Viability of extending parser-generator-frameworks for layout-sensitive languages

Project in Computer Science – EDAN70
January 18, 2016

Moritz Kobler
International exchange program, Faculty of Engineering, Lund University, Sweden
moritz.kobler@gmail.com

Abstract

While building parsers has become much easier in the last decades due to the development of tools that allow the generation of both scanners and parsers, such tools are still lacking if one is considering developing layout-sensitive languages. Existing parser-generators such as Beaver often have rather simple and easy to produce language specifications as input. Building a parser for layout-sensitive grammars on the other hand requires a lot of manual methods specific to the grammar. The reason for this dichotomy is partly the general predominance of research into context-free grammars.
There are languages currently in use that incorporate layout-sensitive syntax, such as *Python*, *Haskell* or *occam*. At the same time and more importantly there are also legitimate applications of layout-sensitivity including enforcement of the source code's readabilty as well as establishing parallelism between source code and program output. This paper discusses layout-sensitivity, its uses and usefulness and further argues that it is possible to build tools that will enable users to define their own layout-sensitive languages. Specifically, a first step towards facilitating the development of layout-sensitive languages is taken by implementing parsers for different languages using a decorator-approach on top of already existing tools: a preprocessor is interposed between the scanning- and parsing-phases in order to handle layout-sensitivity.

1. Introduction

While it is considered good style to use for example indentation in order to clarify meaning and structure of source code, in normal, context-free-grammars (CFG) the active use of whitespace is by no means necessary and is usually completely ignored during the scanning-phase of the compiler – and thus becomes irrelevant for the rest of the process. First mentioned by Landin in 1966, so called layout-sensitive languages (LSLs) actively necessitate certain whitespaces and indentations.
One fundamental example of layout-sensitivity (LS) for example is the offside-rule, also proposed by Landin which states that all tokens in an expression must be indented at least as far as the first token of that same expression (see 1.1 as an example). Some concrete examples of LSLs are *Python* [1], *occam* [2], and *Haskell* [3] and there exist many more. The *Haskell*-code in Listing 1.1 for example shows the organization of block-structures through the use of the offside-rule.
The problem of parsing a LSL is closely related to the problem of context-sensitivity. Other kinds of context-sensitivity being for example operator precedence – which can be solved by changing

```
n = a + b + c
 where a = 1
       b = c + d
        where c = 3
              d = 4
c = 5
```

Listing 1.1: Example code in Haskell, where blocks are structured using indentation.

the grammar of the language – or conditional directives [4]. While not all manifestations of LS do in fact entail context-sensitivity [4], it can be formally proven that languages using indentation to determine nesting levels, cannot be generally described by a CFG [5]. As Afroozeh and Izmaylova point out, even despite a great deal of research in the field of parsing, building parsers for LSLs especially remains a laborious task. One reason is that methods developed to parse CFGs are not applicable to context-sensitive grammars because they explicitly take advantage of the property of context-insensitivity to achieve their parsing result. Thus, tools based on such methods cannot (directly) be used to parse LSLs. Nonetheless, while for the longest time off-the-shelf parser generators that allowed non-experts to use parsers were built mainly for languages with CFGs [6], in recent years there has been a considerable development in the research on creating parser generators that support LS, mostly by extending already existing parser-tools such as *Parsec* [7].
The special quality of parser generators is that they allow the user to declaratively state the language's grammar without having to worry too much about how the parser will be implemented (of course there are still restrictions and rules to be considered). In order to extend these grammars to include LS, several authors have proposed mechanisms to allow for the relative positioning of the grammars' building blocks in relation to each other (more about related work can be found in section 6).

This paper aims to shed light on the questions (i) whether it is possible to generalize and abstract key features of layout-sensitivity in order to build an easy-to-use tool for the creation of such languages and (ii) if so, how powerful and versatile such a tool could be.

This will be done by implementing subsets of two layout-sensitive languages, *Python* and *occam*, in a modified parser generator framework and qualitatively assessing the generalizability of common features found in the examined languages. A preprocessor will be interposed between the scanner and the parser generated by the scanner-generator JFlex and parser-generator Beaver respectively. This preprocessor will be implemented on a case-to-case basis and will edit the token stream in order to resolve problems of context-sensitivity and output a tokens that can be interpreted according to a CFG. In a last step a syntax for expressing LSLs called *NEWLINE* will be developed and parsed by bootstrapping onto the explored technique of parsing LSLs in order to demonstrate the feasibility of the first steps in the creation of a layout-sensitive parsing tool.

