# PERFORMANCE IMPACT OF THE MODULAR ARCHITECTURE IN THE INCREMENTAL SGLR PARSING ALGORITHM

RESEARCH PROJECT
TU DELFT

**Mara Stefania Coman** **Eelco Visser** **Jasper Denkers**

**Daniel Pelsmaeker**

June 27, 2021

## ABSTRACT

JSGLR2 is a modular Java implementation of the SGLR parsing algorithm that supports systematic benchmarking and improvement of its several parsing variants. By splitting the code into several components, they can be tested in isolation and thus optimized more effortlessly. The modular architecture, although beneficial for efficiently identifying and implementing optimizations, negatively impacts the performance of the parsing algorithm. This paper aims to measure the overhead introduced by the code architecture for one of the variants, more specifically the incremental variant, which combines incremental parsing with SGLR parsing. It does so by comparing the original implementation with a version with the modularity removed. The evaluation is done on programming languages used in practice: Java, WebDSL and SDF3. The results show that the inlined parser outperforms the previous one, achieving speedups of up to 16% in batch parsing and up to 10% in incremental parsing.

***Keywords*** Parsing · Incremental parsing · SGLR · ISGLR

# 1 Introduction

As languages become more complex, time spent on maintainability and experimentation becomes a considerable bottleneck in the process of designing new ones. For this reason, the TU Delft Programming Language research group developed the Spoofax Languages Workbench [1], a platform meant to aid the development of programming languages. It provides means for specifying languages, which serves as a base for generating parsers, interpreters, compilers, and other tools. The provided parser is based on the Scannerless Generalized LR (SGLR) parsing [2] algorithm. This algorithm has several advantages, such as removing the need for a scanner component and unifying the lexical syntax and context-free syntax. However, it has not yet achieved its full potential in terms of performance. In practice, better parsing performance creates a more seamless human-computer interaction and provides fast responsiveness necessary for common IDE tools such as auto-completion and syntax highlighting. In efforts of increasing the overall efficiency, JSGLR2 [3] is a re-implementation of the original algorithm (JSGLR), which takes a modular approach. The modular architecture facilitates code maintainability and allows for systematic benchmarking and optimizing of its comprising components. Moreover, optimization experiments encompass several variations of the main implementation that are maintained in parallel. For example, one such variant combines incremental parsing [4] with SGLR parsing and is called the incremental variant. Although JSGLR2 achieved considerable speedups by combining the improved components, it possibly introduced an overhead through its design. Currently, it is unknown to what extent the refactoring [5], through which the modular architecture was achieved, negatively impacts the performance of the system.

This paper aims to quantify the impact of the modular design on the parsing performance of JSGLR2, focusing on the overhead of the incremental variant. Because all variants are implemented in a modular fashion, they all suffer from the same architectural overhead, although possibly to different degrees. To examine the impact of the modular architecture, the refactoring that introduced these changes will be reverted. This process will be further called *inlining*. Consequently, a hypothesis that this work aims to confirm is that inlining reduces the overhead. Moreover, this study will examine how much the efficiency improves when particular components are inlined. Furthermore, which languages and inputs can benefit the most from inlining? What alternatives are there that maintain modularity but minimize the performance overhead? The main contribution of this paper is the inlined implementation of the incremental variant. The modular overhead is identified by comparing the performance of the two programs.

The paper has been organised in the following way. This paper first gives a brief overview of the SGLR algorithm background and Spoofax framework in Section 2. Section 3 is concerned with the methodology used for this study. Consequently, Section 4 will then go to describe the steps of the inlining process. In Section 5 the setup design and results will be discussed. Section 6 mentions reproducibility issues, while Section 7 contains further discussions on limitations and future work. Finally, a summary of the paper will be provided in Section 8.

# 2 Background

This section provides an overview of the main concepts behind SGLR and an outline of the frameworks that will be used in this research paper.

## 2.1 Parsing algorithms

**LR parsing** *LR(k) parsing* [6] is an algorithm that can deterministically parse a string in a given language. This type of parsing works on a subset of context-free grammars. These grammars have the property that they can be parsed by analyzing the input characters from left to right, by only looking ahead a finite number of characters, without looking at previous decisions, at any given time during parsing.

**GLR parsing** LR parsing algorithm fails to parse ambiguous grammars due to conflicts that are encountered in the parse table[1]. These have two origins: either the grammar could be ambiguous or the lookahead[2] needed is unbounded. These two causes cannot be distinguished in practice. This is because determining if a grammar is ambiguous is an undecidable problem [7]. Hence, it's not possible to determine if a conflict is due to ambiguity and therefore due to a lack of lookahead.
*Generalized-LR parsing* solves these issues by running parallel parsers whenever encountering a conflict in the parse table. If the conflict was caused by a lack of lookahead, only one of the parsers will successfully terminate, while the others will eventually fail. Otherwise, if ambiguity was the cause, multiple parsers will survive and a parse forest[3] will be constructed that will represent the possible parses of the input string.

[1] A language-specific table that indicates which action (shift/reduce) ought to be done in the certain scenario.
[2] *Lookahead* is the number of string characters a parser has to look ahead to be able to make a decision.
[3] A parse forest is a compact representation of a set of parse trees.

**SGLR parsing** *Scannerless parsing* aims to remove the separate lexical analysis preceding the parsing stage. During lexical analysis, the input is scanned and transformed into a stream of lexical tokens[4] which would next be provided to the parser. A reason for avoiding the use of a scanner is that disambiguation based on context already appears at the character level. As an example, take the string "matrix", which analysed can be considered as one identifier ("matrix") or a sequence of smaller identifiers ("m"; "a"; "trix"). As another example, take the string "ifx" which contains a reserved word ("if"). In this case, the first one is preferred, since multiple adjacent identifiers are not allowed. A solution to these is to let decisions at the lexical level to informed by context. To this end, the lexical and context-free syntax are combined into one context-free grammar. In doing so, the scanner and scanner-parser interface are removed. Furthermore, the parser is directly provided with a stream of characters instead of tokens.

*Scannerless Generalized-LR parsing* [2] combines the above idea with GLR. SGLR introduces follow restrictions and reject productions to solve disambiguation problems that would normally be tackled by the scanner (they solve the longest match and prefer literals disambiguation, respectively). Priorities and association attributes are further used for rule disambiguation. Moreover, a more compact parse table is achieved by using character classes[5].

**Incremental parsing** *Incremental parsing* [4] is a technique to increase time-efficiency when parsing large inputs. This is achieved by only parsing input differences, rather than the complete input as a batch when incremental changes are made.

### 2.2 Context and framework

**Spoofax language Workbench** [1] is a platform designed for the creation of new languages. It provides a meta-language (SDF3 [8]) in which a new language can be described. This language definition is used for generating a set of tools such as compilers, parsers, IDE plugins, etc. Spoofax also provides an evaluation tool that will be used in this work to benchmark the efficiency of various implementations.

**JSGLR2** [3] is a Java implementation of the SGLR algorithm used in Spoofax. It includes several parsing variants that apply different strategies such as Elkhound optimization, parsing with recovery and incremental parsing. To provide support for performance optimizations and easy maintenance, JSGLR2 takes a modular approach. To achieve this, it introduces code refactoring to divide the algorithm into components for which performance can be measured and consequently improved. This type of code manipulation changes the structure and inner workings of the software without impacting the overall functionality and output. It is a practice that aims to improve the code design. However, it may introduce changes that may be costly on performance [5]. Therefore, the modular architecture achieved through code refactoring allows for time-efficient modification of its components, but it negatively impacts the overall efficiency of the algorithm. Currently, there is not enough information to identify the extent to which the architecture slows down the parsing performance.

## 3 Methodology

In order to evaluate if the parsing performance is affected by the modular architecture, the refactoring that introduced the modularity had to be reverted. If the modified code would exhibit enhanced performance then the modular overhead hypothesis would be true. Thus, a new version of the same algorithm was implemented, which will be further called the *inlined* version. The new implementation will undergo a series of code transformations that would negate some of the refactoring actions. Because refactoring only modifies the internal structure of the code and does not affect the expected behavior, the same will be true for removing the refactoring. Consequently, the two implementations will be functionally equivalent. Thus, by comparing the two, the overhead could be identified and measured. An important factor in this comparison is fairness, therefore to have a valid differentiation no further optimizations were added to the inlined code.

