Indentation-Sensitive Parsing for Parsec

Michael D. Adams
University of Illinois at Urbana/Champaign
http://michaeldadams.org/

Ömer S. Ağacan
TOBB University of Economics and Technology
http://osa1.net/

Abstract

Several popular languages including Haskell and Python use the indentation and layout of code as an essential part of their syntax. In the past, implementations of these languages used *ad hoc* techniques to implement layout. Recent work has shown that a simple extension to context-free grammars can replace these *ad hoc* techniques and provide both formal foundations and efficient parsing algorithms for indentation sensitivity.

However, that previous work is limited to bottom-up, LR(k) parsing, and many combinator-based parsing frameworks including Parsec use top-down algorithms that are outside its scope. This paper remedies this by showing how to add indentation sensitivity to parsing frameworks like Parsec. It explores both the formal semantics of and efficient algorithms for indentation sensitivity. It derives a Parsec-based library for indentation-sensitive parsing and presents benchmarks on a real-world language that show its efficiency and practicality.

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; Parsec; Indentation sensitivity; Layout; Offside rule

1. Introduction

Languages such as Haskell (Marlow (ed.) 2010) and Python (Python) use the indentation of code to delimit various grammatical forms. For example, 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. For example, 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, [])
```


Haskell '14, September 4–5, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/10.1145/2633357.2633369

The indentation of the bindings after each `where` keyword determines the 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`.

While Haskell and Python are well known for being indentation sensitive, a large number of other languages also use indentation. These include ISWIM (Landin 1966), occam (INMOS Limited 1984), Orwell (Wadler 1985), Miranda (Turner 1989), SRFI-49 (Möller 2005), Curry (Hanus (ed.) 2006), YAML (Ben-Kiki et al. 2009), Habit (HASP Project 2010), F# (Syme et al. 2010), Markdown (Gruber), reStructuredText (Goodger 2012), and Idris (Brady 2013a). Unfortunately, implementations of these languages often use *ad hoc* techniques to implement indentation. Even the language specifications themselves describe indentation informally or with formalisms that are not suitable for implementation.

Previous work on indentation sensitivity (Adams 2013) demonstrated a grammar formalism for expressing layout rules that is an extension of context-free grammars and is both theoretically sound and practical to implement in terms of bottom-up, LR(k) parsing. However, Parsec (Leijen and Martini 2012), like many combinator-based libraries, does not use the LR(k) algorithm. It is top-down instead of bottom-up and thus is outside the scope of that work. This paper extends that work to encompass such systems. We show that this extension both has a solid theoretical foundation and is practical to implement. The resulting indentation-sensitive grammars are easy and convenient to write, and fast, efficient parsers can be easily implemented for them. Our implementation of these techniques is available as the `indentation` package on the Hackage repository.

The organization and contributions of this paper are as follows.

- In Section 2, we review parsing expression grammars (PEG) and give an informal description of a grammar formalism for expressing indentation sensitivity.
- In Section 3, we demonstrate the expressivity of this formalism by reviewing the layout rules of Haskell and Python and then showing how to express them in terms of this grammar formalism.
- In Section 4, we formalize the semantics of PEG and define an indentation-sensitive, PEG-based semantics for this grammar formalism.
- In Section 5, we examine the internals of Parsec, show the correspondence between it and PEG, and demonstrate how to implement indentation sensitivity in Parsec.
- In Section 6, we benchmark our implementation on a real-world language, and we show it to be practical, effective, and efficient at defining layout rules.
- In Section 7, we review related work and other implementations of indentation sensitivity.
- In Section 8, we conclude.

Empty string	ε	
Terminal	a	where $a \in \Sigma$
Non-terminal	A	where $A \in N$
Sequence	$p_1 ; p_2$	
Lookahead	$!p$	
Choice	$p_1 \langle \vert \rangle\ p_2$	
Repetition	p^*	

Figure 1. Syntax of PEG parsing expressions

2. The Basic Idea

2.1 Parsing Expression Grammars

The basic idea for indentation sensitivity is the same as in Adams (2013) except that we aim to implement it for the top-down, combinator-based parsing algorithms used in Parsec. In order to do this, we base our semantics on parsing expression grammars (PEG) instead of context-free grammars (CFG) as they more closely align with the algorithms used by Parsec. In Section 4.1, we review the formal semantics of PEG, but at a basic level, the intuition behind PEG is simple. As in a CFG, there are terminals and non-terminals. However, in a CFG, each non-terminal corresponds to several productions that each map the non-terminal to a sequence of terminals and non-terminals. In a PEG, on the other hand, each non-terminal corresponds to a single *parsing expression*. Where in a CFG we might have the productions $A \to$ `'a'`A and $A \to$ `'b'`, in PEG we have the single production $A \to$ (`'a'`$; A) \langle \vert \rangle$ `'b'`.

The syntax of these parsing expressions is defined as shown in Figure 1 where p, p_1, and p_2 are parsing expressions. These operators behave as one would expect with minor adjustments for the choice and repetition operators. These two are special in that they are biased. The choice operator is left biased and attempts p_2 only if p_1 fails. Likewise, the repetition operator is greedy and, when possible, matches more rather than fewer repetitions. These biases ensure the uniqueness of the parse result, and thus PEG avoids the ambiguity problems that can arise with a CFG.

A number of other operators exist in PEG including optional terms, non-empty repetition (i.e., Kleene plus), positive lookahead, and a fail operator, but those operators are derived forms that are not needed in this paper.

2.2 Indentation Sensitivity

In order to support indentation-sensitive parsing, we first modify the usual notion of parsing by annotating every token in the input with the column at which it occurs in the source code. We call this its indentation and write a^i for a token a at indentation i.

During parsing we annotate each sub-tree of the parse tree with an indentation as in Figure 2. These annotations coincide with the intuitive notion of how far a block of code is indented. Thus, the sub-tree rooted at A^5 is a block indented to column 5. We then place constraints on how the indentations of sub-trees relate to those of their parents. This is formally achieved by introducing an operator $p^{\triangleright}$ that specifies that the indentation of a tree parsed by p must have the relation $\triangleright$ relative to that of its parent where $\triangleright$ is a given numeric relation. For example, we write $p^{>}$ to specify that a tree parsed by p must have a strictly greater indentation than its parent. In all other places, parent and child must have identical indentations. Note that the indentation of a sub-tree does not directly affect the indentation of its tokens. Rather, it imposes restrictions on the indentations of its immediate children, which then impose restrictions on their children and so on until we get to tokens. At any point, these restrictions can be locally changed by the $p^{\triangleright}$ operator.

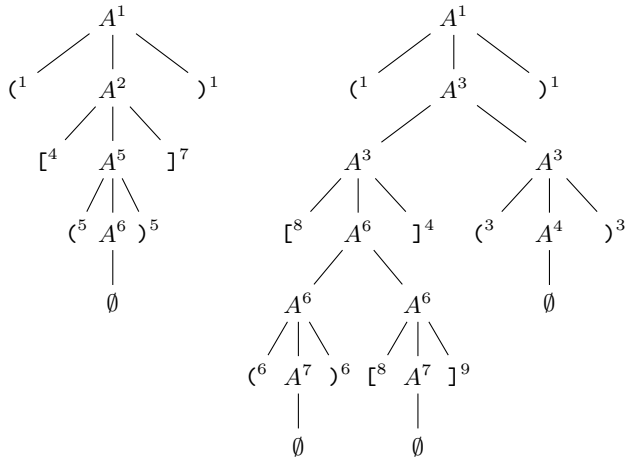

Figure 2. Parse trees for $(^1[^4(^5)^5]^7)^1$ and $(^1[^8(^6)^6[^8]^9]^4(^3)^3)^1$

As a simple example, we may write $A \to$ `'('`$; A^{>};$ `')'` 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 $A \to$ `'['`$^{\geq}; A^{>};$ `']'`$^{\geq}$ to mean the same except that `[` and `]` must be at an indentation greater than or equal to the indentation of the A on the left of the production arrow. In addition, we may write $A \to B^*$ to mean that the indentation of each B must be equal to that of A.

If we combine these, we can get a grammar for indented parentheses and square brackets as follows.

$$A \to \left(\texttt{'('}; A^{>}; \texttt{')'} \langle \vert \rangle\ \texttt{'['}^{\geq}; A^{>}; \texttt{']'}^{\geq}\right)^*$$

In that grammar, matching parentheses must align vertically, and things enclosed in parentheses must be indented more than the parentheses. Things enclosed in square brackets merely must be indented more than the surrounding code. Figure 2 shows examples of parse trees for this grammar on the words $(^1[^4(^5)^5]^7)^1$ and $(^1[^8(^6)^6[^8]^9]^4(^3)^3)^1$. In these parse trees, note how the indentations of the non-terminals and terminals relate to each other according to the indentation relations specified in the grammar.

While in principle any set of indentation relations can be used, we restrict ourselves to the relations $=$, $>$, $\geq$, and $\circledast$ as these cover the indentation rules of most languages. The $=$, $>$, and $\geq$ relations have their usual meanings. The $\circledast$ relation is $\{(i, j) \mid i, j \in \mathbb{N}\}$ and disassociates the indentation of a child from that of its parent.

Finally, indentation-sensitive languages typically have forms where the first token of a subexpression determines the indentation of the rest of the subexpression. For example, in Haskell the branches of a `case` must all align and have their the initial tokens at the same indentation as each other. To handle this, we introduce the $|p|$ operator, which behaves identically to p except that its indentation is always equal to the indentation of the first token of p. In the context of a CFG, this operator can be defined as mere syntactic sugar (Adams 2013). However, PEG's lookahead operator makes this difficult to specify as a desugaring. Thus we introduce it as a first-class operator and formally specify its behavior in Section 4.2.

3. Indentation-Sensitive Languages

Despite the simplicity of this framework for indentation sensitivity, it can express a wide array of layout rules. We demonstrate this by reviewing the layout rules for Haskell and Python and then show-

ing how they can be expressed as indentation-sensitive grammars. Though not shown here, sketches for other indentation-sensitive languages have been constructed for ISWIM, Miranda, occam,[1] Orwell, Curry, Habit, Idris, and SRFI-49. Those already familiar with the techniques in Adams (2013) can safely skip this section.

3.1 Haskell

3.1.1 Language

In Haskell, 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 delimit 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 3. Thus the context-free grammar only has to deal with 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

[1] The additional indentation relation $\{(i+2, i) \mid i \in \mathbb{N}\}$ is required by occam as it has forms that require increasing indentation by exactly 2.