Section 2 of this paper will talk about LS in greater depth and existing approaches for parsing LSLs. In section 3 the implementation of the layout-sensitive-parser as developed during the research will be treated. Section 4 will cover the qualitative evaluation of said parser whereas results will be presented in section 5. In sections 6 and 7 related and future work will be discussed.

2. Background

A first step towards answering the questions of this paper is to understand what LS is and what approaches exist to parse layout-sensitive grammars.

2.1 Layout-sensitivity

Simply put, LS describes the fact that the layout of some given source code (i.e. the positioning of the token-representations) is necessary for its correct interpretation. It entails the active use of whitespace such as spaces, tabs or line breaks. The examples in listings 2.1 to 2.3 – which are only code snippets and not necessarily functioning code – illustrate the difference between layout-sensitive and layout-insensitive syntax and reveal some of the purposes of LS:

1. The most obvious reason for using LS is to enforce readable and structured code. The Java-code in sub-listings 2.1e and 2.1d have the exact same meaning – but the one using (optional) indentation much more clearly conveys the structure of the source code. While in this simple example both are in fact easy to read, it is not hard to imagine more complex code where readability would be severely inhibited – this especially holds true for languages like C where code can oftentimes appear quite cryptic.
2. Solving ambiguity issues such as the dangling-else-problem (see listing 2.2, where indentation is used to clarify to which if-clause the else-clause belongs instead of the insertion of an end-of-if-specifier) or avoiding unnecessary curly braces to delimit blocks or semicolons to delimit statements, is a more technical reason for the use of LS. Comparing sub-listings 2.1a and 2.1c to sub-listings 2.1b and 2.1d respectively showcases exactly this. While *Java* has need of aforementioned curly braces and semicolons, *Python* uses its layout to serve the same purpose. The two *Python*-examples also show that discarding whitespace would turn both snippets into the same (ambiguous) piece of code.
3. Helping the programmer to better see the output of the program by organizing the source code in a specific way, is perhaps the most pure reason to use LS since it can by definition only be solved through LS. This *parallelism* between source code and program output may at first seem very abstract. Looking for example at the markup language Markdown [8], which translates between a plain text formatting syntax and HTML, the matter becomes clearer. In some instances, changing the indentation in the source code will result in a parallel change of the output: listing 2.3 illustrates how the positioning of the second list-element directly translates to the output. Another example could be a language to specify layout-sensitive grammars. By extending already existing specification languages with layout-sensitivity, the layout of the source code could be used to mirror what the language that is to be specified should look like.

```
if x:
  x=0
y=5
```

a: Language: *Python*.

```
if (x) {
  x=0;
}
y=5;
```

b: Language: *Java*.

```
if x:
  x=0
  y=5
```

c: Language: *Python*.

```
if (x) {
  x=0;
  y=5;
}
```

d: Language: *Java*.

```
if (x){x=0;y=5;}
```

e: Language: *Java*.

Listing 2.1: Simple if-conditions in *Python* and *Java* illustrating the differences between LSLs and layout-insensitive languages.

```
if x:
  if y:
    z=0
  else:
    z=1
```

Listing 2.2: Example code in *Python* where indentation is used to resolve the dangling-else-problem.

```
List:
- 1st level
   - 2nd level
```

a: *Markdown* source code.

List:

- 1st level
    - 2nd level

b: Resulting output.

Listing 2.3: Simple list-definition in *Markdown*. The indentation in the source code directly influences the output of the program.

2.2 Existing approaches to parsing LSLs

One main approach to parsing LSLs is generalized parsing. Given an ambiguous grammar and a certain token stream, the basic idea of general parsing is to simply create all possible parse trees, called a parse forest. Only afterwards is it decided which of the parse-trees was the intended one in an additional disambiguation phase [6]. In the context of layout-sensitivity, the layout-conditions could be imposed upon the resulting parse forest and all parse trees that violate the conditions are discarded (until either one valid parse tree remains or all are discarded which leads to an invalid program syntax).
Another approach is interposing a preprocessor between the scanner and the parser. While not exactly the same, such a preprocessor can be compared to a decorator as described by the decorator-design-pattern [9]. The preprocessor will take the token stream produced by the scanner as input and add elements to it if necessary. It can also be argued that such a preprocessor is rather a collection of decorators, each with a special function that it chooses to apply if

the situation necessitates. The decorated token stream will then be given to the parser that tries to produce a parse tree. The approach is visualized in figure1. The essential point in the process is that the token stream outputted by the preprocessor will conform to a CFG. It is the decorator that handles any context-sensitivity.