Throughout this paper, we will refer to inlining as the process of modifying the code to reduce modularity. That includes the following refactoring actions: method inlining, class inlining, removing dead code, collapse inheritance, etc. Choosing these actions was a matter of identifying the refactoring actions that introduce code structure, and reverting those. For example, method inlining is the opposite of method extraction. It is usually applied when there is too much indirection. For instance, when a method calls upon another for functionality. Thus, by inlining, we would effectively reduce the number of method calls. Similarly, with class inlining, we reduce the number of instances created and the method calls.

[4] A lexical token represents a sequence of characters that compose a language element.

[5] A character class efficiently represents a set of characters by either describing one character ('a') or a range of characters ('a-z').

The approach to inlining was to incrementally modify the existing codebase by systematically stripping away its modularity. By using the original implementation as a starting point, we ensure that the base algorithm is the same. More specifically, no subtle changes are added which would skew the results. Moreover, the reason for adding incremental changes, as opposed to modifying the algorithm in one go, is to aid in code validation. Besides, this strategy will allow measuring each modification individually by using the commit history of the inlined program. A detailed account of the aforementioned changes to the parsing algorithm will be explained in Section 4.

While working on the code rewriting, an obvious requirement was its correctness. For this purpose, the already provided integration tests were used. For these tests to work with the new inlined implementation, certain structural elements of the code had to be maintained. This includes certain types and class inheritances. That implies that to ensure the validity of the code, certain elements were not inlined. This decision was taken due to time constraints and only affects a small number of classes.

For comparing the two versions, we will use the already existing benchmark project from JSGLR2 together with the JSGLR2 evaluation suite [9] to evaluate their performance. The benchmarks were executed on a set of languages and various input sources. To assess the performance implications in a more realistic scenario, a set of languages used in practice were chosen for the final results. Furthermore, the comparison will regard both batch parsing and incremental parsing performance.

With some of the components removed in the inlining process, the benchmark project had to be modified to be compatible with the new implementation. These modifications consisted mainly of changing some variable types to allow both variants to be handled. This is achieved by, for example, using a common class parent rather than the implementation class. It is important to mention that these should not change the way the benchmarks are calculated and will not skew the results in favor of either. Further similar adjustments were made for incremental benchmarks. Finally, the evaluation suite was also modified to include the inlined incremental implementation.

# 4 Inlining the algorithm

The main contribution of this paper is the inlined re-implementation of the incremental variant of JSGLR2. This implementation improves over its predecessor by stripping away its modularity and reducing the performance overhead associated with it. This section will provide a detailed description of the modifications brought to the algorithm, benchmark project, and evaluation suite.

## 4.1 Inlining the incremental variant of JSGLR2

First, a new class, *InlinedIncrementalJSGLR2*, was created which would eventually hold the entire inlined algorithm. Initially, it contained a copy of *JSGLR2ImplementationWithCache*, the class implementing the incremental variant. Subsequently, *JSGLR2Variant* class, which handles all different variants, was modified to include the inlined one by adding a new option to *JSGLR2Variant.Preset* called *inlinedIncremental*. This option will return a new *InlinedIncrementalJSGLR2Variant* class that extends *JSGLR2Variant*. The sole purpose of this class is to override the *getJSGLR2* method, which is responsible for building the right implementation, to instead directly return the inlined class (*InlinedIncrementalJSGLR*).

After the above adjustments have been taken, a systematic rewriting of the algorithm was in order. The process of inlining the algorithm uses a combination of well-established refactoring actions [5] as well as modified ones to suit this particular goal. InlinedIncrementalJSGLR2 contains three main components: a parser, an imploder, and a tokenizer. The next step was to inline each class separately to allow measuring the overhead of each. They where further named *IncrementalParser2*, *IncrementalStrategoTermImploder2* and *IncrementalTreeShapedTokenizer2*.

The strategy applied to inlining the classes' contents and their subsequent sub-classes will be exemplified through the change process of *IncrementalParser2*. First, the original class was copied into the new one. The second step was to apply "change function declaration" [5, pp. 124] refactoring by removing parameters from the IncrementalParser2 constructor. The reason for this modification was that the parameters were known in this case. Therefore, the parameters from the constructor were removed and instantiated inside the class. The only parameter kept is the parse table which is needed as input to the algorithm. A high-level abstraction of this step is shown in Listing 1.

```
parser = new IncrementalParser2(value)          parser = new IncrementalParser2()
...                                          →  ...
IncrementalParser2(Type a) {                    IncrementalParser2(Type a) {
}                                                   Type a = value
                                                }
```

Listing 1: Remove parameter from function declaration refactoring.

Furthermore, the generics were replaced with the hard-coded classes. Following this, the "collapse hierarchy" [5, pp. 380] refactoring action was taken. Class inheritance was removed, leaving the parser only implementing the topmost parent, the *IParser<IncrementalParseForest>* interface. This was done by applying the "push down field" [5, pp. 361] and the "push down method" [5, pp. 358] code manipulations to bring the content from the parent into the parser. The main concept of this refactoring is exemplified in Listing 2.

As a next step, the factory pattern classes were removed, and instead, hard-coded component classes (*ParseState, Disambiguator*, etc.) and manager classes (*ReduceManager, StackManager*, etc.) were instantiated directly. Although there is no defined refactoring procedure for this modification, it represents the inverse of "replace constructor with factory function" [5, pp. 334]. Listing 3 shows this code alteration.

```
class AbstractParser {}
class IncrementalParser2 extends
    AbstractParser {}
        ↓
class IncrementalParser2 {}
```

Listing 2: Inheritance removal inheritance refactoring.

```
parseState = parseStateFactory.get()
        ↓
parseState = new IncrementalParseState2()
```

Listing 3: Replace factory function with constructor.

Afterward, each component used by the parser was inlined through a similar process as above. Certain functions, particularly short ones that had 2-3 applications were inlined [5, pp. 115], nullifying "extract function" [5, pp. 106] refactoring. An example is presented in Listing 4. Furthermore, variable inlining [5, pp. 123] was performed (Listing 5) to reduce the number of variable assignments.

```
.createInitialStackNode() {
   createStackNode()                createInitialStackNode() {
}                              →        [code_block]
                                    }
createStackNode() {
    [code_block]
}
```

Listing 4: Inline function refactoring.

```
stackNode = value
return stacknode
    ↓
return value
```

Listing 5: Inline variable refactoring.

"Remove middle man" [5, pp. 192], inverse of "hide delegate" [5, pp. 189] action was used to eliminate forwarding. Listing 6 gives a visual representation of this modification.

```
var = parseState.newParseNodeAreReusable()

newParseNodeAreReusable() {                  →   var = !parseState.multipleStates
   return !multipleStates
}
```

Listing 6: Remove middle man refactoring.

Finally, the component class was inlined [5, pp. 186] (operation opposite to "extract class" [5, pp. 182]), by moving its content into the parser class. Thus, effectively reducing class coupling which in turn reduced parameter passing and the number of method calls.

In the parser class, further method and variable inlining were performed. Moreover, "remove dead code" [5, pp. 237] was applied to eliminate unused lines of code to clean up the source. Redundant code (such as methods that return the same value) removal was also done.

In the above manner, the *IncrementalParser2* and similarly *IncrementalStrategoTermImploder2* and *IncrementalTree-ShapedTokenizer2* were inlined. Fig. 1 shows a a high level abstraction of the architecture changes.