```
L (<n>:ts) (m:ms) = ';' : (L ts (m:ms))   if m = n
                  = '}' : (L (<n>:ts) ms) if n < m
L (<n>:ts)    ms  = L ts ms
L ({n}:ts) (m:ms) = '{' : (L ts (n:m:ms)) if n > m
L ({n}:ts)     [] = '{' : (L ts [n])      if n > 0
L ({n}:ts)     ms = '{' : '}' : (L (<n>:ts) ms)
L ('}':ts) (0:ms) = '}' : (L ts ms)
L ('}':ts)     ms = parse-error
L ('{':ts)     ms = '{' : (L ts (0:ms))
L ( t :ts) (m:ms) = '}' : (L (t:ts) ms)
                    if m ≠ 0 and parse-error(t)
L ( t :ts)     ms =  t  : (L ts ms)
L       []     [] =  []
L       [] (m:ms) = '}' : L [] ms         if m ≠ 0
```

Figure 3. Haskell's `L` function (Marlow (ed.) 2010, §10.3)

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 "`}`" 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. Even minor changes to the error propagation of the parser can affect whether syntactically correct programs are accepted. While we may believe in the correctness of these parsers based on their many years of use and testing, the significant and fundamental structural differences between the language specification and these implementations are troubling.

3.1.2 Grammar

While the specification of Haskell's layout rule is complicated, it can be easily and intuitively specified using our indentation operators. By using these operators 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.

For example, Figure 4 shows productions that specify the `case` form and its indentation rules. With regard to terminals, we annotate most of them with an indentation relation of $>$ in order to allow them to appear at any column greater than the current indentation.

```
case  → 'case'^> ; exp ; 'of'^> ; (eAlts ⟨|⟩ iAlts)
eAlts → '{'^> ; alts^⊛ ; '}'^⊛
iAlts → (|alts|^*)^>
alts  → (alt' ⟨|⟩ alt) ; alt'^*
alt'  → ';'^> ; (alt ⟨|⟩ ε)
```

Figure 4. Productions for Haskell's `case` form

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. An exception to this rule is the closing curly brace (`}`) of an explicitly delimited block. Haskell's indentation rule allows it to appear at any column. Thus, `eAlts` annotates it with $\circledast$ instead of the usual $>$.

In Haskell, a block can be delimited by either explicit curly braces or use of the layout rule. In Figure 4, this is reflected by the two non-terminals `eAlts` and `iAlts`. The former expands to $\texttt{'\{'}^{>}$; $\texttt{alts}^{\circledast}$; $\texttt{'\}'}^{\circledast}$ where `alts` is a non-terminal parsing a semicolon-separated sequence of `case` alternatives. The $\circledast$ relation allows `alts` to not respect the indentation of the surrounding code.

The other non-terminal, `iAlts`, expands to $(|\texttt{alts}|^*)^>$. The $>$ relation increases the indentation, and the repetition operator allows zero or more $|\texttt{alts}|$ to be parsed. Due to the $>$ relation, these may be at any indentation greater than the current indentation, but they still must be at the same indentation as each other as they are all children of the same parsing expression, $|\texttt{alts}|^*$. The use of $|\texttt{alts}|$ instead of `alts` ensures that the first tokens of the `alts` are all at the same indentation as the $|\texttt{alts}|$ itself. Thus the alternatives in a `case` expression all align to the same column as each other. Note that because `iAlts` refers to `alts` instead of `alt`, we have the option of using semicolons (`;`) to separate clauses in addition to using layout. When using curly braces to explicitly delimit a block, semicolons must always be used.

Haskell has a side condition requiring every `case` to contain at least one `alt`. It cannot contain just a sequence of semicolons (`;`). This can be implemented either as a check after parsing or by splitting `alts` and $|\texttt{alts}|^*$ into different forms depending on whether an `alt` has been parsed.

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`).

Finally, GHC also supports an alternative indentation rule that is enabled by the `RelaxedLayout` extension. It allows opening braces to be at any column regardless of the current indentation (GHC 2011, §1.5.2). This is easily implemented by changing `eAlts` to be:

```
eAlts → '{'^⊛ ; alts^⊛ ; '}'^⊛
```

3.2 Python

3.2.1 Language

Python represents a different approach to specifying indentation sensitivity. It is explicitly line oriented and features `NEWLINE` in its grammar as a terminal that separates statements. The grammar uses `INDENT` and `DEDENT` tokens to delimit indentation-sensitive forms. An `INDENT` token is emitted by the lexer whenever the start of a line is at a strictly greater indentation than the previous line. Matching `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 (\), 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. ... The indentation of the continuation lines is not important. (Python, §2.1.6)

This means that `INDENT` and `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 `INDENT`, and the indentation of the third line should be compared to the indentation of the first line instead of the second line.

