Leftrightarrow a a_2 \cdots a_k w$ to denote that $S_0 S_1 \cdots S_n$ is the current stack and $a a_2 \cdots a_k w$ is the input remaining to be consumed by the parser.

To parse a word $w$, we start with the configuration

$$
S_0 \Leftrightarrow w \Leftrightarrow^0 \Leftrightarrow^0 \cdots \Leftrightarrow^0
$$

where $S_0 = \{ \hat{S} \Rightarrow S'; \Leftrightarrow^0 \Leftrightarrow^0 \cdots \Leftrightarrow^0 \}$. We let $\hat{S}$ be a fresh non-terminal and $\Leftrightarrow^1, \Leftrightarrow^2, \cdots, \Leftrightarrow^k$ be fresh terminals that pad the string to have at least $k$ tokens of lookahead. We then run the following parsing algorithm.

**Algorithm 8.** Given configuration $S_0 S_1 \cdots S_n \Leftrightarrow a_1 a_2 \cdots a_k w$, if $\hat{S} \Rightarrow S'; \Leftrightarrow^0 \Leftrightarrow^0 \cdots \Leftrightarrow^0 \in S_k$ and $a_1 a_2 \cdots a_k w = \Leftrightarrow^0 \Leftrightarrow^0 \cdots \Leftrightarrow^0 \Leftrightarrow^0$, then accept. Otherwise:

1. Compute the closure, $S'$, of $S_n$ where $S'$ is the least set of items satisfying the recurrence

$$
S' = S_n \cup \left\{ \left\{ A^i \rightarrow X_1^j \cdots X_m^j \cdot X_{m+1}^{j+1} \cdots X_n^j ; u \right\} \in S_n, X_{m+1}^{j+1} \rightarrow Y_{m+1}^j \cdots Y_n^j \in \delta' \\
v \in H_k \left(X_{m+1}^{j+1} \cdots X_n^j u\right) \right\}
$$

2. Compute the acceptable lookahead set $K$ where

$$
K = \left\{ u \left\{ A^i \rightarrow X_1^j \cdots X_m^j \cdot X_{m+1}^{j+1} \cdots X_n^j u \right\} \in S', \right\}
$$

3. For each production $A^i \rightarrow X_1^j \cdots X_n^j$ in $\delta'$, compute the acceptable lookahead set $K \left(A^i \rightarrow X_1^j \cdots X_n^j\right)$ where

$$
K \left(A^i \rightarrow X_1^j \cdots X_n^j \Rightarrow u \right) \Rightarrow^* \left\{ u \left\{ A^i \rightarrow X_1^j \cdots X_n^j ; u \right\} \in S' \right\}
$$

4. Let $GOTO \left(S, Z\right)$ =

$$
\left\{ A^i \rightarrow X_1^j \cdots X_m^j \cdot X_{m+1}^{j+1} \cdots X_n^j ; u \right\} \in S, X_{m+1}^{j+1} \Rightarrow Z \right\}
$$

and non-deterministically choose one of the following.

(a) If $a_1 a_2 \cdots a_k \in K$, then do a shift action by looping back to the start of the algorithm with the new configuration

$$
S_0 S_1 \cdots S_n GOTO \left(S_n, a_1 a_2 \cdots a_k w \right)
$$

(b) If $a_1 a_2 \cdots a_k \in K \left(A^i \rightarrow X_1^j \cdots X_n^j\right)$ for some production $A^i \rightarrow X_1^j \cdots X_n^j$; then do a reduce action by looping back to the start of the algorithm with the new configuration

$$
S_0 S_1 \cdots S_{n-m} GOTO \left(S_{n-m}, A^i \right) \Rightarrow a_1 a_2 \cdots a_k w
$$

### 5.5 Finite representations of stacks

Algorithm 8 contains both non-determinism and infinite sets. Here we depart from Knuth’s [1963] in order to eliminate these. Up to this point, our parser is simply a standard LR(k) parser, albeit on an infinite II-CFG, and we rely on the correctness of the standard LR(k) parsing algorithm for our correctness. From here forward, we ensure correctness by ensuring that our modified version of the algorithm models the same item sets as Algorithm 8, albeit using a more efficient representation.