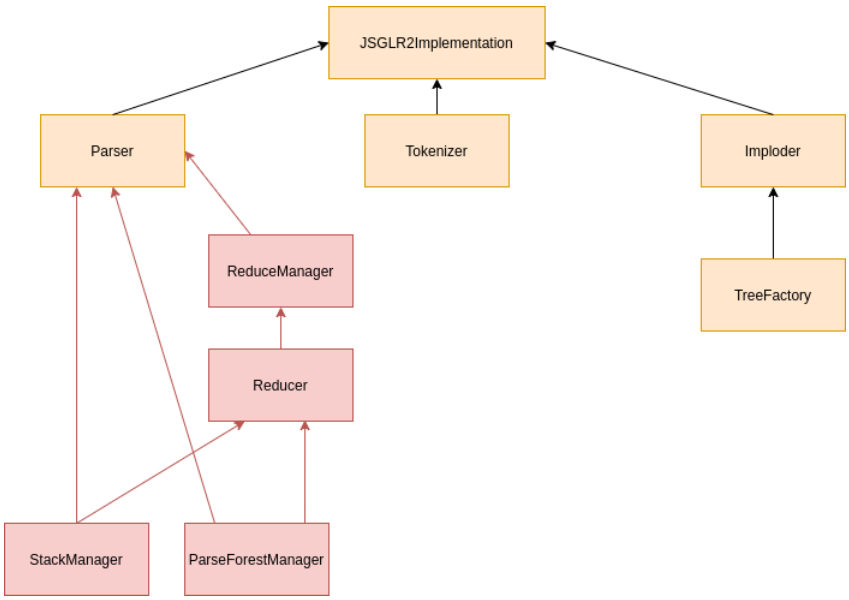



Figure 1: High level diagram of the architecture changes brought by inlining. In red are the removed classes, while in yellow are the remaining inlined classes.

As previously mentioned in Section 3, certain classes were kept in the inlined version such to have it work with the integration tests. This classes were *JSGLR2<IStrategoTerm>*, a common parent for JSGLR2 implementations, and *IParser* interface. As a consequence, a few other classes were partially inlined, meaning they still inherit from a common interface. This is due to the return type of the *parse* method in *IParser*. *ParseResult* class depends on the parse state and forces it to inherit from *AbstractParseState*, which in turn forces the input stack and stack node classes to implement their interface. Thus, IncermentalParseState2, HybridStackNode2, and others still implement an interface.

Finally, all inlined main components and sub-classes reside in the *inlinedIncremenetal* package. However, the InlinedIncrementalJSGLR2Variant and InlinedIncrementalJSGLR2 classes are found in the main package. A list of all modified classes excluding parent classes is available in Appendix A.

### 4.2 Updating the benchmarks

To benchmark the inlined incremental implementation alongside the modular one, the benchmark project was modified to incorporate both options. For this purpose, in the main benchmark class, *JSGLR2Benchmark*, the type of the variable responsible for holding the JSGLR2 implementation was modified to instead hold a common parent to the old implementation interface and the inlined implementation class. This parent is *JSGLR2<IStrategoTerm>* (class parent to both *JSGLR2Implementation* and *InlinedIncrementalJSGLR2*). For incremental parsing benchmarks, a new entry was added to *ParserType* in *JSGLR2BenchmarkIncremental* called *InlinedIncermental*. Further modifications were added to adjust the rest of the benchmark classes to the above modifications. They are available for inspection on a Git repository[6]. Finally, the inlined variant was added to the evaluation suite. The exact files modified to incorporate this are available in the evaluation repository[7].

[6]https://github.com/Marocco000/inlined-jsglr2.
[7]https://github.com/Marocco000/jsglr2evaluation.

## 5 Evaluation Setup and Results

This section includes information about the evaluation setup in Section 5.1, while Section 5.2 discusses results for both batch and incremental parsing scenarios.

### 5.1 Setup

Evaluating the performance of the inlined algorithm was done with the JSGLR2 evaluation suite [9]. The evaluation is performed on different languages, while the input sources for parsing consist of Git repositories. The behavior of the evaluation can be configured through several parameters in a YAML configuration file. They specify the variants under test, the languages used and associated git repositories used as input sources, etc. The modularity overhead hypothesis was tested by analyzing both batch and incremental parsing performances. The exact configurations for the above benchmark cases can be found in Appendix B. It can be observed that in both cases the *incremental* and *inlinedIncremental* variants are included for the comparison. For batch parsing, the relatively high number of warm-up iterations ensures that runtime optimizations are not taken into account in the benchmark calculation. Furthermore, the number of benchmark iterations provides a better impression of the performance by giving a confidence interval. Both values were chosen to reduce measurement errors and to consequently arrive at more consistent results. These parameters, however, have a lower value in the case of incremental parsing. Due to time constraints and the drastic increase in run time when executing incremental parsing, the above compromise has been made. The evaluation was performed on a set of three real languages (Java, WebDSL and SDF3) to better understand the implications of parsing performance in practice.

The benchmarks were run on a computer with an Intel Core i7 processor and 16 GB RAM, with a base frequency of 2.2 GHz. The machine operated on Ubuntu 20.04.2 LTS and used Java OpenJDK version 1.11.0. Moreover, during executions, all possible applications were closed. As mentioned in Section 3, the benchmarks had to be modified to be compatible with the inlined implementation. This, however, should not affect the consistency of the benchmarks when comparing them with other variants.

### 5.2 Results

First, the performance for batch parsing will be discussed. The exact benchmark values can be found in Appendix C. Fig. 2 showcases the comparison between the incremental and inlined incremental variants in terms of throughput[8] (left) and parse time (right). In both cases, the inlined version exhibits better performance than its modular counterpart. When considering character throughput, the difference can be clearly seen for all languages for both parsing time with (Fig. 2, c) and without (Fig. 2, a) imploding. On average, there is a 2%, 11%, 19% improvement[9] of throughput for Java, SDF3 and WebDSL respectively. Regarding parse time, again the inlined variant outperforms the non-inlined one. There is a 2% speedup[10] achieved for Java, 10% for SDF3, and 16% for WebDSL. The variation between performance differences between the languages could be attributed to the size of the input sources. For Java, the input files amount to a total of 83319 bytes, while 43245 bytes for SDF3, and 85496 bytes for WedDSL. A breakdown of the size distribution of the files per language can be examined in Appendix D. When looking at individual inlined component improvements, we found that the parser contributes to the majority of the overhead. This can be observed by comparing the graph bars in Fig. 2, b with the corresponding ones in Fig. 2, d. Parse time with imploding is similar to parse time without imploding. The difference between inlining the parser compared to inlining the imploder and tokenizer can be attributed to the increased complexity of the parser component compared to the other two.

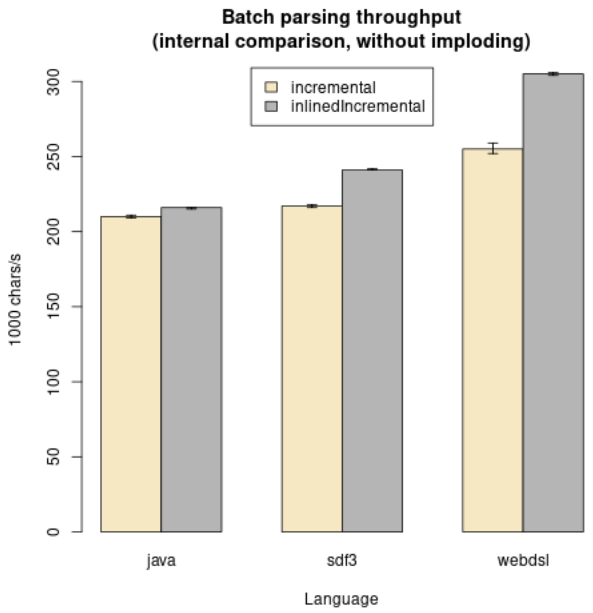
So far, the batch results are in line with the modularity overhead expectations. Parsing performance is increased for all three languages given the above metrics and evaluation corpus.

The next section of the evaluation was concerned with incremental parsing. In this case, the benchmarks are run on several successive versions for any source. The first one will be parsed in batch, while the consecutive ones incrementally (by only looking at the differences from the previous version). Unfortunately, some results are inconclusive because of the large error margins which overlap between the variant. This is possibly due to the small number of warm-up and benchmark iterations. While further extensive benchmarks have to be executed to fully grasp the real implications of parsing performance, this section will examine some of the consistent results. Because incremental benchmarks are calculated separately for each language and source, this section will analyze only a representative subset of them.