```
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.

3.2.2 Grammar

Though Python's specification presents its indentation rules quite differently from Haskell's specification, once we translate it to use our indentation operators, it shares many similarities with that of Haskell. The lexer still needs to produce `NEWLINE` tokens, but it does not produce `INDENT` or `DEDENT` tokens. As with Haskell, we annotate terminals with the default indentation relation $>$.

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

```
while_stmt → 'while'^> ; test ; ':'^> ; suite
```

A `suite` has two forms. The first is for multi-line statements, and the second is for single-line statements that are not delimited by indentation. The following productions handle both of these cases.

```
suite → NEWLINE^>  ; block^>
      ⟨|⟩ stmt_list ; NEWLINE^>
block → |statement|^*
```

When a `suite` is of the indentation-sensitive, multi-line form (i.e., using the left-hand side of the choice), 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`.

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

```
atom → '['^> ; listmaker^⊛ ; ]^⊛
```

Empty string	$(\varepsilon, w) \Rightarrow (1, \top(\varepsilon))$	
Terminal	$(a, aw) \Rightarrow (1, \top(a))$	
	$(a, bw) \Rightarrow (1, \bot)$	if $a \neq b$
	$(a, \varepsilon) \Rightarrow (1, \bot)$	
Non-terminal	$(A, w) \Rightarrow (n+1, o)$	if $(\delta(A), w) \Rightarrow (n, o)$
Sequence	$(p_1; p_2, w_1w_2u) \Rightarrow (n_1+n_2+1, \top(w_1w_2))$	if $(p_1, w_1w_2u) \Rightarrow (n_1, \top(w_1))$
		and $(p_2, w_2u) \Rightarrow (n_2, \top(w_2))$
	$(p_1; p_2, w_1w_2u) \Rightarrow (n_1+1, \bot)$	if $(p_1, w_1w_2u) \Rightarrow (n_1, \bot)$
	$(p_1; p_2, w_1w_2u) \Rightarrow (n_1+n_2+1, \bot)$	if $(p_1, w_1w_2u) \Rightarrow (n_1, \top(w_1))$
		and $(p_2, w_2u) \Rightarrow (n_2, \bot)$
Lookahead	$(!p, wu) \Rightarrow (n+1, \top(\varepsilon))$	if $(p, wu) \Rightarrow (n, \bot)$
	$(!p, wu) \Rightarrow (n+1, \bot)$	if $(p, wu) \Rightarrow (n, \top(w))$
Choice	$(p_1 \langle\vert\rangle p_2, wu) \Rightarrow (n_1+1, \top(w))$	if $(p_1, wu) \Rightarrow (n_1, \top(w))$
	$(p_1 \langle\vert\rangle p_2, wu) \Rightarrow (n_2+1, o)$	if $(p_1, wu) \Rightarrow (n_1, \bot)$
		and $(p_2, wu) \Rightarrow (n_2, o)$
Repetition	$(p^*, w_1w_2u) \Rightarrow (n_1+n_2+1, \top(w_1w_2))$	if $(p, w_1w_2u) \Rightarrow (n_1, \top(w_1))$
		and $(p^*, w_2u) \Rightarrow (n_2, \top(w_2))$
	$(p^*, w_1w_2u) \Rightarrow (n+1, \top(\varepsilon))$	if $(p, w_1w_2u) \Rightarrow (n, \bot)$

Figure 5. Semantics of PEG

4. Parsing Expression Grammars

In order to formalize our notion of indentation-sensitive parsing, we first review the formal semantics of PEG before extending it to support indentation sensitivity. In Section 5, we show how this semantics corresponds to and is implemented in Parsec.

4.1 Parsing Expression Grammars

Parsing expression grammars (PEG) are a modern recasting of top-down parsing languages (TDPL) (Aho and Ullman 1972) that has recently become quite popular and has a large number of implementations. Aside from the fact that PEG uses parsing expressions instead of productions, the main difference between PEG and CFG is that all choices are biased so there is only ever one possible result for an intermediate parse. For example, the choice operator, $\langle|\rangle$, is left biased. Ambiguous parses are thus impossible by construction.

From a practical perspective, this model makes it easy to implement PEG as a top-down parser where each terminal translates to a primitive, each non-terminal translates to a function, and the sequencing operator translates to sequencing in the code. In addition, the backtracking logic is relatively easy to implement. A choice operator first attempts to parse its left-hand side. Only if that fails does it backtrack and attempt to parse its right-hand side.

As formally defined by Ford (2004), a parsing expression grammar, G, is a four-tuple $G = (N, \Sigma, \delta, S)$ where N is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, δ is a finite production relation, and $S \in N$ is the start symbol. This much is identical to the traditional definition of a context-free grammar. The difference comes in how δ is defined. It is a mapping from a non-terminal symbol to a *parsing expression* and we write $A \to p$ if δ maps A to p. Unlike in CFG, there is only one p to which a given A maps, and thus we write $\delta(A)$ to denote that parsing expression.

The formal semantics for the operators in a parsing expression are given in terms of a rewrite relation from a pair, (p, w), of the parsing expression, p, and an input word, w, to a pair, (n, o), of a step counter, n, and a result, o. The result o is either the portion of w that is consumed by a successful parse or, in the case of failure, the distinguished symbol $\bot$. For the sake of clarity, when o is *not* $\bot$, we write it as $\top(w)$ where w is the parsed word. This rewrite relation is defined inductively as shown in Figure 5. Note that while the step counter is used to complete inductive proofs about PEG, it is not needed by the parsing process and can usually be ignored.

The intuition behind these rules is fairly straightforward. The empty parsing expression, ε, succeeds on any input in one step. A terminal parsing expression succeeds on an input where next token is the terminal that the parsing expression expects and fails otherwise. A non-terminal runs the parsing expression associated with that non-terminal. Sequencing succeeds and consumes w_1w_2 if the first parsing expression, p_1, consumes w_1 on input w_1w_2u and the second parsing expression, p_2, consumes w_2 on input w_2u. Lookahead succeeds only if p fails and fails otherwise. The choice form is one of the characteristic features of PEG and is left biased. If p_1 successfully consumes w on input wu, then the choice operator also succeeds by consuming w on input wu. Otherwise, if p_1 fails, then p_2 is run. The repetition operator is greedy. If p successfully consumes w_1 on input w_1w_2u and p^* successfully consumes w_2 on input w_2u, then p^* consumes w_1w_2 on input w_1w_2u. Otherwise, if p fails, then p^* succeeds while consuming no input.

4.2 Indentation Sensitivity

In order to add indentation sensitivity to the semantics of PEG, we need to pass information about layout to each parse. While it is tempting to think that this would just be the value of the current indentation, that is not sufficient. For example, suppose we are parsing the `iAlts` of a `case` expression and the `case` expression is at indentation 1. The body of that `iAlts` is allowed at any indentation greater than 1, but we do not know which indentation grater than 1 to use until `iAlts` consumes its first token. So,

