Principled Parsing for Indentation-Sensitive Languages
Revisiting Landin’s Offside Rule

Michael D. Adams
Portland State University

Abstract
Many languages, such as Haskell, Python, and F#, use the indentation and layout of code as part of their syntax. Because context-free grammars are not able to express these layout rules, existing parsers use *ad hoc* techniques to handle them. These techniques tend to be low-level and operational in nature, and thus 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 for expressing these layout rules and derives CYK, GLR, and LR(k) algorithms for parsing these languages. These grammars are easy to write and can be parsed efficiently. Example 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 [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 with 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:

```
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 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, the 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 lexer and the parser that inserts explicit delimiters, but this pass uses information about whether the parsing that occurs later in the pipeline will succeed or fail on particular inputs. Due to this dependency, Haskell implementations do not structure their parsers this way. As a result, the structural differences between the implementation and

```
def factorial(x):
  result = 1
  for i in range(x):
    result = result * i
  return result
```

Here the indentation determines that the `for` loop ends before the `return` and the `factorial` function ends after the `return`. If the `return` were indented two more spaces, then it would be part of the `for` loop.

In both languages, traditional context-free grammars are not sufficient to uniquely parse code. The parser must track and account for the indentation of each expression.

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. Variations on this rule are used by Haskell, Miranda [Turner 1989], Orwell [Wadler 1985], Curry [Hanus (ed.) 2006], and Habit [HASP Project 2010]. F# is indentation sensitive when its lightweight syntax is enabled [Syme et al. 2010] §15.1. The block styles in the YAML [Ben-Kiki et al. 2009] data serialization language are indentation sensitive, as are many forms in the MarkDown [Gruber] and reStructured-Text [Goodger 2012] markup languages. Even Scheme has an indentation-sensitive syntax in the form of SRFI-49 [Möller 2005], though it is not often used.

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

...
the specification make it difficult to determine if the parser accurately reflects the specification.

This paper proposes a grammar formalism for expressing layout rules. This formalism elegantly expresses the indentation rules for a number of languages and is amenable to traditional parsing techniques such as LR(k) and LL(k) parsing. The expressivity of the technique is shown by defining grammars for ISWIM, Miranda, Haskell, and Python. To demonstrate the practical performance of the technique, we have modified the Happy parser generator [Mar- low and Gill 2009] to implement this formalism and benchmarked it against the performance of a pre-existing Haskell parser that uses traditional, ad hoc techniques.

The primary contributions of this paper are the formalism for indentation-sensitive context-free grammars and the technical insights and algorithmic tricks needed to construct efficient parsers for them. These grammars are thus both theoretically sound and practical to implement. Writing indentation-sensitive grammars using this formalism is easy and convenient, and parsers for these grammars are fast and efficient.

This paper is organized as follows. Section 2 informally defines a notion of indentation-sensitive grammars. Section 3 shows examples of using these grammars to encode the layout rules of several languages. Section 4 presents a formal definition of indentation-sensitive grammars. Section 5 derives CYK, GLR and LR(k) parsing algorithms for indentation-sensitive grammars. Section 6 reports on the performance of a practical implementation. Section 7 reviews related work. Section 8 concludes.

Section 5 assumes a fair amount of familiarity with standard parsing techniques and in particular the work by Knuth [1965] on LR(k) parsing, but otherwise this paper assumes only 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. Usually, this represents the minimum column at which the terminals in the expressions may appear. For example, with the code

\[ \text{factorial} \]

the function as a whole is at indentation 1, the function body is at indentation 3, and the body of the \text{for} loop is at indentation 5.\footnote{Since it simplifies the grammars of several languages, we adopt the convention that the first column is 1 instead of 0. We reserve indentation 0 for a pseudo-column that is further left than any terminal.} While, most layout rules increase the current indentation, some forms, such as parenthesized expressions in Python, decrease it. In these cases, tokens may end up further left than the indentation of a surrounding expression. Indentations are natural numbers, and we write \( X^1 \) to denote a non-terminal or terminal \( X \) that has an indentation of \( i \).

The productions of our grammars will restrict how the indentation of each subexpression can relate to the indentation of the expression as a whole. For example we may write \( A \rightarrow ("A") \) to mean that when using this production (and all) must be at the current indentation but \( A \) must be at a greater indentation. Combined with the production \( A \rightarrow ε \), this forms a grammar where nested parentheses must be indented. In general, we write a production as \( A \rightarrow X_1^{ι_1}X_2^{ι_2}...X_n^{ι_n} \) to mean that the indentation of \( A \) relates to the indentations of each \( X_1, X_2, ..., X_n \) according to the relations \( ι_1, ι_2, ..., ι_n \). That is to say, this production requires that \( ι_1 > 1, ι_2 > 2, ..., ι_n > n \) if \( i \) is the indentation of \( A \) and \( j_1, j_2, ..., j_n \) are the respective indentations of \( X_1, X_2, ..., X_n \).

While in principle any set of indentation relations can be used, we restrict ourselves to the relations \( =, >, ≥, \) and \( ⊛ \). The \( = \), \( > \), and \( ≥ \) relations have their usual meanings. The \( ⊛ \) relation is \( i \{i + 2, i \} [i \in N] \). It requires that the annotated symbol have an indentation of zero but does not restrict the indentation of the non-terminal on the left of the production.

We call these grammars indentation-sensitive context-free grammars (IS-CFG) to contrast them with traditional indentation-insensitive context-free grammars (II-CFG). We formally define IS-CFGs in Section 4.

Many languages have forms that require the first token of an expression to be at the same indentation as the expression. Thus for every non-terminal or terminal, \( X \), we introduce the non-terminal \( [X] \), which behaves identically to \( X \) except that its indentation is always equal to the indentation of its first token. Note that this is merely syntactic sugar, as we can introduce the production \( [a] \rightarrow a^ι \) for each terminal \( a \) and the production \( [A] \rightarrow [X_1]^ιX_2^ι...X_n^ι \) for each production \( A \rightarrow X_1^ιX_2^ι...X_n^ι \) if \( n ≥ 1 \). By replacing \( ⊛ \) with \( = \), the first symbol of each production is forced to have exactly the same indentation as the non-terminal on the left of the production. This is a straightforward, mechanical transformation that does not have to be written by the user.