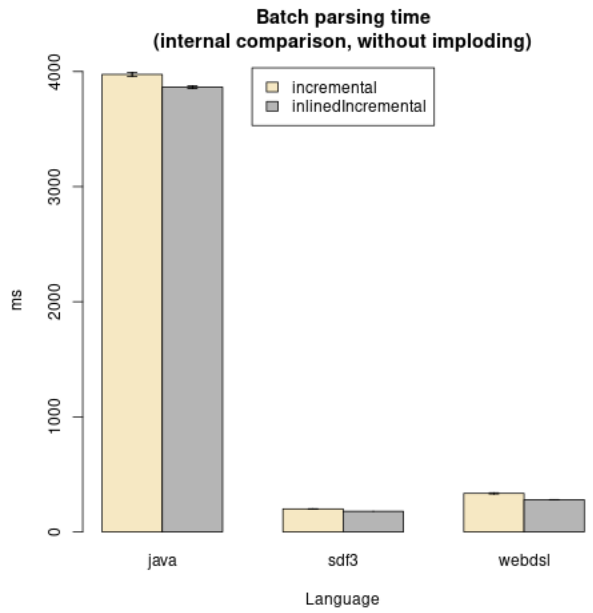
[8] Throughput is defined as the number of input characters parsed in a time unit.

[9] The throughput improvement is calculated with the following formula: $\frac{new-old}{old} * 100\%$. Where *old* refers to the modular version, and *new* is the inlined one.

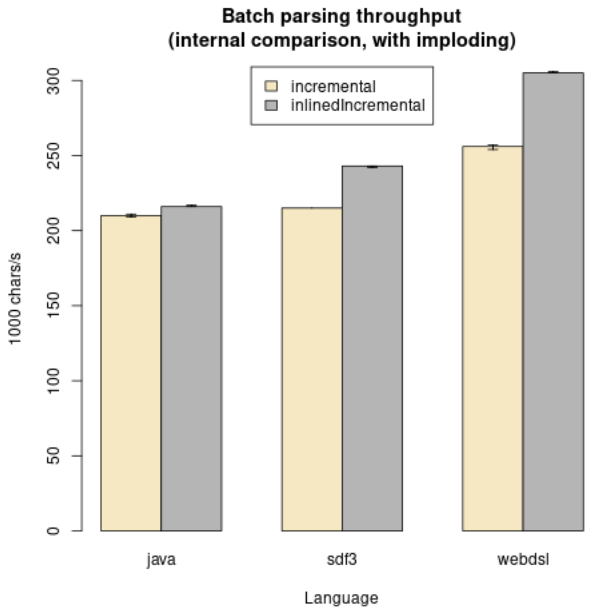
[10] The time speedup is calculated in the same way as for the throughput, but it's multiplied with -1.



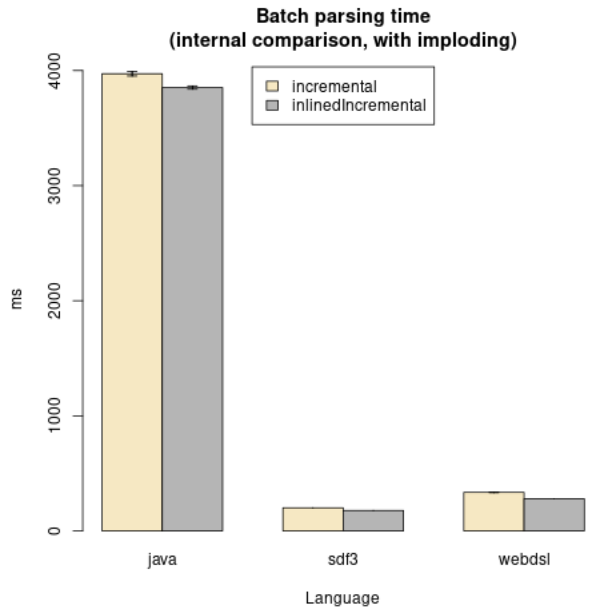

(a) Throughput comparison for batch parsing without imploding.




(b) Parse time comparison for batch parsing without imploding.




(c) Throughput comparison for batch parsing with imploding.




(d) Parse time comparison for batch parsing with imploding.

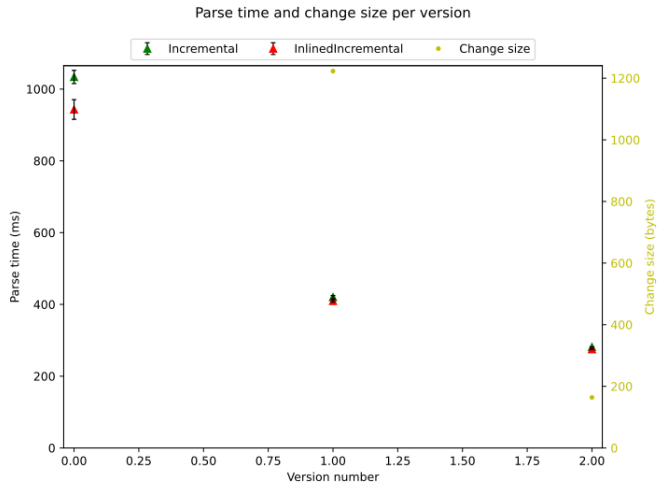
Figure 2: Batch parsing benchmark results.

The full collection of incremental benchmarks results per source can be inspected in Appendix E. The source sizes are available there as well.

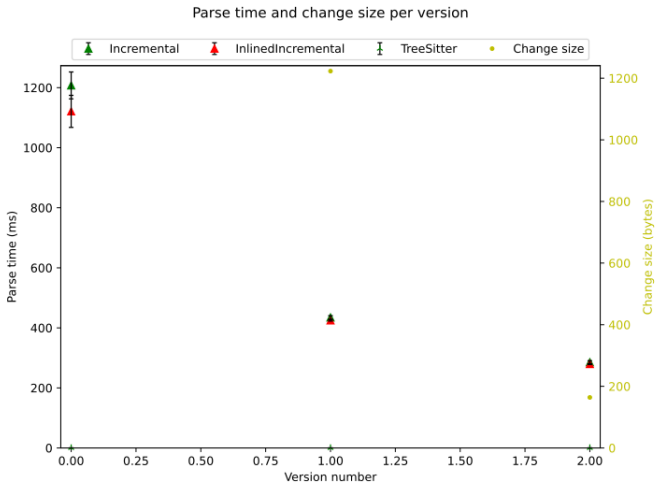
Fig. 3 present the parsing time in the incremental parsing scenario. The leftmost set of symbols (green for the modular variant, and red for the inlined one) represents the first parse time, which would be for the initial batch parse of the input. The following two sets of symbols indicate the incremental parsing time for two subsequent input modifications. For the first version, the inlined program takes less time than the modular one, thus the batch parse time values are in agreement with the previously discussed results. In Fig. 3, a, incremental parsing time results are showcased for parsing without imploding, while Fig. 3, b for parsing with imploding for the *apache.common.lang.stringutil* Java repository.

In both cases, the inlined variant is more efficient than the other one. For the first input modification, with 285 bytes altered, there is a 2% speedup. In the second phase of input editing, the change size is only 4 bytes, the same speedup is encountered.