Empty string	$(\varepsilon, w, I, f) \Rightarrow (1, \top^f_I(\varepsilon))$	
Terminal	$(a, a^i w, I, f) \Rightarrow (1, \top^{\nparallel}_{\{i\}}(a))$	if $i \in I$
	$(a, b^i w, I, f) \Rightarrow (1, \bot)$	if $a \neq b$ or $i \notin I$
	$(a, \varepsilon, I, f) \Rightarrow (1, \bot)$	
Non-terminal	$(A, w, I, f) \Rightarrow (n+1, o)$	if $(\delta(A), w, I, m) \Rightarrow (n, o)$
Sequence	$(p_1; p_2, w_1w_2u, I, f) \Rightarrow (n_1+n_2+1, \top^h_K(w_1w_2))$	if $(p_1, w_1w_2u, I, f) \Rightarrow (n_1, \top^g_J(w_1))$ and $(p_2, w_2u, J, g) \Rightarrow (n_2, \top^h_K(w_2))$
	$(p_1; p_2, w_1w_2u, I, f) \Rightarrow (n_1+1, \bot)$	if $(p_1, w_1w_2u, f) \Rightarrow (n_1, \bot)$
	$(p_1; p_2, w_1w_2u, I, f) \Rightarrow (n_1+n_2+1, \bot)$	if $(p_1, w_1w_2u, I, f) \Rightarrow (n_1, \top^g_J(w_1))$ and $(p_2, w_2u, J, g) \Rightarrow (n_2, \bot)$
Lookahead	$(!p, wu, I, f) \Rightarrow (n+1, \top^f_I(\varepsilon))$	if $(p, wu, I, f) \Rightarrow (n, \bot)$
	$(!p, wu, I, f) \Rightarrow (n+1, \bot)$	if $(p, wu, I, f) \Rightarrow (n, \top^g_J(w))$
Choice	$(p_1 \langle\vert\rangle p_2, wu, I, f) \Rightarrow (n_1+1, \top^g_J(w))$	if $(p_1, wu, I, f) \Rightarrow (n_1, \top^g_J(w))$
	$(p_1 \langle\vert\rangle p_2, wu, I, f) \Rightarrow (n_2+1, o)$	if $(p_1, wu, I, f) \Rightarrow (n_1, \bot)$ and $(p_2, wu, I, f) \Rightarrow (n_2, o)$
Repetition	$(p^*, w_1w_2u, I, f) \Rightarrow (n_1+n_2+1, \top^h_K(w_1w_2))$	if $(p, w_1w_2u, I, f) \Rightarrow (n_1, \top^g_J(w_1))$ and $(p^*, w_2u, J, g) \Rightarrow (n_2, \top^h_K(w_2))$
	$(p^*, w_1w_2u, I, f) \Rightarrow (n+1, \top^f_I(\varepsilon))$	if $(p, w_1w_2u, I, f) \Rightarrow (n, \bot)$
Indentation	$(p^{\triangleright}, wu, I, \nparallel) \Rightarrow (n+1, \top^f_{I'}(w))$	if $(p, wu, J, \nparallel) \Rightarrow (n, \top^f_{J'}(w))$ where $J = \{j \mid j \in \mathbb{N}, \exists i \in I, j \triangleright i\}$ $I' = \{i \mid i \in I, \exists j \in J', j \triangleright i\}$
	$(p^{\triangleright}, wu, I, \nparallel) \Rightarrow (n+1, \bot)$	if $(p, wu, J, \nparallel) \Rightarrow (n, \bot)$ where $J = \{j \mid j \in \mathbb{N}, \exists i \in I, j \triangleright i\}$
	$(p^{\triangleright}, wu, I, \Vert) \Rightarrow (n+1, o)$	if $(p, wu, I, \Vert) \Rightarrow (n, o)$
Absolute alignment	$(\vert p\vert, wu, I, f) \Rightarrow (n+1, o)$	if $(p, wu, I, \Vert) \Rightarrow (n, o)$

Figure 6. Indentation-sensitive semantics of PEG

instead of passing a single indentation, we must pass a set of allowable indentations. In our example, since the `case` expression is at indentation 1, the body of `iAlts` is passed the set $\{2, 3, 4, \cdots\}$ as the allowable indentations.

However, this is still not enough. Consider for example, the parsing expression `'a'`; $\left(\texttt{'b'}^{>} \langle\vert\rangle \varepsilon\right)$. If `a` occurs at indentation i in the input, then `b` must be allowed at only indentations strictly greater than i. This is even though `'a'` does not contain `'b'` and merely occurs sequentially earlier in the parsing expression.

Further, since PEG uses a biased choice, we must use the right-hand side of $\texttt{'b'}^{>} \langle\vert\rangle \varepsilon$ only if it is impossible to parse using its left-hand side. However, whether $\texttt{'b'}^{>}$ succeeds or not is entirely dependent on the indentation at which `'a'` succeeds. For example, on the input word $\texttt{a}^1\texttt{b}^2$, the parser for `'a'` succeeds at 1, and thus `'b'` can be attempted at any indentation greater than 1. Since 2 is in that range, the parser for `'b'` succeeds, and ε is never called. However, with the input word $\texttt{a}^3\texttt{b}^2$, the `a` token is at indentation 3, which restricts the allowed indentations for `'b'` to $\{4, 5, 6, \cdots\}$. Thus the parser for `'b'` fails, and ε is used.

In other words, since choices are biased, parses earlier in the input affect whether the left-hand side of a choice succeeds and thus whether the right-hand side should even be attempted. Thus indentation sets must be passed as both input and output in order to both control the indentations at which a parse is attempted and report the indentations at which it succeeds.

In addition to handling indentation relations, we must also handle the $|p|$ operator. This can be achieved by passing a flag to each parser indicating whether we are inside a $|p|$ that has not yet consumed a token. If we are, we must not change the current indentation set and thus ignore any $p^{\triangleright}$ operators.

We formally specify all this by generalizing the PEG rewrite rules to be a relation from a tuple (p, w, I, f) to a pair (n, o) where p is a parsing expression, w is an input word, $I \subseteq \mathbb{N}$ is an input indentation set, $f \in \{\Vert, \nparallel\}$ is an absolute-alignment flag, n is a step counter, and o is a result. The absolute-alignment flag is $\Vert$ to indicate that we are inside a $|p|$ that has not yet consumed a token and $\nparallel$ otherwise. The result o is either a pair of the portion of w that is consumed by a successful parse along with a result indentation set $I \subseteq \mathbb{N}$ and flag $f \in \{\Vert, \nparallel\}$ or, in the case of failure, the distinguished symbol $\bot$. When o is *not* $\bot$, we write it as $\top^f_I(w)$ where w, I, and f are respectively the parsed word, the output indentation set, and the absolute-alignment flag. Finally, the tokens in words are all annotated with indentations so $w \in (\Sigma \times \mathbb{N})^*$.

The rules from Figure 5 then straightforwardly generalize to the rules in Figure 6. The empty parsing expression, ε, succeeds on any input and so returns I and f unchanged. The terminal parsing