3. Indentation-Sensitive Languages

Despite this system’s simplicity, it can express a wide array of layout rules. To demonstrate this, this section shows examples of how to encode the layout rules of several languages. Where possible, we use the non-terminal names from the original grammar of each language. Though not shown here, sketches for other indentation-sensitive languages have been constructed including for Curry, Occam\footnote{Occam requires adding the indentation relation \( \{i + 2, i \} [i \in N] \) to \( = \) because it has forms that require increasing the indentation by exactly 2.} Orwell, Habit, and SRFI-49.

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”\footnote{Landin spells it as both “offside” and “off-side”. We adopt the spelling consistent with the rule from sports.} that specifies that:

The southeast quadrant that just contains the phrase’s first symbol must contain the entire phrase, except possibly for [parenthesized] subsegments. [Landin 1966]

The IS-CFG for ISWIM is similar to the II-CFG for ISWIM that ignores the offside rule, but we annotate each production of the IS-CFG with appropriate indentation relations. For productions that do not involve the offside rule, 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 then or equal to the current indentation.

Next, we consider the forms that involve the offside rule\footnote{Landin is not clear about whether his offside rule applies to all syntactic forms. His examples imply that it does not apply to all expressions.}. Wherever a non-terminal \( A \) should trigger the offside rule, we use \( |A|^ι \) instead of \( A^ι \). The expression parsed by \( |A|^ι \) is at a new indentation that is greater than or equal to the current indentation but simultaneously equal to the first token of that expression. All other tokens in that expression must be at an indentation equal to or greater than this new indentation.
Using these rules, we then have the following automatically generated productions:

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

An example where the offside rule might be used is the \text{where}\[7\] expression:

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

Note the use of \text{expr} \geq \text{expr} instead of \text{expr} \geq \text{expr}. This encodes the offside rule in this production.

Finally, since parenthesized expressions are exempt from the indentation constraints of the surrounding code, the production for parenthesized expressions is:

\[
\text{expr} \rightarrow ( \geq \text{expr} \geq \, \text{expr} \geq )
\]

This sets the indentation of the expression inside the indentation to zero and thus frees it from any indentation constraints coming from the context of the expression. Uses of the offside rule inside the parentheses may still have effect, of course.

As an example, consider using these productions along with the production \text{expr} \rightarrow \text{ID} \geq \text{to parse the expression:}

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

The second difference is that the language specification adopts the notational convention that for any non-terminal \text{x},

\[
\text{x}(; ) \text{ means that x is followed by an optional semicolon and is subject to the offside rule..., so that every token of x must lie below or to the right of the first. Provided the layout makes it clear where x terminates, the trailing semicolon may be omitted. [Turner [1989] §25]}
\]

This notation is easily handled by introducing, for each non-terminal \text{A}, the non-terminal \text{A} (:) and the two productions \text{A} :) \rightarrow |A| \geq and \text{A} :) \rightarrow |A| \geq \text{; ; ; .}

### 3.2 Haskell

#### 3.2.1 Language

Haskell uses a more sophisticated offside rule than ISWIM. Blocks of code that use the layout rule consist of multiple clauses. Each line of code that is at the same indentation as the block starts a new clause within the block. Lines of code that are further indented than the block continue previous clauses within the block.

Semicolons ( ; ) can be used to explicitly separate clauses, and curly braces ( { } ) can be used to explicitly delimit blocks. Code within explicit blocks is exempt from the indentation restrictions due to surrounding blocks.

This indentation rule is given in the Haskell language specification in terms of both the lexer and an extra pass between the lexer and the parser. First, the lexer inserts special \text{<} \text{n} \text{>} and \text{<} \text{n} \text{>} tokens. Roughly speaking, \text{<} \text{n} \text{>} is inserted where a new block might start and \text{<} \text{n} \text{>} is inserted where a new clause within a block might start. They are inserted according to the following rules:

- If a \text{let}, \text{where}, \text{do}, or \text{of} keyword is not followed by the lexeme \text{;}, the token \text{<} \text{n} \text{>} is inserted after the keyword, where \text{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 \text{let} or \text{module}, then it is preceded by \text{<} \text{n} \text{>} where \text{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 \text{<} \text{n} \text{>, where n is the indentation of the lexeme, provided that it is not, as a consequence of the first two rules, preceded by \text{<} \text{n} \text{>. [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 \text{L} \text{tokens} \text{ where tokens is the stream of tokens from the lexer and L is the function in Figure 2} Thus the context-free grammar only has to deal with semicolons and curly braces. It does not deal with layout.

This \text{L} function is fairly intricate, but the key clauses are the ones dealing with \text{<} \text{n} \text{>} and \text{<} \text{n} \text{>. After a \text{let}, \text{where}, \text{do}, or \text{of} keyword, the lexer inserts a \text{<} \text{n} \text{>} token. Suppose the first clause for \text{<} \text{n} \text{>} handles this token. A open brace (\{ ) will be inserted, and the indentation \text{n} will be pushed on the stack of indentations that is the second argument to \text{L}. If a line starts at the same indentation, then the first clause for \text{<} \text{n} \text{>} executes and a semicolon ( ; ) is inserted to start a new clause. If it starts at a smaller indentation, then the second clause for \text{<} \text{n} \text{>} 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 indent, 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 \text{; ; , and } \} \text{ tokens are inserted wherever layout indicates that blocks start, new clauses begin, or blocks end, respectively. The other clauses in \text{L} handle a variety of other scenarios and edge cases.