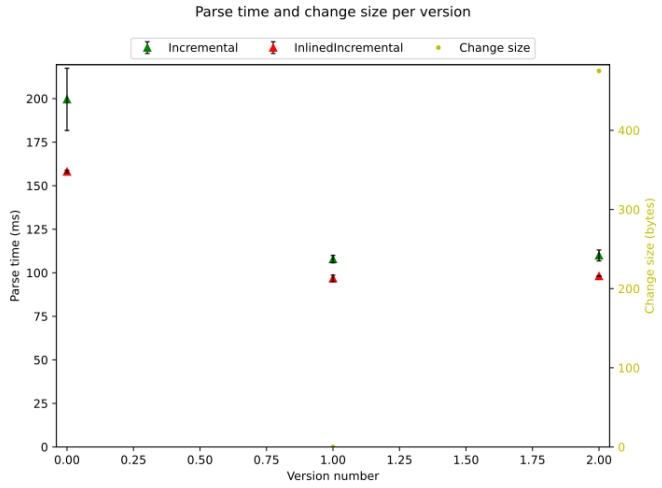
Meanwhile, the largest improvement can be observed in Fig. 3, c for an SDF3 repository. This graph shows that there is a 10% speedup in the two consecutive code versions. In Fig. 3, d a performance improvement of 6% is presented for parsing time without imploding of a WebDSL repository.



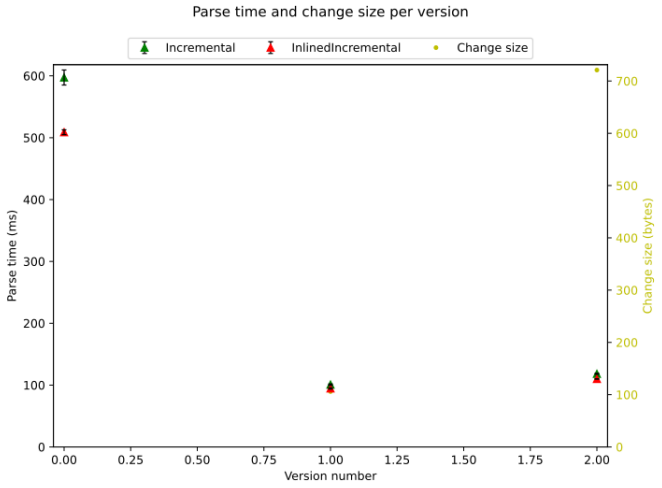

(a) Parsing time without imploding for the Java *apache.commons.lang.stringutils* repository.




(b) Parsing time with imploding for the Java *apache.commons.lang.stringutils* repository.



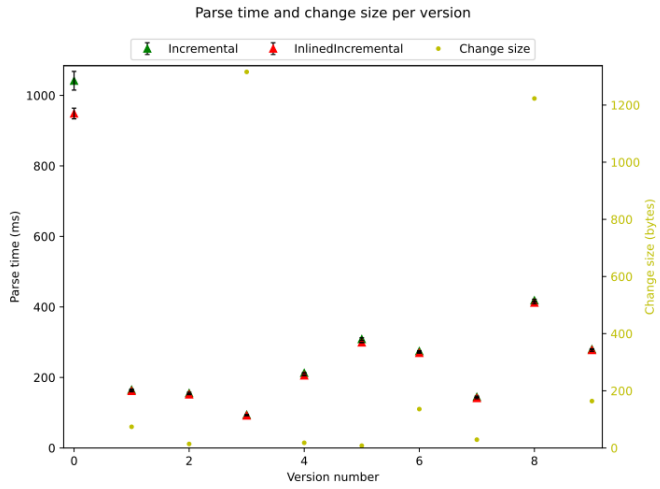

(c) Parse time without imploding for the SDF3 *Java* repository.



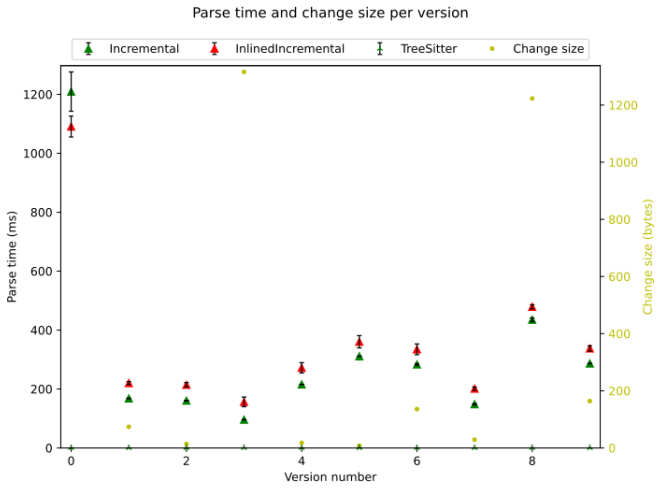

(d) Parse time without implding for the WebDSL *Yellowgrass* repository.

Figure 3: Incremental parsing benchmark results on 3 consecutive versions.

In an attempt to gather more data concerning incremental parsing, a larger benchmark was executed for the *apache.commons.lang.stringutils* Java source. This run consists of 10 consecutive versions with a maximum size change of 1316 bytes. The benchmark details are available in Appendix F. The results of this benchmark run are displayed in Fig. 4 In the case of parse time only (Fig. 4, a) the results are consistent with what has been observed so far. Interestingly, Fig. 4, b reveals otherwise. In this case, the inlined imploding components (imploder and tokenizer) slow down the variant. This is a rather unexpected result. More research is needed to explain this outcome. This case proves that the results should be interpreted with caution given the small evaluation suite. Such threats to validity will be discussed in Section 7. Since this only affects the imploder part, this case will be treated as an exception and will be ignored from the conclusions.




(a) Parsing time without imploding for the Java *apache.commons.lang.stringutils* repository.




(b) Parsing time with imploding for the Java *apache.commons.lang.stringutils* repository.

Figure 4: Incremental parsing benchmarks on 10 consecutive versions.

We have shown that in the above test conditions there is a performance improvement when the modular structure is removed from the code. Since the two implementations are functionally the same, the difference can only be attributed to the modularity itself. Thus, we have proven that the modular architecture negatively impacts the parsing efficiency. Moreover, we have shown that applying inlining reduces that overhead. On average, the inlined algorithm has achieved speedups between 2-16% in batch parsing time and between 2-10% in incremental parsing time compared to the old version. It is important to bear in mind that the results for parsing with imploding are questionable due to unexplained results.

## 6 Responsible Research

To allow the results to be validated by the scientific community, guaranteeing the reproducibility of the methods is a must. For that, the chosen approach is explained in a great amount of detail in Section 3. Two distinct components affect reproducibility: the modification of the algorithm and the evaluation method. Regarding the inlined implementation, the steps used to rewriting the algorithm are laid out in Section 4. In this way, the implementation can be replicated by an external party. Furthermore, the code has open access and will be hosted on a public Git repository [11] to facilitate further investigations. Another critical component that impacts reproducibility is the evaluation suite. In a similar manner as above, the modified evaluation suite is described and will be publicly available [12]. The selected tests used for the final results are detailed in Section 5 and can also be found in the repository. It is important to note that the results reached in this paper were possible in the context of the provided evaluation suite for which validation was not in the scope of this paper.

## 7 Discussion

**Limitations** These results show increased performance when removing the modular architecture from the original incremental variant of JSGLR2. This data must be interpreted with caution because it only represents an estimation of the overhead. The data can be regarded as a lower bound improvement since not all modularity was removed as explained in Section 4.1. Furthermore, while there is a comparison that indicates the largest overhead factor among the three main components (parse, imploder, and tokenizer), further research is needed to conclude which types of refactoring and code alterations bring the largest improvement. While benchmarking the individual commits in the inlining process might provide some insight into which steps added the highest performance boost, it cannot relate the efficiency increase to individual code refactoring actions in the current form. This is due to the high inter-dependency between the program components, which usually implies creating a chain of needed adjustments when one line is altered.

Another limitation of this work is posed by the incremental benchmarks. To better understand the implications of inlining in the incremental parsing context, a larger corpus of sources for each language is needed. This entails including subsequent Git repository versions and disabling *shrinkBatchSources*. Moreover, larger values for benchmark iterations and benchmark warm-ups are needed to obtain more precise results. During this project, these results were limited by time, since the execution time of an evaluation, as mentioned above, would take a week. Future work is required to investigate the impact of modularity in the case of incremental parsing.