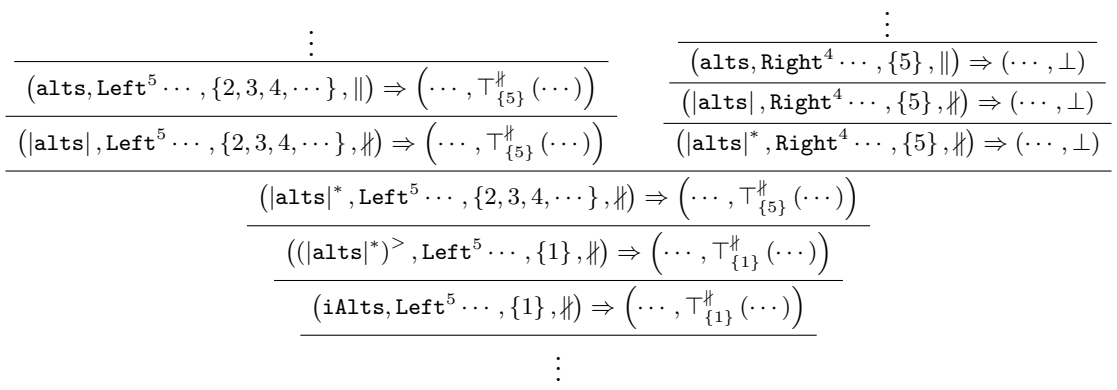

Figure 7. Example parse derivation

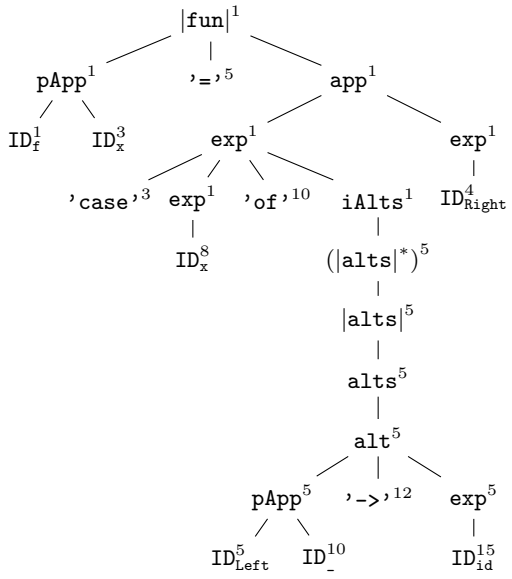

Figure 8. Example parse tree

expression, however, succeeds only when i, the indentation of the consumed token, is in the set of allowed indentations. Then, as a token has now been consumed, it clears the flag. In that case, it returns the singleton $\{i\}$ as the only indentation at which it succeeds. In all other cases, it fails.

The sequencing operator just threads the indentation set and flag through both p_1 and p_2. Lookahead is similar and just passes the indentation set and flag through unchanged. The choice operator passes the same indentation set and flag to both parsers.

The interesting cases here are the newly added operators for indentation, $p^{\triangleright}$, and absolute alignment, $|p|$. The indentation operator runs the parsing expression p with a new indentation set J computed according to $\triangleright$ and I. Specifically, every element of J is related by $\triangleright$ to some element of I. For example, if we have $p^{>}$ with $I = \{1, 2\}$, then $J = \{2, 3, 4, \cdots\}$. Once the parsing of p completes, the indentations at which it succeeded, J', are compared to the original indentation set, I, to see which elements of I are compatible according to $\triangleright$. Those elements of I are then returned in the output indentation set, I'.

An exception to this is when we are parsing in absolute mode. That is to say, when f is $\|$. In that case, the parent and child must have identical indentations despite the $p^{\triangleright}$ operator. Thus, the indentation set does not change, and the $p^{\triangleright}$ is effectively ignored.

Finally, the $|p|$ operator is trivial and merely sets the flag to $\|$.

4.3 Example Derivation

As an example of this semantics, consider parsing the following Haskell code with the productions in Figure 4.

```
f x =
  case x of
    Left _ -> id
   Right
```

Because `case` occurs at column 3, `Left` occurs at column 5, and `Right` occurs at column 4, the `Right` token should not be part of the `case` expression. Thus this code is equivalent to the following.

```
f x = (case e of Left _ -> id) Right
```

When initially parsing the right-hand side of `f`, the indentation set and flag will be $\{1\}$ and $\nparallel$. As the parser proceeds, it will consume the `case`, `x`, and `of` tokens. In the grammar, the terminals for these are annotated with the $>$ indentation relation, and in the input, the indentations of these tokens are all greater than 1. Thus, these tokens are successfully consumed without changing the indentation set or flag. Once we get to the `Left` token though, the current parsing expression will be `eAlts` $\langle|\rangle$ `iAlts`. Since the next token is not `{`, `eAlts` will fail and a parse of `iAlts` will be attempted.

At this point, indentation sensitivity starts to play a role. The fragment of the parse derivation for this part is shown in Figure 7. First, `iAlts` unfolds into $(|\texttt{alts}|^*)^{>}$. The $>$ relation means that we change from using the $\{1\}$ indentation set to the $\{2, 3, 4, \cdots\}$ indentation set. The $|\texttt{alts}|^*$ then calls $|\texttt{alts}|$, which in turn sets the flag to $\|$. With this flag set, intermediate indentation relations are ignored so the indentation set does not change until we get to the parsing expression that actually consumes `Left`. Though the terminal for consuming this token will be wrapped with the $>$ relation as explained in Section 3.1.2, this will be ignored as the flag is $\|$ at that point. Thus, when consuming the `Left` token, the indentation set is $\{2, 3, 4, \cdots\}$. Since the indentation of `Left` (i.e., 5) is in that set, the token is successfully consumed. The flag is then set to $\nparallel$, and the indentation set becomes $\{5\}$. This indentation set is used when parsing the remainder of the clause. Since terminals are wrapped by the $>$ relation, this means that each token in that clause is allowed at any column in the set $\{j \mid i \in \{5\}, j > i\} = \{6, 7, 8, \cdots\}$. This distinction between the first token of $|\texttt{alts}|$ (which must have an indentation equal to the indentation of $|\texttt{alts}|$

```
data IndentationRel = Eq | Ge | Gt | Any

localIndentation :: IndentationRel
  -> ParsecT (IndentStream s) u m a
  -> ParsecT (IndentStream s) u m a

absoluteIndentation ::
     ParsecT (IndentStream s) u m a
  -> ParsecT (IndentStream s) u m a

localTokenMode :: IndentationRel
  -> ParsecT (IndentStream s) u m a
  -> ParsecT (IndentStream s) u m a
```

Figure 9. Parsec combinators for indentation sensitivity

itself) and the other tokens of |`alts`| (which must have indentations greater than the indentation of |`alts`|) allows us to handle the distinction that Haskell makes between tokens at an indentation equal to the current indentantion (which start a new clause) and tokens at a greater indentation (which continue the current clause).

In Figure 7, once the remainder of that `alts` is parsed, the indentation set $\{5\}$ is threaded back out through |`alts`| to |`alts`|*. The indentation set and flag are then used in the second branch of |`alts`|* where the process proceeds as it did before. This time, however, the next token (i.e., `Right`) is at indentation 4, which is not an element of the indentation set $\{5\}$. Thus that token cannot be consumed, and the result is $\bot$. This causes the `case` expression to stop at this point and leaves the `Right` token for a surrounding function application to consume.

The final parse tree for this expression is then as shown in Figure 8. We can see in this tree how $\texttt{ID}^4_{\texttt{Right}}$ could not be a descendant of $(|\texttt{alts}|^*)^5$ as their indentations do not relate according to the relations specified in the grammar.

5. Parsec

With this formal model, we can now consider how to implement indentation sensitivity for Parsec. The basic types and operators that we add to Parsec are shown in Figure 9. The `IndentationRel` type represents an indentation relation where `Eq` is $=$, `Ge` is $\geq$, `Gt` is $>$, and `Any` is $\circledast$. The expression `localIndentation r p` applies the indentation relation `r` to `p` and corresponds to p^r. Likewise, `absoluteIndentation p` ensures that the first token of `p` is at the current indentation and corresponds to $|p|$. Finally, `localTokenMode` locally sets a default `IndentationRel` that is applied to all tokens. This eliminates the need to explicitly annotate the tokens in most productions.

To see how to implement these operations, first, we examine how PEG relates to Parsec. Then, we discus the practical implementation of the indentation-sensitive semantics in Parsec.

5.1 Parsec Internals

The semantics of PEG corresponds closely to the behavior of Parsec. Since this connection is not often made explicit, we now delve into the details of how Parsec is implemented and show how it corresponds to the PEG semantics.