L (<n>:ts) (m:ms) = {n} : (L (t:ts) m:ms) if m = n
L (<n>:ts) ′ = {n} : (L (l:tl) m:ms) if m < n
L (t:ts) ms = L ts ms
L (t:ts) (m:ms) = ′t′ : (L ts (m:ms)) if n > m
L (t:ts) [] = ′t′ : (L ts [n]) if n > 0
L (t:ts) ms = ′t′ : ′t′ : (L (t:ts) ms)
L (t:ts) (0:ms) = ′t′ : (L (t:ts) ms)
L (′t′:ts) ms = ′t′ : (L (ts:ms) ms)
L (′t′:ts) (0:ms) = ′t′ : (L ts ms)
L (′t′:ts) (m:ms) = ′t′ : (L ts (m:ms)) if m ≠ 0 and parse-error(t)
L (′t′:ts) [] = ′t′ : (L (′t′:ts) [])
L (′t′:ts) (m:ms) = ′t′ : (L ts (m:ms)) if m ≠ 0

Figure 2. Haskell’s L function [Marlow (ed.) 2010] §10.3

Note that L uses parse-error to signal that the result is 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 started after the do needs to be terminated before the in. This requires knowledge about the parse structure in order to be handled property, and thus parse-error(t) is used to query the parser.

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 the L function. Instead, the lexer and parser share state consisting of a stack of indentations that is similar to the second parameter to L. 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].

While preparing the parser in Section 3.2.1 we found that even minor changes to the error propagation of the parser affected whether syntactically correct programs were accepted.

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 troublesome.

3.2.2 Grammar

Haskell’s layout rule is more complicated than for 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 parse-error(t) is simply implicit in the structure of the grammar.

Figure 3 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 >. We use > instead of ≥ because Haskell distinguishes between tokens that are at an indentation equal to the current indentation and tokens that are at a strictly greater indentation. The former start a new clause while the latter continue of the current clause.

In Haskell, a block can be either delimited by explicit curly braces or it can use the layout rule. In Figure 3, this is reflected by the two different productions for altBlock. If it expands to ′{alt<br>alt’s close}′, then the ⊛ relation sets the current indentation to zero so that code in alt’s does not have to respect the indentation constraints from the surrounding code. On the other hand, if it expands to altLayout, then the > relation increases the indentation. In the production for altLayout, the use of [alt’s] ensures that the first tokens of the alt’s align to the same column. Note that when using curly braces to delimit a block, each alt must be separated by a semicolon (;), but when using the layout rule, each alt can be separated either using a semicolon or by indentation.

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 decl in place of alt) and structures (e.g., a do block is a sequence of stmt ending in an exp). Note that a subtlety of Haskell’s layout rule is that closing braces may occur at any column. We use the combination of the ⊛ on close and the > on ′?′ to support this. In addition, tokens on the same line as, but after, a closing brace may not have to respect the current indentation, as the L function only considers the indentation of the first token of a line (i.e., where ′<r>′ is inserted) and of tokens after a let, where, do or of keyword (i.e., where ′{n}′ 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. 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 the linear order of tokens controls what tokens are indentation sensitive. For example, the token after the do keyword is indentation sensitive regardless of the structure of the expression following the do.

Finally, GHC also supports an alternative indentation rule that is enabled by the RelaxedLayout extension and that allows opening braces to be at any column [GHC 2011 §1.5.2]. This is easily implemented by tweaking the first production for altBlock to be:

altBlock → altCurly
altCurly → '{' altCurly '}'

While we do not formally prove that this IS-CFG is equivalent to the use of L, the tests in Section 3 support this claim. The main theoretical challenge of such a proof is connecting the recursion in L, which inducts over the token stream, with the recursion in the IS-CFG, which inducts over the phrase structure. It may be possible to prove this equivalence by showing the equivalence of the LR(k)
parsers for the two different approaches, but we have not explored that approach. Since the major Haskell implementations do not use \texttt{L} in their parsers, they face the same problem, but also lack the formal foundations that IS-CFGs provide.

### 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’s line joining rules complicate matters.

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

Python’s explicit line joining rule is simple enough to implement directly in the lexer, but Python also has an 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.

```python
... The indentation of the continuation lines is not important.
```

This rule 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.

```python
x = [
    y |
    z = 3
```

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 convert to an IS-CFG, we will see that they share many similarities. The lexer still needs to produce \texttt{NEWLINE} tokens, but it does not need to produce \texttt{INDENT} or \texttt{DEDENT} tokens. As with Haskell, we start with a grammar where the non-terminals are annotated with an indentation relation of \(=\) and the terminals with an indentation relation of \(>\). In Python, the only form that changes the indentation is the \texttt{suite} non-terminal, which represents a block of statements contained inside a compound statement. It always follows a header ending in a colon (:). For example one of the productions for \texttt{while} is:

```plaintext
while_stmt \rightarrow \texttt{\'while\'} test\texttt{\'\:'}\texttt{\'suite\'
```

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 handles when the suite is a multi-line statement. The productions for these cases are:

```plaintext
suite \rightarrow \texttt{stmt_list} \texttt{\'NEWLINE\'}
```

When a \texttt{suite} is of the multi-line form, the initial \texttt{NEWLINE} token ensures that the \texttt{suite} is on a separate line from the preceding header. The block inside a \texttt{suite} must then be at some indentation greater than the current indent. Such a block is a sequence of \texttt{statement} forms that all start with their first token in the same column. In Python’s grammar, the productions for \texttt{statement} already include a terminating \texttt{NEWLINE} so they are not needed here.

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 \(>\) indentation relation. Since the final delimiter can also appear at any column, we ensure that it also can appear at any column. For example, one of the productions for list construction becomes:

```plaintext
atom \rightarrow \texttt{\'[\'] listmaker\texttt{\'\'] close_bracket\texttt{\'}
```

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 with infinite indentation as we do for Haskell.