As a first step, consider the sets of items that form the stack. Each item is of the form

$$
A^i \rightarrow X_1^j \cdots X_n^j \cdot X_{m+1}^{j+1} \cdots X_n^j ; u
$$

where $A^i \in N \times N, X_1^j, \cdots, X_n^j \in (N \cup \Sigma) \times N$, and $u \in (\Sigma \times N)^\infty$. Observe that the indeterminations to the left of the bullet,
stated in the following lemma.

This property is preserved by each loop through Algorithm 8, as $X$ preserves completeness.

The only indentations that are in the lookahead only because $C$ is in the $L$ look ahead and, in the general case, we consider a node to be an ancestor of itself so some of $C_1a_1, C_2a_2, \ldots, C_k$ or $B^\omega$ may actually be the same node.

The lookahead token $a_1^1$ is in the lookahead only because $A$ shares it with the ancestor $C_1$ at indentation $l_1$. The $\delta_{1,1}$ and $\delta_{1,2}$ relations record the possible indentation relations between $C_1$ and, respectively, $A$ and $a_1^1$. The second lookahead token $a_1^2$ is in the lookahead only because $C_1$, the common ancestor of the item and the first lookahead token, also shares with $a_2$ the common ancestor $C_2$ at indentation $l_2$. The $\delta_{2,1}$ and $\delta_{2,2}$ relations record the possible indentation relations between $C_2$ and, respectively, $C_1$ and $a_2$. And so on, until we reach the final ancestor, $C_k$, at indentation $l_k$. This ancestor has a minimum indentation at which it can occur so we use $\delta_{k+1}$ to record the indentation relation between $C_k$ and $B$, the start terminal.

The lookahead computation, $H_k$, that is defined in Section 5.3 must of course be modified to account for this representation. We omit this because, while conceptually simple, its formal definition is fairly intricate.
With this representation, the lookahead set is finitely represented so the only potentially infinite part remaining is the indentation set. However, as before, we can show that these sets are always presented so the only potentially infinite part remaining is the indentation set. 

Note that each item in an item set may have a different set of variable references (i.e., ID) and do blocks with vertically aligned indentations. For example, we may have an item set containing both ID and ID' which may come from the code:

\[
\text{do do x y}
\]

Note that y does not align with either the second do or the x, and thus this code should be rejected. Put another way, the valid lookaheads when looking ahead at the ID for y but before reducing x to expr are ID', ID', 'do' and 'do'\(^+\). The string should thus be rejected since ID', 'do' and 'do'\(^+\). The string that contains the lookahead for y, is not in that set. But the LR(1) lookahead set using the new representation is \{((\geq, =, \text{ID}) \text{ >}), ((\geq, =, \text{'do'} \text{ >})\}\. Since the current indentation is 7 and there exist l such that 7 \geq l, 2 = l, and l > 0, the string will not be immediately rejected.

This representation thus appears to over approximate the set of indentations for a particular lookahead. This means that in the non-deterministic choice at the end of Algorithm\[8\] there could be more reductions possible than there should be. However, as we restrict ourselves to LR(k) grammars, this turns out to not be a problem. This is because there are only four cases where these spurious reductions occur:

**Case 1.** There should be no reductions or shifts possible, but the approximation makes one extra reduction possible.

In this case, the parser should reject the program, but will instead reduce and continue parsing. However, this case happens only when some other item set further up the stack also checks the lookahead tokens. Though the string is not rejected immediately, it will be rejected once we reach that point in the stack. In our example, once x reduces to expr and then do x reduces to an expr that finally reduces to stmts, the lookahead set will be \{((\geq, =, \text{ID}) \text{ >}), ((\geq, =, \text{'do'} \text{ >})\}. Since the indentation at that point is 4 but the next token is ID', the parser will reject the program. The program might not be rejected as soon as we expect, but it is eventually rejected. This case may arise even when the grammar is LR(k).