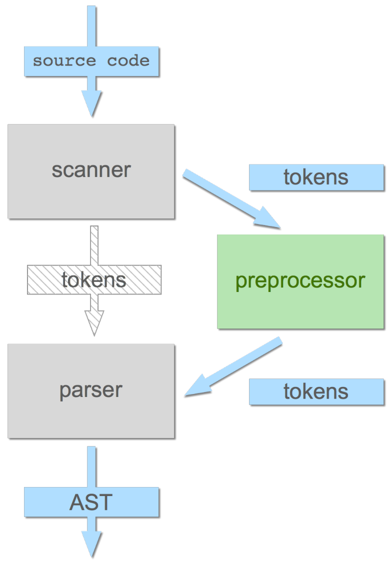

Figure 1: Diagram of the decorator-approach to parsing LSLs. Instead of tokens being given from scanner to parser directly, a preprocessor is interposed.

3. Implementation of layout-sensitive parsing

The approach chosen in this research project was to interpose a preprocessor between the scanner and the parser, from now on referenced as the decorator-approach. It was chosen for several reasons:

- There are already very good and easy-to-use tools allowing for an easy implementation of both scanners and parsers. Using a decorator-approach takes advantage of these tools and extends them.
- Separating scanner and parser allows for high versatility and modularity, since tools that become less well-maintained or simply out of date can be easily replaced by more current ones.
- It seems a very intuitive solution to take the original token stream and edit it in order to fulfill the parser's requirement. This makes it easy to comprehend and implement.
- While other approaches such as generalized parsing have limited practical applicability (due to efficiency-considerations in the case of generalized parsing), this is not the case with the decorator pattern.

Specifically, the tools chosen were *JFlex* and *Beaver*. JFlex is a scanner generator that takes a specification with a set of regular expression and outputs a scanner written in Java [10]. Beaver is a LALR(1) parser generator taking a CFG in Extended Backus-Naur form and outputting a Java-based parser for the grammar [11].
As described in the previous section, the basic idea was to build a preprocessor that would take the token stream as outputted by JFlex and produce a new token stream fulfilling the requirements of Beaver by adding certain tokens at the appropriate place. Looking at the tools chosen, using Java as a base for the preprocessor was the obvious choice. The entire project can be found online [12].

3.1 Languages

The next question presenting itself was which languages, or subsets of which languages, were to be used in order to be able to find solutions for general features used in many LSLs. In the end, the choice fell on *Python* and *occam*. *Python* on the one hand is a good example of a LSL that is currently in use and widely known and points of interest include block-structures and forced line breaks (i.e. forcing the use of line breaks in order to achieve for example a separation of statements). On the other hand, *occam* is a much less known language, ”designed to express concurrent algorithms and their implementation on a network of processing components” [2]. It has nonetheless interesting features in regard to its LS such as continuation lines and exact indentation requirements of blocks (it necessitates an indentation of exactly two spaces, see listing 3.1 for an exemplary code-snippet). The subsets examined were then chosen by the relevance of the features in regard to LS. In the case of *Python* this meant, that a program could be a collection of statements, with one statement type being full fledged if-statement allowing nesting – thus both forced line breaks and nested blocks were considered. The relevant part of the *occam*-subset included assignments with possible continuation lines and sequence definition that allowed for exactly indented blocks.

```
SEQ
  x := 0
  y := 1
```

Listing 3.1: Simple sequence-definition in *occam*. The block-structure has to be indented exactly two spaces.

Two other candidate-languages should be mentioned. Firstly, there is *Haskell* which is also very well known, especially with functional programming languages currently getting a lot of attention. But it might not in fact be the best example of a language that makes use of LS since code can also be written in a layout-insensitive way by using curly braces and semicolons. Another problem lies in the parsing-process of the *Haskell* language: a clear separation of scanning and parsing in Haskell is not entirely possible. Certain circularity-issues require the interaction of scanner and parser and thus prohibit a linear process. This is the reason why practical implementations of *Haskell* compilers often merge scanning and parsing phases [13]. Considering the decorator-approach of this research, *Haskell* did not seem an appropriate choice. Secondly, there is *Markdown* which very nicely illustrates the parallism-argument for the use of LS. However, there is no standard way of interpreting input, since it has never been properly defined, and is thus very heterogeneous with many different possible syntaxes. Another issue is that Markdown produces no parsing errors because every part of the source code is just translated into HTML – even though it might not make for a great result.
In a last step of the research, an independent language titled *NEWLINE* was developed that specifies a syntax for expressing LS grammars and could potentially be used in a LSL-tool. Using the insight and code from the previous efforts of building parsers for *Python* and *occam* based on the decorator-approach, a parser for *NEWLINE* was programmed. Listing 3.2 shows exemplary code in the *NEWLINE*-syntax where the right-hand-side of first production (lines 1&2) specifies that statements are seperated by line breaks, the third production (lines 6&7) defines an if-statement as consisting of an if-part and an equally indented else-part – the single quotation mark serves as a point of reference – and the forth and