Second, while a lexer based on regular expressions can detect the start of a line and thus produce finite indentations for the first token on a line but infinite indentations for other tokens, it cannot detect matching parentheses to determine that \texttt{NEWLINE} tokens should be omitted inside delimited forms. Non-terminals that occur inside them need to allow the insertion of \texttt{NEWLINE} tokens at arbitrary locations. In some cases, 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 \([A]\). Alternatively, it may be possible to use a grammar that does not use \texttt{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 triggered by a line ending in a backslash (\). It gives the tokens of an explicitly joined line the same indentation as the backslash itself. The backslash is not emitted as a token.

### 3.4 Conventions and syntactic sugar

In an IS-CFG every symbol in every production needs to be annotated with an indentation relation, but in most languages, the majority of grammatical forms allow terminals to appear at any indentation greater than a certain minimum 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 \(=\) if it is a non-terminal and \(>\) if it is a terminal. By using this convention most productions in a grammar do not have to be annotated with indentation relations and thus they look like ordinary II-CFG productions.
the forms that explicitly deal with indentation must be explicitly annotated.

In addition, just as II-CFGs often allow the use of alternation bars (|) or Kleene stars (*) to simplify the writing of such 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. For example, this would allow defining altBlock as:

\[ \text{altBlock} \to \{ \text{altBlock}^* \}^* \]

This is merely a notational convenience and does 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 \( A \to X_1X_2 \cdots X_n \) for a tuple \((A, X_1, X_2, \ldots, X_n)\) 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 \( \Rightarrow \) by \( \Rightarrow \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^* \) such that \( UAV \Rightarrow UX_1X_2 \cdots X_nV \) if \( A \to X_1X_2 \cdots X_n \) is an element of \( \delta \).

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

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

As a notational convention, let \( i, j, l \) be indensions and \( \Rightarrow \) be an indentation relation. For the sake of compactness, we adopt the notations \( X^i \) and \( X^i \), respectively, for a pair of \( X \) and either an indentation \( i \) or an indentation relation \( \Rightarrow \). Thus we write \( A \to X_1^i_1X_2^i_2 \cdots X_n^i_n \) for a tuple \((A, (X_1, i_1), (X_2, i_2), \ldots, (X_n, i_n))\) that is an element of \( \delta \).

As with II-CFGs, we define a rewrite relation

\[ \Rightarrow \subseteq ((N \times \Sigma) \times N)^* \times ((N \times \Sigma) \times N)^* \]

where \( UAV \Rightarrow UX_1^i_1X_2^i_2 \cdots X_n^i_nW \) if \( A \to X_1^i_1X_2^i_2 \cdots X_n^i_n \) and \( i_1, i_2, i_3, \ldots, i_n \). The \( \Rightarrow^* \) relation, the language \( \mathcal{L}(G) \), derivations, and parse trees are all defined as with II-CFGs except that they are now in terms of this new rewrite relation.

Note that every II-CFG is encodable as an IS-CFG by translating every production \( A \to X_1X_2 \cdots X_n \) to \( A \to X_1^iX_2^i \cdots X_n^i \) and every word \( a_1a_2 \cdots a_m \to a_1^i a_2^i \cdots a_m^i \). Erasing the indensions and indentation relations in an IS-CFG results in an II-CFG.

5. Parsing

Of course, a grammar is not practically useful if we cannot effectively parse it. In this section, we show how to modify traditional parsing techniques for II-CFGs to handle IS-CFGs. We show this for CYK parsing, GLR parsing and LR(k) parsing. These techniques also work for SLR, LALR, and LL(k), but we do not present those here as they are straightforward once the techniques for LR(k) parsing are understood.

In the following we restrict ourselves to finite indentations, but extending these constructions to allow indentations of infinity is straightforward.

5.1 Basic properties

There are a few technical definition and properties that we will use in the development of these algorithms. We present these without discussion.

**Definition 1.** A non-terminal \( A \) is nullable if \( A^* \Rightarrow \varepsilon \) for all \( i \).

**Lemma 1 (Composition of indentation relations).** Every finite sequence of compositions of elements from \( I \) is one of \( =, >, \geq \), \( >^n \), or \( >^n\oplus \) for some \( n \geq 1 \).

**Proof.** \( = \) is a left and right identity under composition. \( \oplus \) is a left annihilator under composition. The compositions of \( > \) with \( \geq \) and \( \geq > \) are both \( \geq \).

**Lemma 2 (Closure of indentation relations).** The closure of \( I \) under finite sequences of composition and either finite or infinite sets of unions is the set of unions of one or more of \( =, >, \geq, >^n \), or \( >^n\oplus \) for some \( n \geq 1 \) where each of these is in the union only once. We call this closure \( \bar{I} \) and as a notational convention let \( \bar{\delta} \) be an element of \( \bar{I} \).

**Proof.** By Lemma 2, every finite composition is of the form \( =, >, \geq, >^n \), or \( >^n\oplus \). Since \( >^n\geq >^m \) if \( n \geq m \) and \( >^n\oplus >^m \) if \( m \geq n \), at most one occurrence of \( >^n \) and one occurrence of \( >^n\oplus \) needs to be in the union.

**Lemma 3 (Inductions of unique parse).** If \( A^* \Rightarrow^* W \), then the set of all indents \( i \) such that \( A^i \Rightarrow^* W \) using the same sequence of productions is either \( N \), a singleton or the upper bounded set \( \{ i | i \leq n \} \) for some \( n \in N \). Furthermore, if it is a singleton or upper bounded set, then the maximum indentation is limited to be at most the maximum indentation in \( W \).

**Proof.** Consider the derivation as a parse tree. Each edge can be annotated with the indentation relation between the parent and child nodes. The relation between the indentation of the root \( A^i \) and each leaf \( a_1^i \) is then a composition of the edges in the path from \( A^i \) to \( a_1^i \). The possible values of \( i \) are the values compatible with every leaf indentation and the root’s relation to them. By Lemma 2, every such relation is one of \( =, \geq, \geq, >^n \) or \( \geq \) and for a given leaf the compatible root indentations are thus are either \( N \), a singleton, or \( \{ i | i \leq m \} \) for some \( m \in N \). The intersection of these over all leaves is thus either \( N \), a singleton, or \( \{ i | i \leq n \} \) for some \( n \in N \).