**Threats to validity** Due to the above limitations and the generally restricted evaluation suite, two threats to validity arise. First, in terms of representativeness, the results discussed in this paper for the Java, SDF3, and WebDSL languages may not be entirely representative. A factor in this is the small benchmark test sizes. This is due in part because of the reduced number of repositories used for the sources and in part because of the small number of files from each source. The outcomes of the benchmarks might be different on a larger benchmark suite, as observed in the incremental parsing case. Therefore caution is warranted for concluding the impact on the respective languages. Another aspect is generality. Since the behavior of the modularity differs drastically for the above languages, it is difficult to generalize the effects on other languages and inputs.

**Related work** Regarding batch parsing, these results reflect those of Kapitonenko's [10] on the recovery variant, who also found that the architecture negatively impacts the performance. The improvement levels observed in this investigation are far below those observed by Kapitonenko [10]. The discrepancy could be attributed to the difference in the inlining approaches. While this work removes several classes, it fails to remove multiple inheritance relations. This approach is in contrast to the other paper, where removing inheritance took precedence over class removal. Comparing the performance improvement in the two papers also shows that these findings cannot be generalized to all variants.

**Future work** Since the ultimate goal is increased performance, future work could benefit from incorporating optimizations found by De Ruiter [11]. The improvements include data structure specializations and other optimizations. For further experimentation and improvement to the JSGLR2 variants, the modular architecture is highly beneficial. It allows quick adjustment and replacement of components and a high-grained view of their individual parsing performance. Refactoring "can certainly make software go more slowly—but it also makes the software more amenable to performance tuning" [5]. With the inlined program, future work would be impeded by the time needed to comprehend the code, resulting in less time spent on implementing adjustments. As a consequence, future optimization work

[11] Inlined algorithm is available at: https://github.com/Marocco000/inlined-jsglr2.

[12] Evaluation fork is available at: https://github.com/Marocco000/jsglr2evaluation.

would benefit the most by improving the modular implementation. Afterward, when a satisfactory performance has been achieved, the code can be further inlined for an extra efficiency boost. An alternative that could remove the modularity from the main algorithm but allow experimenting with multiple variants could be achieved by considering feature-oriented programming [12]. This programming paradigm allows composing software from a set of features. In doing so, it removes the need for class inheritance and dependencies. The modularity is lifted at the composition level, while the generated variant can be *inlined*. Therefore, the flexibility and modularity are maintained, but it does not affect the performance of the exact implementation.

## 8 Conclusion

This paper aimed to determine the performance overhead introduced by the modular architecture of JSGLR2, a Java implementation of the SGLR parsing algorithm. This work focuses particularly on the negative impact in the incremental variant of JSGLR2, an implementation that combines incremental parsing with SGLR parsing. To tackle this problem, an inlined version of the algorithm was created, which was then compared to the variant in question. To inline the variant, the original codebase was modified through refactoring to reduce the modularity. Consequently, the two algorithms are still equivalent.

Regarding the evaluation method, the comparison was achieved by benchmarking the parsing time and character throughput of the two programs with the JSGLR2 evaluation suite. The assessment was conducted on a set of three real languages (Java, SDF3, and WebDSL) using Git repositories as input sources. Therefore, the evaluation corpus simulates performance implications in a practical scenario. The evaluation was performed for both batch and incremental parsing cases. The results on the above test suite show that the inlined implementation outperforms its modular counterpart in batch parsing, as well as in incremental parsing. In the batch parsing case, speedups of 2% were registered for Java, 10% for SDF3, and 16% for WebDSL. For incremental parsing, results showcase up to 10% speedup. A note of caution is due here since this data might not be the same for larger test suites. This data provides proof that inlining reduces the overhead. Further investigation into the source of the overhead indicates that, out of the three main components (parser, imploder, tokenizer), the parser is responsible for the largest contribution in batch parsing.

While this paper proves inlining to be an effective method for increasing the performance of a given codebase, applying it is not recommended in the development cycle. For this purpose, a modular architecture will be more effective, by allowing efficient optimization identification and implementation. However, inlining parts of the code that attribute to most of the overhead, in this case, the parser, without compromising the modularity could prove beneficial for future development. As an alternative approach, feature-oriented programming could provide the needed modularity for experimenting with different parsers without sacrificing efficiency. Further research might explore the effectiveness of this alternative.

# A Modified classes

| Original class | New class |
|---|---|
| org.spoofax.jsglr2.JSGLR2ImplementationWithCache | InlinedIncrementalJSGLR2 |
| org.spoofax.jsglr2.incremental.IncrementalParser | IncrementalParser2 |
| org.spoofax.jsglr2.imploder.incremental.IncrementalStrategoTermImploder | IncrementalStrategoTermImploder2 |
| org.spoofax.jsglr2.stack.hybrid.HybridStackNode | HybridStackNode2 |
| org.spoofax.jsglr2.stack.collections.ActiveStacksArrayList | ActiveStacksArrayList2 |
| org.spoofax.jsglr2.stack.hybrid.EmptyStackPath2 | EmptyStackPath2 |
| org.spoofax.jsglr2.stack.collections.ForActorStacksArrayDeque | ForActorStacksArrayDeque2 |
| org.spoofax.jsglr2.stack.hybrid.HybridStackManager | -Removed- |
| org.spoofax.jsglr2.stack.hybrid.HybridStackNode | HybridStackNode2 |
| org.spoofax.jsglr2.imploder.incremental.IncrementalImplodeInput | IncrementalImplodeInput2 |
| IncrementalParseForestManager | -Removed- |
| org.spoofax.jsglr2.incremental.IncrementalParser | IncrementalParser2 |
| org.spoofax.jsglr2.incremental.IncrementalParseState | IncrementalParseState2 |
| org.spoofax.jsglr2.imploder.incremental.IncrementalStrategoTermImploder | IncrementalStrategoTermImploder2 |
| org.spoofax.jsglr2.stack.paths.NonEmptyStackPath | NonEmptyStackPath2 |
| org.spoofax.jsglr2.imploder.incremental.IncrementalTreeImploder.ResultCache | ResultCache2 |
| org.spoofax.jsglr2.stack.StackLink | StackLink2 |
| org.spoofax.jsglr2.stack.paths.StackPath | StackPath2 |
| org.spoofax.jsglr2.imploder.treefactory.StrategoTermTreeFactory | StrategoTermTreeFactory2 |
| org.spoofax.jsglr2.imploder.TreeImploder | TreeImploder2 |
| org.spoofax.jsglr2.reducing.ReducerOptimized | -Removed- |
| org.spoofax.jsglr2.parser.failure.DefaultParseFailureHandler | -Removed- |
| org.spoofax.jsglr2.incremental.diff.ProcessUpdates | -Removed- |
| org.spoofax.jsglr2.parser.observing.ParserObserving | -Removed- |
| org.spoofax.jsglr2.reducing.ReduceActionFilter | -Removed- |

Table 1: List of modified and removed classes in the inlining process.