fifth productions (lines 9&10 and 12&13 respectively) specify that the statements inside if- and else-blocks be further indented than the reference points.

```
1  stmtlist     =   stmt | 'stmtlist
2                          stmt
3
4  stmt         =   ifstmt | ...
5
6  ifstmt       =   'ifsuite
7                   (=)elsesuite
8
9  ifsuite      =   'IF cond
10                      stmtlist
11
12 elsesuite    =   'ELSE
13                      stmtlist
```

Listing 3.2: Possible *NEWLINE* grammar specification for if-statements. Indentation is used to convey information about layout-sensitivity in the output-language.

3.2 Insertion of relevant tokens

The insertion of the relevant tokens was managed by analyzing the different roles every token might play in regard to features of layout-sensitivity and then having the same methods executed at each instance of a certain role. Consider aforementioned example of a simple if-clause from listing 3.3, and a grammar as specified in listing 3.4.

```python
1  if x: # IF-token triggers block
2    x=0 # ID("x")-token starts block
3  y=5   # ID("y")-token ends block
```

Listing 3.3: A simple if-statement in *Python* illustrating that tokens can be assigned several roles.

```
program      =   stmt_list;

stmt_list    =   stmt |
                 stmt_list NEWLINE stmt;

stmt         =   if_stmt | assign_stmt;

if_stmt      =   IF expr COLON
                     INDENT stmt_list DEDENT;

assign_stmt  =   ID ASSIGN NUMERAL;

expr         =   ID;
```

Listing 3.4: Simple grammar-specification using Beaver-conform syntax.

Assuming JFlex discards any whitespace but line breaks, the token stream would be missing two tokens in the if-statement that have to be inserted: the `INDENT`- and `DEDENT`-tokens that delimit the block-structure. Although they were named *INDENT* and *DEDENT*, they do not as such represent any whitespace in the source code and could obviously be arbitrarily named. Looking at the roles of the tokens, it is clear that the `IF`-token in the example is both trigger of the coming block as well as reference point for any indentation inside the block. The first token of a statement-list, i.e. any element of `FIRST(stmt_list)`, is the token deciding whether the statement is inside or outside the if-clause. Taking into account the two statement-types, if- and assignment-statements, we get the following:

`FIRST(stmt_list) = ... = {ID, IF}`

Considering that the block of an if-statement contains a statement-list and a program can be comprised of several consecutive statements, an `ID`-token in the example can have two different roles: (i) starting a block-structure (as seen at the beginning of line 2) and (ii) ending a block-structure (as seen at the beginning of line 3). Thus, when reading an `ID`-token, the relative indentation to the reference point needs to be checked, and in the case of a bigger indentation, an `INDENT`-token needs to be inserted before the token whereas in the case a smaller indentation the insertion of a `DEDENT` token is required. In the example this would lead to an if-statement parsable by the above production-rule.[1] These same considerations would also need to be made for nested or consecutive if-statements, which would assign additional roles to the `IF`-token, since `IF` $\in$ `FIRST(stmt_list)`. In principle this process of defining relevant waypoints needs to be repeated for every layout-sensitive feature that requires the insertion of tokens. In addition, possible inter-dependencies between these different decorator methods, mainly caused by single tokens being assigned multiple roles, need to be checked and if needed resulting errors solved on a case-to-case-basis.