**Lemma 4 (Inductions of ambiguous parse).** If \( A^i \Rightarrow^* W \), then the set of all indents \( i \) such that \( A^i \Rightarrow^* W \) using the same sequence of productions is either \( N \) or a finite subset of \( N \). Furthermore, the finite subsets of \( N \) are bounded by the maximum indentation in \( W \).

**Proof.** By Lemma 2 and the union over all parses.

5.2 General approach

In practice, translating traditional techniques to IS-CFGs is a difficult task if we attempt to translate them directly. However, we can use an indirect approach that makes the task much easier. First, we model the IS-CFG by an infinite II-CFG. Then, we consider the traditional parsing algorithms in terms of this II-CFG. Finally, we factor out the parts of the construction representing indentations so these algorithms can be expressed finitely. As we will see, though this last step sounds simple, it still sometimes requires an intuitive leap. Stated in the abstract, this technique is rather vague, so the
first algorithm we will consider is the CYK algorithm as it is comparably easy to understand.

5.3 CYK parsing

Since the CYK algorithm [Cocke and Schwartz 1970; Younger 1967; Kasami 1965] operates only on grammars in a normalized form, for the purposes of this construction we assume that every production in our IS-CFG has two or fewer symbols on its right-hand side. If it does not, it is trivial to translate the IS-CFG into an equivalent IS-CFG that is normalized by translating every production \( A \rightarrow X_1^n X_2^n \cdots X_m^n \) to the following productions, where \( A_1, A_2, \ldots, A_{m-2} \) are fresh non-terminals.

\[
A \rightarrow X_1^n A_1^n \\
A_1 \rightarrow X_2^n A_2^n \\
\vdots \\
A_{m-2} \rightarrow X_m^n A_m^n.
\]

Once we have this normalized grammar, 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 will derive a computable parsing algorithm. Given an IS-CFG \( G = (N, \Sigma, \delta, S) \), this II-CFG is \( G' = (N', \Sigma', \delta', S') \) where:

\[
N' = N \times \mathbb{N} \\
\Sigma' = \Sigma \times \mathbb{N} \\
\delta' = \{ (A \rightarrow X_1^n X_2^n \cdots X_m^n) \mid A \rightarrow X_1^n X_2^n \cdots X_m^n \in \delta, \ i, j_1, j_2, \ldots, j_m \in \mathbb{N}, j_i \nleq j_1 i, j_2 \nleq j_1 i, \ldots, j_m \nleq j_1 i \} \\
S' = S
\]

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 \( G' \).

If we use the standard CYK parsing algorithm on \( G' \), the dynamic-programming table that stores the intermediate parsing results needs to store subsets of \( N' \times \Sigma' \), that is to say subsets of \( (N \times \mathbb{N}) \cup (\Sigma \times \mathbb{N}) \). This is equivalent to storing mappings from \( N \times \mathbb{N} \) to subsets of \( N \times \mathbb{N} \). Due to Lemma 6 the subsets of \( N \times \mathbb{N} \) that can appear in the ranges of these mappings are either \( N \times \mathbb{N} \) or a finite subset of \( N \times \mathbb{N} \). Thus they are finitely representable, and we can translate the CYK parsing algorithm on \( G' \) to the following algorithm, which operates directly on \( G' \).

**Algorithm 7.** Given input \( w \) of length \( n \), if \( n = 0 \) and \( S \) is nullable, then accept. Otherwise:

- Construct a parsing table \( P \) indexed by \( \{0, 1, \ldots, n-1\} \times \{1, \ldots, n\} \) and containing mappings from \( N \cup \Sigma \) to \( \mathbb{N} \). Initially, these mappings map all values to the empty set.

- For each terminal \( a_i \) at position \( p \) in \( w \), add \( a_i \rightarrow \{i\} \) to the mapping at position \( (p, p+1) \) of the table.

- Next, for each \( k \in \{1, 2, \ldots, n\} \) in ascending order:
  - For each production \( A \rightarrow X_1^n X_2^n \), add \( A \rightarrow \{i\} j_1 \in J_1, j_2 \in J_2, j_1 \nleq j_2, j_1 \nleq j_2 \) to the mapping at position \( (p, p+k) \) of the table if \( X_1 \mapsto J_1 \) and \( X_2 \mapsto J_2 \) at positions \( (p, q) \) and \( (q, p+k) \) respectively for some \( p < q < p+k \).
  - For each production \( A \rightarrow X_1^n A \rightarrow X_2^n Y^n \) or \( A \rightarrow Y^n X^n \), where \( Y \) is nullable add \( A \rightarrow I \) to the mapping at position \( (p, p+k) \) of the table if \( X \mapsto I \) at position \( (p, p+k) \) of the table. Repeat this process until it reaches a fixed point.

- Accept if \( S = A^{\delta_0} \rightarrow I \) is in position \( (0, n) \) of the table and \( i_0 \in I \). Otherwise, reject.

Note that when processing the production \( A \rightarrow X_1^n X_2^n \) or certain values of \( J_1, J_2, J_3 \) and \( J_2 \) produce a set of indentations that is the entire set \( \mathbb{N} \). These must be detected and the finite representation of \( \mathbb{N} \) directly produced instead of looping over every possible indentation.

Algorithm 7 runs in \( O(mn^3) \) time for a string of length \( n \) with a maximum indentation of any terminal of \( m \). This is reasonably efficient given that the traditional, indentation-insensitive CYK algorithm takes \( O(n^3) \). This example shows the general approach to extending traditional parsing techniques to handle IS-CFGs. The remainder of this section shows how to apply the same technique to GLR and LR(\( k \)) parsing.