**Case 2.** There should be no shifts or reductions possible, but the approximation makes two or more extra reductions possible.

The representation for lookaheads in Section 2.5 is designed so that every lookahead word that it represents comes from a valid parse. Thus if this representation generates multiple possible reductions, then there is some string that would also generate those multiple possible reductions with the original representation in Algorithm\[8\]. In that case, the grammar is not LR(k), and the grammar should be rejected by the parser generator.

**Case 3.** There should be one reduction or shift possible, but the approximation makes one or more extra reductions possible.

The same reasoning applies as in the preceding case.

**Case 4.** There should be two or more reductions or shifts possible.

Then the original grammar is not LR(k), and the grammar should be rejected by the parser generator.

This means that the representation in Section 2.5 is valid for any grammar that is LR(k), and furthermore we can detect when a grammar is not LR(k) by using this representation.

### 5.8 An efficiency consideration

Note that each item in an item set may have a different set of indentations. For example, we may have an item set containing both of the following two items:

\[
\begin{align*}
[A & \rightarrow \bullet a' b^*; I; u] \\
[A & \rightarrow \bullet a' b^*; I; u]
\end{align*}
\]

Even if they start with the same indentation set, I, after reading an a, the indentation sets for these items will be different from each other. For the first item, the indentation set will be restricted to indentations strictly greater than the indentation for a. For the second item, the indentations will be restricted to indentations equal to that of a. Thus, different items can have different indentation sets, and a naive factoring (e.g., sharing I between items in an item set) is insufficient. This does not preclude the possibility of a clever refactoring like the one done with the lookahead sets, but we have been unable to find such a factoring that works in all cases. Nevertheless, as a practical matter the following techniques seem to work well. Observe that we need to keep only the indentation sets for items before the closure is taken. Items generated by the closure operation can be annotated with the \(\text{\texttt{c}}\) that can lead to them and this value can be incorporated into the lookahead check. In addition, when we can determine that some set of items will always have the same indentation set, we can represent them using a common

![Figure 6. The structure of lookahead tokens in a parse tree.](image-url)
The parser is LALR instead of LR($k$), but the techniques shown in Section 5 generalize straightforwardly to LALR. Note that this implementation is only a prototype and is only intended for testing the practical feasibility of parsing with IS-CFGs. In particular, no significant effort was put into optimizing the performance of the generated parser.

The Haskell parser from the haskell-src package [Marlow et al. 2011] uses techniques for implementing layout similar to those used by GHC. However, it is packaged as a standalone parser, and this makes it easy to isolate for benchmarking purposes. This parser was modified to use an IS-CFG instead of using shared header files. Of the 178 remaining Haskell modules, 85 cannot be parsed by the unmodified haskell-src parser. There is a slight upward trend in the factor by which the IS-CFG based parser is slower than the unmodified parser. This is primarily due to the fact that we are graphing the ratio between the performance of the modified and unmodified parsers and the unmodified parser has low-order performance overheads that are more significant on small inputs. Given that this is a prototype implementation with little optimization, the fact that the IS-CFG version is only one to three times slower than the standard haskell-src parser is promising. This overhead is likely due to the manipulation of the indentation sets as the representation of indentation sets is naive. Since in practice only certain sorts of sets are common (e.g., singletons and the set $\mathbb{N}$), an improved version could optimize for these sorts of sets. In addition, we could take advantage of the tokens with an indentation of infinity by adding a fast path through the parser that short circuits the indentation computations.

7. Related Work

The uulib parser library [Swierstra 2011] and the indents [Anklesaria 2012] and indentparser [Kurz 2012] extensions to the Parsec [Leijen and Martini 2012] parser library provide support for indentation-sensitive parsing. To the best of our knowledge there is no published, formal theory for the sort of indentation that these parsers implement. They are all combinator-based, top-down parsers and use some variation of threading state through a parser monad to track the current indentation.