Note that we are considering the *semantics* of PEG and Parsec and not their implementations. PEG implementations commonly cache the results of parses in order to ensure a linear bound on parsing time. Parsec does not do this, and relatively simple Parsec grammars can take exponential time. Nevertheless, though the implementation and the run times of these parsers can vary quite widely, the semantics of these systems correspond.

```
newtype ParsecT s u m a = ParsecT {
  unParser :: forall b.
       State s u
    -> (a -> State s u -> ParseError -> m b)
    -> (ParseError -> m b)
    -> (a -> State s u -> ParseError -> m b)
    -> (ParseError -> m b)
    -> m b
}

data State s u = State {
  stateInput :: s,
  statePos   :: SourcePos,
  stateUser  :: u
}
```

Figure 10. Data types for Parsec

In Parsec, a parser is represented by an object of type `ParsecT`. This type is shown in Figure 10. The `s` parameter is the type of the input stream. The `u` parameter is the type of the user state that is threaded through parser computations. The `m` parameter is the type of the underlying monad, and the `a` parameter is the type of the result produced by the parser.

The `State s u` parameter to `unParser` is the input to the parser. It is similar to the w in a $(p, w) \Rightarrow (n, o)$ rewrite and contains the input stream in the `stateInput` field. In addition, `statePos` contains the source position, and `stateUser` contains user-defined data.

The remaining parameters to `unParser` are continuations for different types of parse result. The continuations of type `a -> State s u -> ParseError -> m b` are for successful parses. The parameter `a` is the object produced by the parse. `State s u` is the new state after consuming input, and `ParseError` is a collection of error messages that are used if the parser later fails. On the other hand, the continuations of type `ParseError -> m b` are for failed parses where the `ParseError` parameter contains the error message to be reported to the user.

These two types of continuations are very similar to the success and failure continuations often used to implement backtracking. One difference, however, is that there are two each of both sorts of continuation. This is because by default Parsec attempts further alternatives in a choice operator only if the previous failures did not consume any input. For example, consider the parsing expression `('a';'b')` $\langle|\rangle$ `('a';'c')` on the input `ac`. The parsing expression `'a';'b'` will fail but only after consuming the `a`. Thus in Parsec, the failure of `'a';'b'` is a consumed failure, and the alternative parsing expression `'a';'c'` is not attempted.

Parsec also includes the `try` operator, which makes a consumed failure be treated as an empty failure. For example, if we use `(try ('a';'b'))` $\langle|\rangle$ `('a';'c')` on the same input, then the failure of `'a';'b'` is treated as an empty failure, and the alternative `'a';'c'` is attempted.

In the `ParsecT` type, the second and third arguments to the `unParser` function are continuations used for consumed success or consumed failure, respectively. Likewise, the fourth and fifth arguments are continuations used for empty success or empty failure, respectively. For example, the parser for the empty string does not consume any input and should thus always produce an empty success. Such a parser is easily implemented as follows, where `a` is the object to be returned by the parser, and `e` is an appropriately defined `ParseError`.

```
parserReturn a =
  ParsecT $ \s _ _ eOk _ -> eOk a s e
```

```
data Consumed a  = Consumed a
                 | Empty a

data Reply s u a = Ok a (State s u) ParseError
                 | Error ParseError
```

Figure 11. Data types for Parsec parse results

```
type Indentation = Int
infInd = maxBound :: Indentation

data IState = IState {
  minInd   :: Indentation,
  maxInd   :: Indentation,
  absMode  :: Bool,
  tokenRel :: IndentationRel
}
```

Figure 12. Data types for indentation sensitivity

This parser simply calls `eOk`, which is the continuation for empty success.

On the other hand, the parser for a character `c` consumes input and is implemented as follows, where `e1` and `e2` are appropriately defined `ParseError` objects.

```
parseChar c = ParsecT $ \s cOk _ _ eErr ->
  case stateInput s of
    (x : xs) | x == c ->
        cOk x (s { stateInput = xs }) e1
    _ -> eErr e2
```

This parser checks the input `s` to see if the next character matches `c`. If it does, `cOk`, the consumed success continuation, is called with an updated `State`. Otherwise, `eErr`, the empty failure continuation, is called.

The continuation passing style of `ParsecT` can be difficult to reason about, but we can convert it to direct style where it returns an object with different constructors for different kinds of results. Parsec provides such an alternate representation using the types in Figure 11. Thus, the `ParsecT` type is equivalent to a function from `State s u` to `m (Consumed (Reply s u a))`.

Represented in these terms, the correspondence between PEG and Parsec is straightforward. The Parsec parser contains extra information that is not present in PEG such as the `SourcePosition` and user state stored in the `State`, whether a parser consumes input or not, the monad `m`, and the result value of type `a`. However, if we elide this extra data, then a Parsec parser is simply a function from an input word stored in the `State` to either a successful or failed parse stored in `Reply`. This corresponds to a PEG rewrite $(p, w) \Rightarrow (n, o)$ from an input word, w, to either a successful or failed result, o.[2]

5.2 Indentation Sensitivity

Given the correspondence between PEG and Parsec, we can now implement indentation sensitivity in Parsec. The primary challenge here is the representation of the indentation set, I. Since this set may be infinitely large (such as at the start of p in $p^{>}$), we need to find an efficient, finite way to represent it. Fortunately, the following theorem allows us to construct just such a representation.

[2] There is still a difference in that a Parsec `Reply` stores the *remaining* input whereas in PEG o contains the *consumed* input, but these are equivalent in this context.

```
class (Monad m) => Stream s m t | s -> t where
    uncons :: s -> m (Maybe (t,s))
```

Figure 13. Code for the `Stream` class

```
data IStream s = IStream {
  iState :: IState,
  tokenStream :: s
}

instance (Stream s m (t, Indentation)) =>
         Stream (IStream s) m t where
  uncons (IStream is s) = do
    x <- uncons s
    case x of
      Nothing -> return Nothing
      Just ((t, i), s') ->
        return $ updateIndentation is i ok err
          where
            ok is' = Just (t, IStream is' s')
            err = Nothing
```

Figure 14. Code for `IStream` and its `Stream` instance

Theorem 1. *When parsing a parsing expression p that uses indentation relations only from the set $\{=, >, \geq, \circledast\}$, all of the intermediate indentation sets are of the form $\{j \mid j \in \mathbb{N}, i \leq j < k\}$ for some $i \in \mathbb{N}$ and $k \in \mathbb{N} \cup \{\infty\}$ provided the initial indentation set passed to p is also of that form.*

Proof. By induction over p and the step counter n. □

As a result of this theorem, each indentation set can be represented by a simple lower and upper bound. This leads to the `IState` type defined in Figure 12, which we thread through the parsing process to keep track of all the state needed for indentation sensitivity. The `minInd` and `maxInd` fields of `IState` represent the lower and upper bounds, respectively. The `infInd` constant represents when `maxInd` is infinite. The `absMode` field is used to keep track of whether we are in absolute alignment mode. It is `True` when the flag f would be $\|$ and `False` when it would be $\nparallel$. The `tokenRel` field stores a default indentation relation that surrounds all terminals. For example, in Haskell, most terminals are annotated with $>$ in the grammar. Since requiring the user to annotate every terminal with an indentation relation would be tedious and error prone, we can instead set `tokenRel` to `Gt`. Implementing the `localIndentation`, `absoluteIndentation`, and `localTokenMode` operators is then a simple matter of each operator modifying the `IState` according to the semantics in Figure 6.

The final consideration is how to thread this `IState` through the parsing process and update it when a token is consumed. The design of Parsec restricts the number of ways we can do this. The type `ParsecT` is parameterized by the type of the input stream, `s`, the type of the user state, `u`, the type of the underlying monad, `m`, and the result type, `a`. We could store an `IState` in the user state, `u`, and require the user to call some library function at the start of every token that then updates the `IState`. However, that would be a tedious and error prone process. On the other hand, for parsers that use Parsec's `LanguageDef` abstraction, adding the check to the `lexeme` combinator would handle many cases, but even then, many primitive operators such as `char`, `digit`, and `satisfy` do not use `lexeme` so we would have to be careful to also add checks to such primitives.