5.4 LR(\( k \)) parsing

In order to derive GLR and LR(\( k \)) parsers for IS-CFG grammars, we use the same technique as with CYK parsing and consider the standard parsing algorithms applied to \( G' \), the infinite II-CFG that models the IS-CFG \( G \). We then determine appropriate finite representations that can be used in a practical parser. In this development, we closely follow the original presentation of LR(\( k \)) parsing by [Knuth 1965] with only minor changes to use modern notation.

Recall that in a right-most derivation, the right-most non-terminal is always expanded before any of the other non-terminals. The symbols resulting from this expansion are called the handle. For example, if

\[ S \Rightarrow^{*} UXJ/V \Rightarrow UX_1^{j_1} \cdots Y_1^{j_1} Y_2^{j_2} V \]

is a right-most derivation, then a handle of \( UY_1^{j_1} \cdots Y_2^{j_2} V \) is \( UX_1^{j_1} \cdots Y_1^{j_1} Y_2^{j_2} V \). (Note that in a rightmost derivation \( V \) will necessarily contain only elements of \( \Sigma \), though \( U \) and \( Y_1^{j_1} \cdots Y_2^{j_2} V \) may also contain elements of \( N \).

Since an LR(\( k \)) parser works from the result of a right-most 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 \( G' = (N', \Sigma', \delta', S') \) we construct the right-linear (and thus regular) grammar \( G'' = (N'', \Sigma' \cup \delta', \delta'') \) for recognizing prefixes that end in a handle. Here the non-terminals are

\[ N'' = \{ A_i \mid a_1^{i_1} a_2^{i_2} \cdots a_k^{i_k} \} \in N', a_1^{i_1}, a_2^{i_2}, \cdots, a_k^{i_k} \in \Sigma \} \]

and represent the part of the string that contains the handle. The \( a_1^{i_1} a_2^{i_2} \cdots a_k^{i_k} \) track the \( k \) terminals expected after the handle and are the lookahead. For each \( A_i \mapsto X_1^{j_1} \cdots X_{m+1}^{j_{m+1}} \cdots X_n^{j_n} \) in \( \delta' \) and each \( u = a_1^{i_1} a_2^{i_2} \cdots a_k^{i_k} \), we include the following in \( \delta'' \):

\[ A_i \mapsto \{i\} j_1 \in J_1, j_2 \in J_2, j_1 \nleq j_2, j_1 \nleq j_2 \]
where $A$ is-CFG may produce multiple productions that can match a given $S$ again. This is merely following the development by Knuth [1965] and proceed from there. Thus we have the following construction.

We rewind to the state just before the first symbol of the handle. Start as done in Algorithm 8 is inefficient, of course. Instead, we can rewrite the entire string and restarting the automaton from the start of the algorithm with the new configuration.

Algorithm 8. Given input string $W$, if $S = W$ then stop and accept the string. Otherwise, find all prefixes of $W$ that match $\mathcal{L}(G')$ and non-deterministically choose one of the following:

- If there are no such matches, then reject the string.
- Given a match where the last production of the match is 
    \[ A_i \rightarrow X^1_j \cdots X^j_n u \]
    the word $X^1_j \cdots X^j_n$ is a handle for $W$. In $W$, replace this occurrence of $X^1_j \cdots X^j_n$ with $A_i$, and repeat this 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 such as $A \rightarrow B^a$ and $A \rightarrow C^b$ in the IS-CFG may produce multiple productions that can match a given word (e.g., $A \rightarrow B^0$ and $A \rightarrow C^j$ where $j > i$). Once we convert to a finite version of the parsing algorithm, we can eliminate this non-determinism.

5.5 Parsing with stacks

Rewriting the entire string and restarting the automaton from the start as done in Algorithm 8 is inefficient, of course. 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. Thus we have the following construction. Again this is merely following the development by Knuth [1965] albeit on an infinite II-CFG.

We begin with the notion of an item. We denote an item by 
\[ [A_i \rightarrow X^1_j \cdots X^j_n \ast X^j_{m+1} \cdots X^j_n u] \]
where $A_i \rightarrow X^1_j \cdots X^j_n$ is a production in $G'$ and $u \in \Sigma^k$ is the lookahead. The algorithm maintains a stack of sets of items $S_0S_1 \cdots S_n$ where $S_0$ is the top element of the stack. We use the notation $S_0S_1 \cdots S_n | a_1a_2 \cdots a_kw$ to denote that $S_0S_1 \cdots S_n$ is the current stack and $a_1a_2 \cdots a_kw$ is the input word remaining to be consumed.

To parse a word $w$, we start with the configuration 
\[ S_0 \mid w \uparrow 0 \uparrow 1 \uparrow 2 \cdots \uparrow k \]
where $S_0 = \{ \hat{S} \rightarrow \cdot \hat{S}' \mid \hat{S}' \uparrow 0 \uparrow 1 \uparrow 2 \cdots \uparrow k \}$. $\hat{S}$ is a fresh non-terminal and $\uparrow 0, \uparrow 1, \uparrow 2, \cdots, \uparrow k$ are fresh terminals. We then run the following algorithm.

Algorithm 9. Given configuration $S_0S_1 \cdots S_n | a^1_1a^2_2 \cdots a^k_kw$, if $\hat{S} \rightarrow \cdot S'$, $u \in S_0$ and $a^1_1a^2_2 \cdots a^k_kw = \uparrow 0 \uparrow 1 \cdots \uparrow k$, then accept. Otherwise:

1. Compute the closure, $S'$, of $S_n$ where $S'$ is the least set of items satisfying the recurrence
\[
S' = S_n \cup \bigg\{ \begin{array}{c}
X^{m+1}_{m+1} \rightarrow Y^j_{m+1} \cdots Y^n_{m+1} ; v \\
A_i \rightarrow X^1_j \cdots X^j_n \ast X^j_{m+1} \cdots X^j_n u \\
v \in H_k \bigg( X^{m+1}_{m+1} \cdots X^j_n u \bigg) 
\end{array} \bigg\}.