Hutton [1992] describes an approach to parsing indentation-sensitive languages that is based on filtering the token stream. This idea is further developed by Hutton and Meijer [1996]. In both cases, the layout combinator searches the token stream for appropriately indented tokens and passes only those tokens to the combinator for the expression to which the layout rule applies. As each use of layout scans the remaining tokens in the input, this can lead to quadratic running time. Given that the layout combinator filters tokens before parsing occurs, this technique also cannot support subexpressions, such as parenthesized expressions in Python, that are exempt from layout constraints. Thus, this approach is incapable of expressing many real-world languages including ISWIM, Habit, Haskell, and Python.

Erdweg et al. [2012] propose a method of parsing indentation-sensitive languages by effectively filtering the parse trees generated by a GLR parser. The GLR parser generates all possible parse trees irrespective of layout. Indentation constraints on each parse node then remove the trees that violate the layout rules. For performance reasons, this filtering is interleaved with the execution of the GLR parser when possible. Aside from the fact that they require a GLR parser and thus generate parse trees that might not be used, a critical difference between their system and the one presented in this paper is that their indentation constraints are in terms of the set of tokens under a non-terminal whereas the system in this paper uses constraints between non-terminals and their immediate children. Thus, the two approaches look at the problem from different perspectives. Erdweg et al. [2012] do not consider the question of an LR($k$) parser.

Brunauer and Mühlbacher [2006] take a unique approach to specifying the indentation-sensitive aspects of a language. They use a scannerless grammar that uses individual characters as tokens and has non-terminals that take an integer counter as parameter. This integer is threaded through the grammar and eventually specifies the number of spaces that must occur within certain productions. The grammar encodes the indentation rules of the language by carefully arranging how this parameter is threaded through the grammar and thus how many whitespace characters should occur at each point in the grammar.

![Figure 7. Benchmark results.](image-url)
While encoding indentation sensitivity this way is formally precise, it comes at a cost. The YAML specification [Ben-Kiki et al. 2009] uses the approach proposed by Brunauer and Mühlbacher [2006] and as a result has about a dozen and a half different non-terminals for various sorts of whitespace and comments. With this encoding, the grammar cannot use a separate tokenizer and must be scannerless, each possible occurrence of whitespace must be explicit in the grammar, and the grammar must carefully track which non-terminals produce or expect what sorts of whitespace. The authors of the YAML grammar establish naming conventions for non-terminals that help manage this, but the result is still a grammar that is difficult to comprehend and even more difficult to modify.  

While this approach bears some similarity to the technique proposed in this paper, a key difference is that their method uses the parameters of non-terminals to generate explicit whitespace characters and thus incurs a significant accounting overhead in the design of the grammar. On the other hand, the system presented in this paper operates at a higher level, using the parameter to indicate the column or indentation at which non-terminals and terminals should occur. This is subtle distinction, but it has a profound impact. As shown in Section 5, layout rules are comparatively simple to encode this way, and as shown in Section 5, this formalism is amenable to traditional parsing techniques such as LR(k) parsing.  

Note that none of the systems reviewed above present an LR(k) parsing algorithm. They use either top-down parsers or, in the case of Erdweg et al. [2012], a GLR parser.

8. Conclusion

This paper presents a grammatical formalism for indentation-sensitive languages. It is both expressive and easy to use. We derive provably correct GLR and LR(k) parsers for this formalism. Though not shown here, CYK, SLR, LALR, GLL, and LL(k) parsers can also be constructed by appropriately using the key technique of factoring item sets. Experiments on a Haskell parser using this formalism show that the parser runs between one and three times slower than a parser using traditional grammar engineering. Though not shown here, CYK, SLR, LALR, GLL and LL(k) parsing.