A more robust solution is to update the `IState` every time Parsec reads a token from the input. Parsec reads tokens using the `uncons` operation of the `Stream` class shown in Figure 13. Unfortunately, within this class we do not have access to the user state, `u`, and thus cannot store the `IState` there. We must store the `IState` in either the stream, `s`, or the monad, `m`. Normally, the monad would be the natural place to store it. However, the choice operator, $\langle|\rangle$, in Parsec does not reset the monad when the left-hand side fails. Thus any changes to the state made by the left-hand side would be seen in the parser for the right-hand side. This is not what we want. The `IState` used in the right-hand side should be the original one before any changes were made by the left-hand side. The `Stream`, `s`, is the only place where we can store the `IState`. Thus in Figure 14 we define a new stream type, `IStream`, that takes a stream of tokens paired with indentations and calls `updateIndentation` whenever a token is read by `uncons`. Given the current `IState`, `is`, the indentation of the current token, `i`, and success and failure continuations, `ok` and `err`, `updateIndentation` computes whether `i` is in the current indentation set. If it is, `updateIndentation` calls `ok` with a new `IState`, `is'`, that is updated according to the semantic rule for terminals from Figure 6. Otherwise, it calls `err`. This ensures that `updateIndentation` is called for every terminal and properly backtracks for operators such as $\langle|\rangle$.

Due to limitations of the Parsec interface, storing the `IState` here does have a significant drawback, however. In `uncons` there is no way to signal a parse error except by returning `Nothing`. Signaling some sort of error in the monad, `m`, will not work. Since `m` is the monad *inside* `ParsecT` and not the `ParsecT` monad itself, the error will not be caught by combinators such as $\langle|\rangle$ that should try alternatives when an indentation check fails.

Returning `Nothing` achieves the desired integration with the Parsec combinators, but it is not an ideal solution as that is also the signal for the end of a `Stream`. Since invalid indentation and input exhaustion are conflated, a parse could appear to finish and consume all of its input when it has merely met an invalidly indented token. Another problem is that if a parse fails due to an invalid indentation, the error message will be one for input exhaustion instead of one for an indentation violation. To remedy this problem, it is important to run `localTokenMode (const Any) eof` at the end of the parse to detect this situation and report an appropriate error message.

Alternative solutions would be to have the user insert explicit indentation checks or change the design of Parsec to allow `uncons` to signal errors other than input exhaustion. The latter option would require changes to Parsec as a whole but would make Parsec more flexible and is relatively straightforward.

6. Benchmarks

In order to test the practicality of this implementation of indentation sensitivity on a real-word language we converted the Idris 0.9.8 compiler to use our parsing library. While a Haskell compiler would have been a natural choice, in order to get a meaningful performance comparison, we needed to modify a language implementation that was already based on Parsec. The only Haskell implementation we found that does this is Helium, but Helium supports only a subset of Haskell forms. After considering several options, we chose Idris as its parser is based on Parsec and uses syntax and layout rules similar to those of Haskell.[3]

[3] More recent versions of Idris use Trifecta instead of Parsec. We have successfully ported our implementation to also work with Trifecta and used the resulting library to parse Idris code. However, that port is still in its infancy, and we do not have benchmark results for it yet.

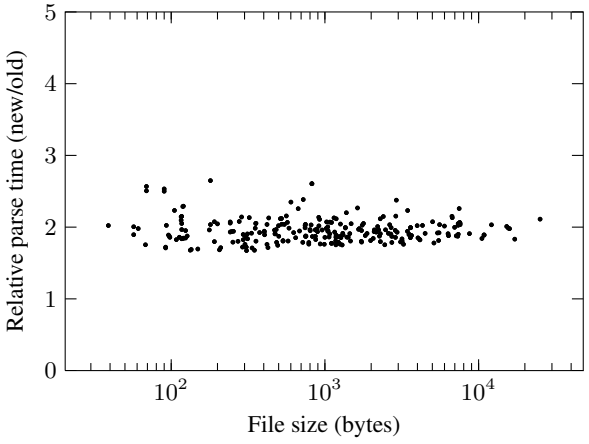

Figure 15. Initial benchmark results

6.1 Implementation

Porting Idris to use our library was straightforward. The changes mainly consisted of replacing the *ad hoc* indentation operators in the original Idris parser with our own combinators. Since our combinators are at a higher level of abstraction, this significantly simplified the parts of the Idris parser relating to indentation. In the core Idris grammar, approximately two hundred lines are dedicated to indentation. Those were replaced with half that many lines in our new system. In addition, this conversion fixed some rather significant bugs in how Idris's parser handles indentation. We describe these bugs in Section 6.3.

6.2 Testing

In order to test the performance of our parser, we tested it on Idris programs collected from a number of sources. These include:

- the Idris 0.9.8 standard library (Brady 2013e);
- the Idris 0.9.8 demos (Brady 2013c);
- the `Idris-dev` examples, benchmarks, and tests (Brady 2013d);
- the `IdrisWeb` web framework (Fowler 2013);
- the `WS-idr` interpreter (Brady 2013b);
- the `bitstreams` library (Saunders 2013); and
- the `lightyear` parsing library (Tejiščák 2013).

First, we tested that our parser produced the same abstract syntax trees as the original parser. In a few cases, it did not, but when we investigated, we found that these were all due to bugs in the implementation of indentation in the original Idris parser. In all other cases, we produced the same results as the original Idris parser.

Next, we benchmarked both parsers using Criterion (O'Sullivan 2012). The benchmarks were compiled with GHC 7.6.3 and the `-O` compilation flag. They were run on a 1.7GHz Intel Core i7 with 6GB of RAM running Linux 3.11.10. The results of our benchmarks are shown in Figure 15. For each parsed file, we plot the parse time of our new parser relative to Idris's original parser. Our parser ranged from 1.67 to 2.65 times slower than the original parser and averaged 1.95 times slower.

6.3 Analysis

One of the reasons our parser is slower is that, like Idris's original parser, we are scannerless. Thus, `uncons` checks the indentation of every single character of input. This is unlike Idris's original

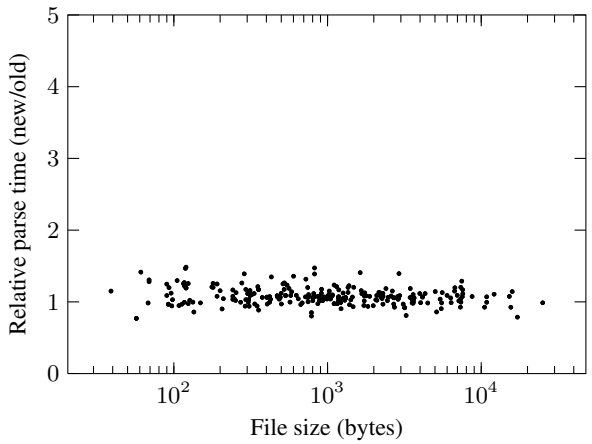

Figure 16. Benchmark results with modified indentation checks

parser, which checks the indentation at only certain manually-chosen points. As a result, however, the original parser has some significant bugs in how it handles indentation. In fact, we found several examples of Idris code that were erroneously parsed by the original parser. For example, in `IdrisWeb` we found the following code.

```
expr = do t <- term
          do symbol "+"
             e <- expr
             pure $ t + e
       `mplus` pure t
```

In this example, `mplus` occurs an an indentation that should cause it to be parsed as being outside both `do` expressions. The original Idris parser, however, does not check the indentation of the `mplus` lexeme. As a result, `mplus` is parsed as part of the last statement in the inner `do` expression. Since our new parser checks the indentation of every character, it does not have this problem.