## B Evaluation configuration files

```yaml
warmupIterations: 10
benchmarkIterations: 10
individualBatchSources: false
#implode: false
variants:
  - incremental
  - inlinedIncremental
shrinkBatchSources: 30
languages:
  - id: java
    name: Java
    extension: java
    parseTable:
      repo: https://github.com/metaborg/java-front.git
      subDir: lang.java
    sources:
      batch:
        - id: apache-commons-lang
          repo: https://github.com/apache/commons-lang.git
        - id: netty
          repo: https://github.com/netty/netty.git
  - id: webdsl
    name: WebDSL
    extension: app
    parseTable:
      repo: https://github.com/webdsl/webdsl-statix.git
      subDir: webdslstatix
    sources:
      batch:
        - id: webdsl-yellowgrass
          repo: https://github.com/webdsl/yellowgrass
  - id: sdf3
    name: SDF3
    extension: sdf3
    parseTable:
      repo: https://github.com/metaborg/sdf.git
      subDir: org.metaborg.meta.lang.template
    sources:
      batch:
        - id: nabl
          repo: https://github.com/metaborg/nabl
        - id: dynsem
          repo: https://github.com/metaborg/dynsem
        - id: flowspec
          repo: https://github.com/metaborg/flowspec
```

Listing 7: *config.yml* for batch benchmarks.

```yaml
shrinkBatchSources: 30
batchSamples: 3
warmupIterations: 5
benchmarkIterations: 5
jsglr2variants:
  - inlinedIncremental
  - incremental
variants:
  - incremental
  - inlinedIncremental
languages:
  - id: java
    name: Java
    extension: java
    parseTable:
```

```
      repo: https://github.com/metaborg/java-front.git
      subDir: lang.java
    sources:
      incremental:
        - id: apache-commons-lang-stringutils
          name: StringUtils
          repo: https://github.com/apache/commons-lang.git
          files:
            - src/main/java/org/apache/commons/lang3/StringUtils.java
          versions: 3 # 16
        - id: gson
          repo: https://github.com/google/gson.git
          versions: 3 # 16
        - id: slf4j
          repo: https://github.com/qos-ch/slf4j.git
          versions: 3 # 16
  - id: webdsl
    name: WebDSL
    extension: app
    parseTable:
      file: ../parsetables/WebDSL.tbl
      subDir: webdslstatix
    sources:
      incremental:
        - id: webdsl-yellowgrass-incremental
          name: YellowGrass
          repo: https://github.com/webdsl/yellowgrass
          fetchOptions:
            - '--depth=200'
          versions: 3 # 16
        - id: webdsl-elib-utils
          name: elib-utils
          repo: https://github.com/webdsl/elib-utils
          versions: 3 # 16
  - id: sdf3
    name: SDF3
    extension: sdf3
    parseTable:
      repo: https://github.com/metaborg/sdf.git
      subDir: org.metaborg.meta.lang.template
    sources:
      incremental:
        - id: nabl
          name: NaBL
          repo: https://github.com/metaborg/nabl
          versions: 3 # 16
        - id: dynsem
          name: DynSem
          repo: https://github.com/metaborg/dynsem
          versions: 3 # 16
        - id: flowspec
          name: FlowSpec
          repo: https://github.com/metaborg/flowspec
          versions: 3 # 16
        - id: webdsl
          name: WebDSL
          repo: https://github.com/webdsl/webdsl-statix.git
          versions: 3 # 16
        - id: java
          name: Java
          repo: https://github.com/metaborg/java-front.git
          versions: 3 # 16
```

Listing 8: *config.yml* for incremental benchmarks.

## C Parsing results for batch benchmarks

| | Java | | | | SDF3 | | | | WebDSL | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Variant | Score | Error | Low | High | Score | Error | Low | High | Score | Error | Low | High |
| Incremental | 3975 | 17 | 3958 | 3992 | 200 | 1 | 198 | 201 | 335 | 5 | 330 | 340 |
| Inlined | 3865 | 12 | 3853 | 3877 | 179 | 0 | 179 | 180 | 280 | 1 | 279 | 281 |

Table 2: Parse time results for batch parsing without imploding.

| | Java | | | SDF3 | | | WebDSL | | |
|---|---|---|---|---|---|---|---|---|---|
| Variant | Score | Low | High | Score | Low | High | Score | Low | High |
| Incremental | 210 | 209 | 211 | 217 | 216 | 218 | 255 | 252 | 259 |
| Inlined | 216 | 215 | 216 | 241 | 241 | 242 | 305 | 304 | 306 |

Table 3: Throughput results for batch parsing without imploding.

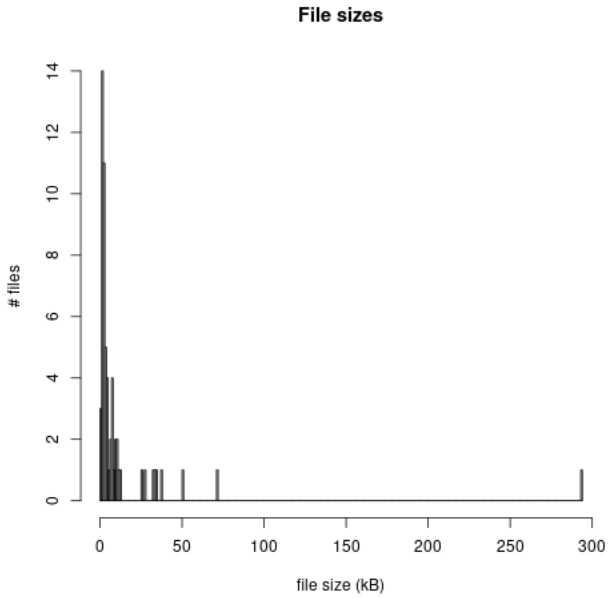
| | Java | | | | SDF3 | | | | WebDSL | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Variant | Score | Error | Low | High | Score | Error | Low | High | Score | Error | Low | High |
| Incremental | 3972 | 20 | 3951 | 3992 | 201 | 0 | 201 | 201 | 335 | 2 | 332 | 337 |
| Inlined | 3852 | 14 | 3838 | 3866 | 178 | 0 | 178 | 178 | 280 | 0 | 279 | 280 |

Table 4: Parse time results for batch parsing with imploding.

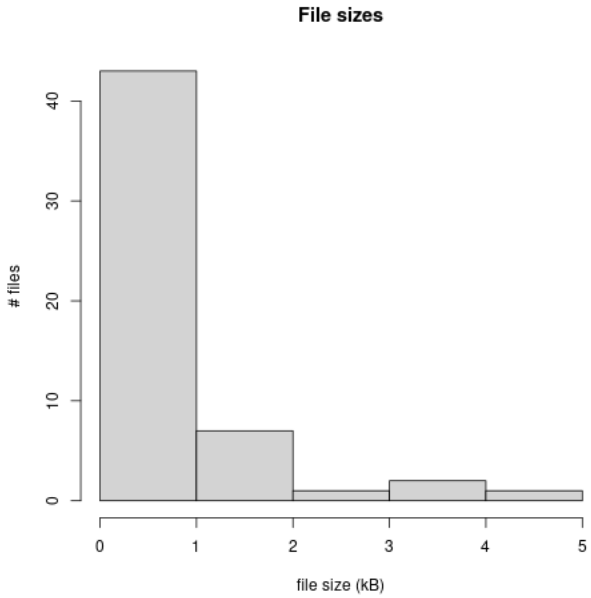
| | Java | | | SDF3 | | | WebDSL | | |
|---|---|---|---|---|---|---|---|---|---|
| Variant | Score | Low | High | Score | Low | High | Score | Low | High |
| Incremental | 210 | 209 | 211 | 215 | 215 | 215 | 256 | 254 | 257 |
| Inlined | 216 | 216 | 217 | 243 | 242 | 243 | 305 | 305 | 306 |

Table 5: Throughput results for batch parsing with imploding.

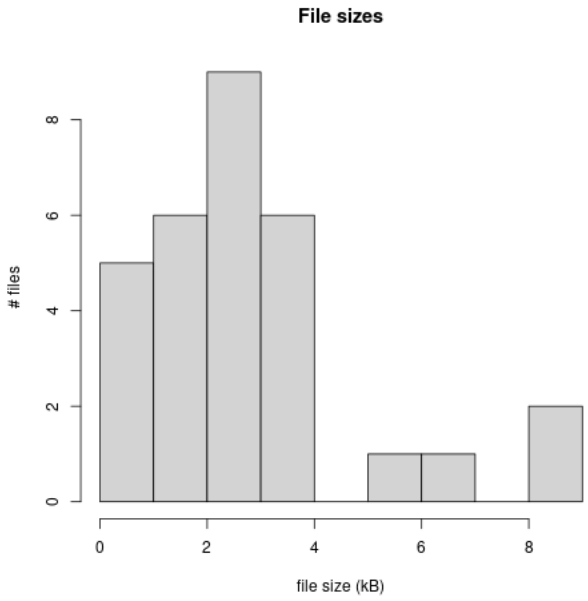
## D File size for batch benchmarks




(a) Java




(b) SDF3




(c) WebDSL

Figure 5: File size distribution per language for the batch benchmarks.