\]

2. Compute the acceptable lookahead set $K$ where
\[
K = \bigg\{ v \bigg[ A_i \rightarrow X^1_j \cdots X^j_n \ast X^j_{m+1} \cdots X^j_n u \bigg] \in S', \bigg\} \bigg\} v \in H_k \bigg( X^{m+1}_{m+1} \cdots X^j_n u \bigg) \bigg\}
\]

3. For each production $A_i \rightarrow X^1_j \cdots X^j_n$ in $\delta'$, compute the acceptable lookahead set $K \bigg( A_i \rightarrow X^1_j \cdots X^j_n \bigg)$ where
\[
K \bigg( A_i \rightarrow X^1_j \cdots X^j_n \bigg) = \bigg\{ u \bigg[ A_i \rightarrow X^1_j \cdots X^j_n \ast u \bigg] \in S' \bigg\}
\]

4. Let $GOTO \ (S, Z') =$$
\bigg\{ \begin{array}{c}
A_i \rightarrow X^1_j \cdots X^j_n \ast X^j_{m+1} \cdots X^j_n u \\
A_i \rightarrow X^1_j \cdots X^j_n \ast X^j_{m+1} \cdots X^j_n u \bigg] \in S, \\
X^{j+1}_{j+1} = Z'
\end{array} \bigg\}
\]

and non-deterministically choose one of the following.

- If $a_1a_2 \cdots a_k \in K$, then do a shift action by returning to the start of the algorithm with the new configuration $S_0S_1 \cdots S_nGOTO \ (S_n, a^1_1)$.
- If $a_1a_2 \cdots a_k \in K \bigg( A_i \rightarrow X^1_j \cdots X^j_n \bigg)$ for some production $A_i \rightarrow X^1_j \cdots X^j_n$, then do a reduce action by returning to the start of the algorithm with the new configuration $S_0S_1 \cdots S_nGOTO \ (S_n, a^1_1a_2 \cdots a_kw)$.

5.6 Finite representations of the stacks and GLR parsing

Algorithm 9 contains both non-determinism and infinite sets. Here we depart from Knuth [1965] 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 the standard LR(k) 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 the original algorithm 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 $\bigg[ A_i \rightarrow X^1_j \cdots X^j_n \ast X^j_{m+1} \cdots X^j_n u \bigg]$ where $A_i \in N \times N, X^j_{j_1} \cdots, X^j_{j_m} \in \{ N \cup \Sigma \} \times N$, and $u \in (\Sigma \times N)^k$. Observe that the indentations to the left of the bullet, $j_1, \cdots, j_m$, do not effect the parsing process and multiple items that differ only in the the values of $j_1, \cdots, j_m$ can thus be represented by a single item. This reduces the state space slightly, but
we can go further and also factor out $j_{m+1}, \ldots, j_n$ by observing that the algorithm preserves the following completeness property.

**Definition 10** (Item set completeness). We say that a set of items, $S$, is complete if for every $A \to X_1^j \cdots X_m^j \bullet X_{m+1}^i \cdots X_n^i \in \delta$, 

$$[A^i \to X_1^j \cdots X_m^j \bullet X_{m+1}^i \cdots X_n^i ; u] \in S$$

implies that $S \supseteq \{[A^i \to X_1^j \cdots X_m^j \bullet X_{m+1}^j \cdots X_n^j ; u] \in S | j_{m+1}, \ldots, j_n \in \mathbb{N}, j_{m+1} \geq m+1, i, \ldots, j_n \geq n, i\}$

**Lemma 11.** If every item set on the stack is complete at the start of a loop through Algorithm 9 then every item set on the new stack at the start of the next loop is also complete.

**Proof.** For every production $X_{m+1} \to Y_{1}^{b_1} \cdots Y_{p}^{b_p}$, if the closure step adds the item $[X_{m+1}^{j_{m+1}} \to Y_{1}^{b_1} \cdots Y_{p}^{b_p} ; u]$, then by construction it adds all items of the form $[X_{m+1}^{j_{m+1}} \to Y_{1}^{l_1} \cdots Y_{p}^{l_p} ; u]$ where $l_1 \geq 1, j_{m+1}, \ldots, l_p \geq 1, j_{m+1}$. Thus this step of the algorithm preserves completeness.

The GOTO operation filters the set of items by requiring that $X_{m+1}^{j_{m+1}} = X_1^l$, but afterwards $X_{m+1}^{j_{m+1}}$ moved to the left of the bullet and therefore is no longer relevant to completeness. For example, the indentation of the start terminal. Despite this, the lookahead set is finitely representable. With this representation, the lookahead set is finitely representable.

With the original representation of item sets, we did not apply this criterion because there might be multiple reductions that differ only in their indentations. For example, if we have $A \rightarrow B^o$ in $G$, then both $A^3 \rightarrow B^o$ and $A^3 \rightarrow B^o$ are in $G'$.
computed dynamically. For a word of length $n$ with maximum indentation $m$, this algorithm runs in $O(nm)$ time.

But we must consider this representation carefully. At first glance, the representation looks like it could lead to lookahead values matching when they should not. Indeed, the following grammar shows how this can arise:

$$\begin{align*}
\text{expr} & \rightarrow \text{ID} > \\
\text{expr} & \rightarrow \text{do}\ ^{-1} \text{do}\ ^{-4} \text{ID}\ ^7 \text{ID}^2 \\
\text{stats} & \rightarrow \text{expr} \\
\text{stats} & \rightarrow \text{expr}^m | \text{stats}^m
\end{align*}$$

Consider the parse of the word $\text{do}\ \text{do} \ x \ y$

The valid lookaheads when about to parse the ID for $y$ are $\text{ID}^1$, $\text{ID}^2$, $\text{do}\ ^{-1}$, and $\text{do}\ ^{-4}$. The string should thus be rejected since $y$ is at an indentation of 2. But the LR(1) lookahead set using the new representation is $\{(\geq , = , \text{ID}) > \}, \{(\geq , = , \text{do}\ ^{-1}) > \}$. Since the current indentation is 7 and there exist $l$ such that $7 \geq 1, 2 = l,$ and $l > 0$, the string will not be immediately rejected.