In order to determine how much of the performance difference is due to this difference in where checks occur, we modified the original Idris parser to check all lexemes and our parser to check once per lexeme instead of once per character. We then reran the benchmarks. The Idris parser took on average 1.50 times what it did before, and our parser took on average 0.82 times what it did before. The relative performance of these modified parsers is plotted in Figure 16. Similar to before, for each parsed file, we plot the parse time of our parser relative to the Idris parser. Our parser ranged from 0.77 to 1.48 times slower than the original parser and averaged 1.07 times slower. Thus, once we account for the differences in where indentation checks occur, the performance of our parser is on par with that of Idris's parser.

7. Related Work

As discussed in Section 3.1.1, due to circularities introduced by `parse-error(x)`, the parsing technique that uses an `L` function as described in the Haskell language specification (Marlow (ed.) 2010) is generally not used by practical implementations. Instead, GHC and Hugs use shared state to coordinate between the lexer and parser. This relies on "some mildly complicated interactions between the lexer and parser" (Jones 1994) and is tricky to use. The resulting code is difficult to reason about, and minor changes to error propagation in the parser can affect parse results. Even worse, this technique embeds assumptions about the `L` function and does not easily generalize to other indentation rules.

The `uulib` parser library (Swierstra 2011) implements indentation using a similar approach, but it uses some intricate code involving continuations to handle the circularity between the lexer and parser. Like the previous approach, this is hard coded to Haskell-style indentation and cannot easily handle other layout rules.

The `indents` (Anklesaria 2012) library is an extension to Parsec that provides a combinator to store the current position in a monad for later reference. It then provides combinators to check that the current position is on the same line, the same column, or a greater column than that reference position. The `indentparser` (Kurur 2012) library is similar but abstracts over the type of the reference position. This allows more information to be stored than in `indents` at the cost of defining extra data types. In both systems, the user must explicitly insert indentation checks in their code. The resulting code has a much more operational feel than in our system. In addition, since these checks are added at only certain key points, the sorts of bugs discussed in Section 6.3 can easily arise. To the best of our knowledge there is no published, formal theory for the sort of indentation that these libraries implement.

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, Haskell, Idris, 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.

Our paper is an extension of the work in Adams (2013), but where that work focused on bottom-up, LR(k) parsing, this paper considers top-down parsing in Parsec and PEG.

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.

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.

8. Conclusion

This paper extends previous work on grammatical formalisms for indentation-sensitive languages to handle the top-down, combinator-based Parsec parsing framework. The resulting formalism is both expressive and easy to use. We use the connection between the semantics of PEG and Parsec to define a formal semantics for indentation sensitivity in these frameworks. Experiments on an Idris parser using this formalism show that, due to differences in how often the indentation is checked, the parser runs about twice as slow as a parser using *ad hoc* techniques. Once the differences in how often indentation is checked are eliminated, our technique performs on par with the *ad hoc* techniques. The resulting library is available on Hackage as the `indentation` package and provides convenient indentation-sensitivity for Parsec-based parsers.

References

Michael D. Adams. Principled parsing for indentation-sensitive languages: revisiting landin's offside rule. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 511–522, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. .

Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1972. ISBN 0-13-914556-7.

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

Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain't Markup Language (YAML) Version 1.2*, 3rd edition, October 2009. URL `http://www.yaml.org/spec/1.2/spec.html`.

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, September 2013a. ISSN 1469-7653. .

Edwin Brady. Ws-idr, February 2013b. URL `https://github.com/edwinb/WS-idr`. Commit `db65516b87863fcc0b066d26cb262bcddfff5514`.

Edwin Brady. idris-0.9.8-demos, December 2013c. URL `https://github.com/edwinb/idris-demos`. Commit `9c1355445dee0a41e6850a9c8d33cb0f2072cf78`.

Edwin Brady. idris-0.9.8-examples-benchmarks-tests, December 2013d. URL `https://github.com/idris-lang/Idris-dev`. Commit `869564663b8309a4984ba8ad700baf7b65c926bb`.

Edwin Brady. idris-0.9.8-stdlib, January 2013e. URL `https://github.com/idris-lang/Idris-dev`. Commit `a3c8020d50def27d7e1eb01d0ec8e10a00e9b90e`.

Leonhard Brunauer and Bernhard Mühlbacher. Indentation sensitive languages. Unpublished manuscript, July 2006. URL `http://www.cs.uni-salzburg.at/~ck/wiki/uploads/TCS-Summer-2006.IndentationSensitiveLanguages/`.

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In *Software Language Engineering*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012. URL `http://sugarj.org/layout-parsing.pdf`. To appear.

Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. .

Simon Fowler. idrisweb, December 2013. URL `https://github.com/idris-hackers/IdrisWeb`. Commit `0c823ff5af0fd9f04b66d05a138585acdc656722`.

The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.2.1. The GHC Team, August 2011. URL `http://www.haskell.org/ghc/docs/7.2.1/html/users_guide/`.

David Goodger. *reStructuredText Markup Specification*, January 2012. URL `http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html`. Revision 7302.

John Gruber. *Markdown: Syntax*. URL `http://daringfireball.net/projects/markdown/syntax`. Retrieved on June 24, 2012.

Michael Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Technical report, March 2006. URL `http://www.informatik.uni-kiel.de/~curry/report.html`.

HASP Project. The Habit programming language: The revised preliminary report, November 2010. URL `http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf`.

Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(03):323–343, July 1992. .

Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

INMOS Limited. *occam programming manual*. Prentice-Hall international series in computer science. Prentice-Hall International, 1984. ISBN 978-0-13-629296-8.

Mark P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, May 1994.

Piyush P. Kurur. indentparser version 0.1, January 2012. URL `http://hackage.haskell.org/package/indentparser/`.

P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966. ISSN 0001-0782. .

Daan Leijen and Paolo Martini. parsec version 3.1.3, June 2012. URL `http://hackage.haskell.org/package/parsec/`.

Simon Marlow (ed.). *Haskell 2010 Language Report*, April 2010. URL `http://www.haskell.org/onlinereport/haskell2010/`.

Egil Möller. *SRFI-49: Indentation-sensitive syntax*, May 2005. URL `http://srfi.schemers.org/srfi-49/srfi-49.html`.

Bryan O'Sullivan. Criterion version 0.6.0.1, January 2012. URL `http://hackage.haskell.org/package/criterion/`.

Python. *The Python Language Reference*. URL `http://docs.python.org/reference/`. Retrieved on June 26, 2012.

Benjamin Saunders. bitstreams, August 2013. URL `https://github.com/Ralith/bitstreams`. Commit `b4da0ea346d506e7fd9fc7b2c9637281addec9ba`.

S. Doaitse Swierstra. uulib version 0.9.14, August 2011. URL `http://hackage.haskell.org/package/uulib/`.

Don Syme et al. *The F# 2.0 Language Specification*. Microsoft Corporation, April 2010. URL `https://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.html`. Updated April 2012.

Matúš Tejiščák. lightyear, December 2013. URL `https://github.com/ziman/lightyear`. Commit `d74e48ad13451e763250ec1412989fdebe7af66a`.

D. A. Turner. *Miranda System Manual*. Research Software Limited, 1989. URL `http://www.cs.kent.ac.uk/people/staff/dat/miranda/manual/`.

Philip Wadler. An introduction to Orwell. Technical report, Programming Research Group at Oxford University, 1985.