3.3 Practical considerations

While the premise of LS is to not discard whitespace, in the practical implementation of the parsers some whitespace such as spaces were in fact discarded by the scanner. At the same time, JFlex itself provided column- and line-counting methods that assigned corresponding values to every token. These values were then used in the preprocessor to determine the indentation relationships.
In all the implementations where the use of tabs was supposed to be supported, those tabs were not discarded by the scanner. This had the simple reason that the provided counting-methods actually counted characters and as such tabs where considered to have a width of 1 which lead to unwanted errors. Thus, the preprocessor needed to perform a column-correction, given some arbitrary width of a tab, and delete all `TAB`-tokens post-scanning.
In the exemplary implementation in section 3.2, line breaks were not discarded. This was also the approach used in the parser for the *NEWLINE*-language. It has the advantage of not having to compare line-counts as well as column-counts, but to simply incorporate the NEWLINE token in the parser specification. It does require the deletion of some unnecessary `NEWLINE`-tokens in certain cases though and some preprocessor-methods need to be programmed containing an offset in order to account for the `NEWLINE`-tokens. Contrarily, in some of the implementations for *Python* and *occam* the line breaks are in fact discarded by the scanner. While using this approach necessitates the comparison of line-counts at the appropriate places, it also seems to be a more consequent implementation in the case of block-structures. Both approaches are viable as long as the preprocessor code is adapted accordingly.

[1] In this specific case, the `COLON`-token could have been used in some form as well – such a delimiting token is not always at hand though, as can be seen at an *occam*-sequence.

4. Qualitative evaluation

The qualitative evaluation was made with the paper's main questions in mind of whether it is possible to create a tool that makes building LSLs easier and if so, how versatile such a tool can be. The possibility of quantitative evaluation was considered, for example in the form of a comparison of the same languages (or subsets), once using a layout-sensitive syntax implemented in the JFlex and Beaver environment and once using layout-insensitive syntax within the same environment with the difference of the preprocessor being interposed. It was concluded that the effort of doing this would outweigh the gain and a qualitative evaluation would contribute to a better understanding of the results.

4.1 Generalizability

Having considered only small subsets of languages or only small languages, it is hard to judge the overall generalizability. It was mostly possible to find general solutions for the main features of layout-sensitivity as studied in this research project, namely block-structures, forcing of line breaks and continuation lines. Especially in the case of blocks, a generalization could be achieved regardless of what kind of indentation was required (e.g. exact indentation, bigger indentation, same indentation etc.).

4.2 Difficulties encountered

Even given the overall success at generalizing key concepts, it is important to note that there were considerable difficulties even at these small scales. There were two kinds of problems that arose during development. The first kind was of a technical nature, and mainly caused by the choice of scanner- and parser-generator, such as an initial issue disallowed the mixing of tabs and spaces due to a overly simple method of counting columns by JFlex or the question whether it is better to let the scanner recognize line breaks and insert an appropriate token. Issues of this first kind were largely solved and do not inhibit generalizability.
The second, and far more relevant set of problems were of a conceptual nature and had the following two reasons:

- Interconnectedness of layout-sensitive features (e.g. a continuation line within a block)
- The same token can have many different roles (e.g. beginning and ending of a block)

Even in the minimal example in section 3.2, problems of this kind appear, as several tokens are assigned multiple roles. Imagining larger languages with multiple different instances of layout-sensitive features would only worsen such issues. In the research project these problems often had to be solved using conditions specific for each case and were thus not dealt with in a general fashion. This however does not necessarily mean these problems **cannot** be solved in a general way, just that it might not be trivial and it was not possible in the given time frame.

5. Results

During this research, key features of LSLs were identified, namely

- block structures (e.g. nested statements inside if-clauses),
- forced line breaks (the use of line breaks as a requirement to separate certain elements such as statements), and
- continuation lines (after specific tokens a language might allow line breaks that escape the usual treatment in regard to LS, in order to avoid extremely long lines of code)

Moreover, general patterns of how these features can be described and implemented, were found and analyzed. Parsers for three different LSLs – small subsets of the existing languages *Python* and *occam* as well as a new language called *NEWLINE* that can be used to specify layout-sensitive grammars – with said features were implemented by extending already existing tools for parsing context-free languages. While implementing those parsers required some specificity for each implementation, it is believed that the development of an easy-to-use tool for parsing LSLs is indeed possible. It remains open however, how versatile such a tool can be, since only small subsets of languages have been examined. In particular, more complex grammars with many different kinds of layout-sensitive features, and where single tokens are assigned many different roles in solving the context-sensitivity, will need to deal with a high degree of interdependencies which could complicate the implementation considerably.