Thus, this representation appears to overapproximate the set of indentations for a particular lookahead. This means that in Step 4 there could be more reductions possible than there should be. However, because we are restricting ourselves to LR(k) grammars, 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.

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. 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 new representation for lookaheads is designed so that every lookahead word that it represents comes from a valid parse. Thus if this representation generates extra conflicts, then there is some string that would have generated those same conflicts with the old representation. In that case, the original 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 not only is this new representation valid for any grammar that is LR(k), but using this representation we can detect when a grammar is not LR(k).

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 the following items:

$$\begin{align*}
[A & \rightarrow \bullet a^7 b^7; I; u] \\
[A & \rightarrow \bullet a^7 b^7; I; u]
\end{align*}$$

Even if we start with the same indentation sets, after reading an $a$ the two indentation sets will be different. For the first item, the indentation set will be restricted to indentations greater than the indentation for $a$. For the second item, the indentations will be restricted to indentations equal to that for $a$. Thus, different items can have different items sets, and a naive factoring, say 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 only need to keep the indentation sets for items before the closure is taken. Items generated by the closure operation can be annotated with the occurrence 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 indentation set. These techniques reduce the number of indentation sets that must be passed from one state to another, and based on the experience implementing the Haskell parser in Section 5 the number of such sets is usually one and almost always small.

6. Implementation

In order to verify the real-world practicality of this parsing technique, we modified the Happy parser generator [Marlow and Gill 2009] to support the parsing techniques presented in this paper. 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. 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 that makes it easy to isolate for benchmarking purposes. This parser was modified to use an IS-CFG instead of using shared state between the lexer and parser. Both the modified and unmodified versions were then run on the preprocessed source files from the base package [bas 2012]. Two modules (GHC.Float and GHC.Constants) could not be preprocessed due to missing header files. Of the 178 remaining Haskell modules, 85 cannot be parsed by the unmodified haskell-src parser due to syntactic extensions, such as rank-2 types, that are not supported by haskell-src. This left 93 source files that are parsable by the unmodified parser. All of these files parsed and produced the same parse tree when parsed using the IS-CFG based parser.
A performance problem as each use of layout scans the remaining binator for the expression to which the layout rule applies. This has been cases, the layout combinator searches the token stream for appror-
sensitive languages that is based on filtering the token stream. This
dentation.

As expected, the IS-CFG based parser runs in linear time. It is
gometrically on average 1.73 times slower than the unmodified haskell-sr parser. There is a slight upward trend in the factor by which the IS-CFG based parser is slower than the unmodified parser, but this growth is slow given the log scale on the horizontal axis. Given that this is a prototype implementation with little optimization, the fact that the IS-CFG version is only a factor of two slower than the standard haskell-sr 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 (e.g., singletons) are common, 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 [An-
klesaria 2012] and indentparser [Kurur 2012] extensions to the Parsec parser library provide support for indentation-sensitive pars-
ing. 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 the parser monad to track the current indenta-
tion. 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 appropri-
ately indented tokens and passes only those tokens to the com-
binator for the expression to which the layout rule applies. This has a performance problem as each use of layout scans the remaining tokens in the input and this can lead to a quadratic running time.

Given that the layout combinator selects tokens before parsing has a chance to look at them, this technique also cannot support subex-
pressions, such as parenthesized expressions in Python, that are ex-
empt from layout constraints. This makes this approach incapable of expressing many real-world languages.

Erdweg et al. [2012] propose parsing indentation-sensitive grammars by filtering the parse trees generated by a GLR parser. The GLR parser generates all possible parse trees irrespective of layout. The indentation constraints for each parse node then remove the trees that violate the layout rules. Aside from the fact that they require a GLR parser and are generating parse trees that might not be used, a critical difference between their system and the one pre-

sent in this paper is that their indentation constraints are in terms of the set of tokens under a non-terminal whereas the system we present uses constraints between parents and their immediate chil-
dren. Thus, the two approaches look at the problem from different perspectives. Placing constraints on all tokens within an expression has a global nature while placing constraints between parent and child nodes has a more local and compositional nature. In any case, they use a GLR parser and 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 where.

While encoding indentation sensitivity this way is formally pre-
cise, 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 ex-
plicit in the grammar, and the grammar must carefully track which non-terminals produce or expect what sorts of whitespace. The au-
thors 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 understand and even more difficult to modify.

While the YAML approach bears some similarity to the tech-
nique proposed in this paper, a key difference is that their method uses the parameters of non-terminals to generate explicit whites-


space characters and thus incurs a significant accounting overhead in the design of the grammar. On the other hand, the system pre-

ent in Section 4 operates at a higher level, using the param-
eter to indicate the column or indentation at which non-terminals and terminals should occur. This is a subtle distinction, but it has a profound impact. As shown in Section 5, layout rules are compara-
atively simple to encode this way and, as shown in Section 5, this formalism is amenable traditional parsing techniques such as LR(k) parsing.

Note that none of the systems reviewed present an LR(k) pars-
ing algorithm. They either use 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 CYK, GLR and LR(k) parsers for this formalism. Though not shown here, SLR, LALR and LL(k) parsers can also

Figure 5: Parse times
be constructed. Experiments on a Haskell parser using this formalism show the parser runs between about times slower than a parser using traditional ad hoc techniques for handling indentation sensitivity, though improvements in the handling of indentation sets may reduce even this overhead. Using these techniques, not only can the layout rules of a wide variety of languages be easily expressed, but they can also be effectively parsed.

Acknowledgments

Feedback from Andrew Tolmach and Nathan Collins helped improve the presentation of this paper.

References

base version 4.5.1.0, June 2012. URL http://hackage.haskell.org/package/base/

Sam Anklesaria. indents version 0.3.3, May 2012. URL http://hackage.haskell.org/package/indents/