6. Related work

As mentioned in the introduction, there has been some important work on layout-sensitive parsing in recent years. In his paper, Adams proposes an extension to CFGs that can express layout-rules such as the offside rule and does some theoretical work on deriving algorithms for layout-sensitive generalized LR (GLR) and LR(k) parsing [14]. Afroozeh et al. develop a framework dealing with indentation-sensitivity among other things on the basis of a generalized LL parsing algorithm [4]. In their work, Brunauer and Mühlbacher show that use of indentation for determining nesting level cannot be described by a CFG and moreover also present a way of extending CFGs to handle indentation-sensitivity and introduce a method to build an efficient LL recursive descent parser from it [5].
A very visual approach was taken by Erdweg et al. They propose a theoretical generalization of block-structures via delimiting polygons and the layout-conditions these polygons adhere to. Furthermore, they use generalized parsing techniques to handle the parsing of indentation-sensitive syntax and develop a framework to allow users to specify such a layout-sensitive grammar [6].

7. Future work and concluding discussion

Layout-sensitivity is not a recent concept and is currently used in several languages. Moreover, there are legitimate reasons for using LS. Still, writing parsers for such languages is much more work-intensive than implementing parsers for languages based on CFGs. In this research a first step has been taken towards the goal of simplifying the development of LSLs. By building parsers for three different languages (NEWLINE and subsets of *Python* and *occam*) using existing scanner- and parser-generators and interposing a preprocessor to decorate the token stream, case-to-case solutions could be implemented. Furthermore, the research showed that certain parts of the code could be factored out and reused in all three cases.
Future work building on this research specifically would entail finding a general solution for building LSLs using the decorator-approach and, if possible, developing a preprocessor-generator. Ideally, such a tool would take a layout-sensitive language-specification as input (for example written in the *NEWLINE*-syntax) and generate (i) the according scanner specification (e.g. for JFlex), (ii) the preprocessor, and (iii) the appropriate parser specification (e.g. for Beaver).
While researching the topic of LS, the question whether a layout-sensitive syntax is even desirable and worth the effort, was raised at several occasions. In the end it is important to consider that there are in fact several prominent examples of LSLs currently in use. Thus, the ongoing interest in LS alone makes it worthwhile to facilitate the development of languages using LS. Moreover, research in the direction of context-sensitive-grammars is worth something

in itself. It is possible that new concepts can be derived, furthering parsing in its entirety. One might compare it to research into methods for generalized parsing, where an everyday user might find only limited applicability, whereas many applications can be derived in more advanced settings.

References

[1] Python Software Foundation. The python language reference, 2012. URL `https://docs.python.org/3.5/reference/`. Retrieved on January 1, 2016.

[2] Geoff Barrett. occam 3 reference manual, 1992. URL `http://wotug.org/occam/documentation/oc3refman.pdf`. Retrieved on January 1, 2016.

[3] Simon Marlow. Haskell 2010 language report, 2010. URL `https://www.haskell.org/onlinereport/haskell2010/`. Retrieved on January 1, 2016.

[4] Ali Afroozeh and Anastasia Izmaylova. One parser to rule them all. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2015*. Association for Computing Machinery (ACM), 2015. URL `http://dx.doi.org/10.1145/2814228.2814242`.

[5] Leonhard Brunauer and Bernhard Mühlbacher. Indentation sensitive languages, July 2006. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.2933&rep=rep1&type=pdf`. Retrieved on January 9, 2016. Unpublished manuscript.

[6] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, pages 244–263, 2012. . URL `http://dx.doi.org/10.1007/978-3-642-36089-3_14`.

[7] Daan Leijen. *Parsec a fast combinator parser*. University of Utrecht, Dept.of Computer Science, Utrecht, The Netherlands, October 2001.

[8] John Gruber. Markdown. URL `https://daringfireball.net/projects/markdown/`. Retrieved on January 9, 2016.

[9] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN 0-201-63361-2.

[10] Steve Rowe Gerwin Klein and Régis Décamps. *JFlex User's Manual*, 2015. URL `http://jflex.de/manual.html`. Retrieved on January 9, 2016.

[11] *Beaver - a LALR Parser Generator*. URL `http://beaver.sourceforge.net/index.html`. Retrieved on January 9, 2016.

[12] Moritz Kobler. Layout-sensitive parsing using jflex and beaver, 2016. URL `https://bitbucket.org/edan70/2015-layout-sensitive-parsing`. Retrieved on January 11, 2016.

[13] Michael D. Adams and Ömer S. Ağacan. Indentation-sensitive Parsing for Parsec. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell - Haskell 14*. Association for Computing Machinery (ACM), 2014. URL `http://dx.doi.org/10.1145/2633357.2633369`.

[14] Michael D. Adams. Principle parsing for indentation-sensitive languages. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 13*. Association for Computing Machinery (ACM), 2013. URL `http://dx.doi.org/10.1145/2429069.2429129`.