# E Parsing results for incremental benchmarks

The results are displayed on three consecutive source versions. The initial one is parsed as batch file, while the next two are parsed incrementally. Score represents the parse time results, while error entails the confidence interval of the result. The last four columns give an overview of the changes.

## E.1 Incremental benchmark results for Java sources

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 1033.809818 | 18.444811 | 943.149426 | 27.456445 | 397077 | 0 | 397077 | 1 |
| 1 | 420.282313 | 4.396608 | 409.541925 | 2.972963 | 396792 | 754 | 469 | 78 |
| 2 | 281.252805 | 0.571876 | 274.944223 | 0.917415 | 396796 | 80 | 84 | 7 |

Table 6: Parse time results for incremental parsing without imploding for the *apache.commons.lang.stringutils* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 1207.633681 | 44.828865 | 1121.123156 | 53.371931 | 397077 | 0 | 397077 | 1 |
| 1 | 434.621008 | 6.052436 | 424.774793 | 2.167861 | 396792 | 754 | 469 | 78 |
| 2 | 285.467009 | 5.085665 | 279.462036 | 1.63767 | 396796 | 80 | 84 | 7 |

Table 7: Parse time results for incremental parsing with imploding for the *apache.commons.lang.stringutils* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 6315.817796 | 33.414149 | 5737.880433 | 30.710028 | 1267762 | 0 | 1267740 | 205 |
| 1 | 671.292385 | 6.041778 | 636.393676 | 4.403128 | 1268412 | 788 | 1438 | 40 |
| 2 | 667.910698 | 2.050482 | 629.914798 | 1.115894 | 1268030 | 1135 | 753 | 25 |

Table 8: Parse time results for incremental parsing without imploding for the *Gson* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 7604.091235 | 79.48801 | 6990.964926 | 56.833465 | 1267762 | 0 | 1267740 | 205 |
| 1 | 711.237101 | 10.884736 | 671.58207 | 8.781639 | 1268412 | 788 | 1438 | 40 |
| 2 | 702.014724 | 2.020836 | 671.373367 | 12.398547 | 1268030 | 1135 | 753 | 25 |

Table 9: Parse time results for incremental parsing with imploding for the *Gson* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 3244.491091 | 13.953685 | 3026.688003 | 1.546083 | 881618 | 0 | 881618 | 239 |
| 1 | 513.622617 | 1.787211 | 473.233137 | 1.185892 | 883547 | 202 | 2131 | 5 |
| 2 | 512.084029 | 1.597435 | 471.635066 | 8.467117 | 883490 | 399 | 342 | 14 |

Table 10: Parse time results for incremental parsing without imploding for the *SLF4J* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 3880.372401 | 293.834349 | 3685.690771 | 140.316691 | 881618 | 0 | 881618 | 239 |
| 1 | 548.003901 | 2.643984 | 506.056588 | 2.2188 | 883547 | 202 | 2131 | 5 |
| 2 | 547.405188 | 6.707607 | 504.8967 | 2.487885 | 883490 | 399 | 342 | 14 |

Table 11: Parse time results for incremental parsing with imploding for the *SLF4J* git repository.

### E.2 Incremental benchmark results for SDF3 sources

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 49.760579 | 0.232469 | 41.733383 | 0.155687 | 10027 | 0 | 10027 | 4 |
| 1 | 20.226442 | 0.699069 | 17.57428 | 0.271149 | 10248 | 0 | 221 | 2 |
| 2 | 18.769945 | 0.80611 | 16.560819 | 0.479382 | 10248 | 2 | 2 | 2 |

Table 12: Parse time results for incremental parsing without imploding for the *DynSem* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 63.561911 | 0.285142 | 55.350358 | 0.5216 | 10027 | 0 | 10027 | 4 |
| 1 | 24.45504 | 1.128161 | 21.846444 | 0.632374 | 10248 | 0 | 221 | 2 |
| 2 | 22.610002 | 0.800692 | 20.262662 | 0.595706 | 10248 | 2 | 2 | 2 |

Table 13: Parse time results for incremental parsing with imploding for the *DynSem* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 74.179238 | 3.542244 | 62.143451 | 0.1754 | 13540 | 0 | 13540 | 20 |
| 1 | 24.197385 | 0.144094 | 22.496737 | 0.119463 | 13980 | 0 | 440 | 6 |
| 2 | 22.836035 | 0.091008 | 21.257564 | 0.161094 | 14154 | 14 | 188 | 5 |

Table 14: Parse time results for incremental parsing without imploding for the *flowspec* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 93.16225 | 6.99976 | 79.511199 | 1.057024 | 13540 | 0 | 13540 | 20 |
| 1 | 30.007047 | 0.339842 | 28.120143 | 0.374352 | 13980 | 0 | 440 | 6 |
| 2 | 28.32554 | 0.465432 | 26.459504 | 0.35741 | 14154 | 14 | 188 | 5 |

Table 15: Parse time results for incremental parsing with imploding for the *flowspec* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 199.57989 | 17.872918 | 158.095056 | 0.723302 | 45752 | 0 | 45752 | 65 |
| 1 | 107.859821 | 2.1245 | 96.780585 | 1.857393 | 45752 | 0 | 0 | 0 |
| 2 | 109.94456 | 3.150701 | 98.074098 | 0.16577 | 46227 | 0 | 475 | 1 |

Table 16: Parse time results for incremental parsing without imploding for the *Java* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 240.08968 | 22.021079 | 195.522932 | 6.080613 | 45752 | 0 | 45752 | 65 |
| 1 | 133.062811 | 3.268712 | 120.519673 | 1.207274 | 45752 | 0 | 0 | 0 |
| 2 | 134.590323 | 2.859786 | 122.511302 | 0.57671 | 46227 | 0 | 475 | 1 |

Table 17: Parse time results for incremental parsing with imploding for the *Java* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 491.115527 | 23.59855 | 390.089755 | 1.714005 | 100115 | 0 | 100115 | 127 |
| 1 | 248.233383 | 0.4939 | 223.217672 | 0.69141 | 100148 | 191 | 224 | 9 |
| 2 | 248.191583 | 0.791198 | 221.669483 | 0.791668 | 100115 | 224 | 191 | 9 |

Table 18: Parse time results for incremental parsing without imploding for the *nabl* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 615.117686 | 61.470319 | 495.080774 | 3.393949 | 100115 | 0 | 100115 | 127 |
| 1 | 302.191144 | 2.650512 | 274.266443 | 4.249153 | 100148 | 191 | 224 | 9 |
| 2 | 302.143027 | 2.173932 | 275.124839 | 1.894727 | 100115 | 224 | 191 | 9 |

Table 19: Parse time results for incremental parsing with imploding for the *nabl* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 353.768407 | 1.649842 | 327.14673 | 1.20991 | 86370 | 0 | 86370 | 26 |
| 1 | 132.226049 | 4.516577 | 124.262882 | 1.996462 | 86543 | 0 | 173 | 1 |
| 2 | 132.689236 | 6.128813 | 124.17377 | 2.602851 | 86538 | 35 | 30 | 4 |

Table 20: Parse time results for incremental parsing without imploding for the *WebDSL* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 449.714572 | 3.036198 | 414.201825 | 3.824561 | 86370 | 0 | 86370 | 26 |
| 1 | 159.0602 | 7.51142 | 149.866223 | 3.662193 | 86543 | 0 | 173 | 1 |
| 2 | 159.724543 | 6.42259 | 149.986069 | 5.260329 | 86538 | 35 | 30 | 4 |

Table 21: Parse time results for incremental parsing with imploding for the *WebDSL* git repository.

### E.3 Incremental benchmark results for WebDSL sources

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 151.810157 | 1.852306 | 132.331717 | 0.337403 | 39821 | 0 | 39821 | 17 |
| 1 | 26.380207 | 0.499133 | 24.589184 | 0.039509 | 40480 | 169 | 828 | 59 |
| 2 | 18.852763 | 0.046475 | 17.721104 | 0.157806 | 40589 | 22 | 131 | 12 |

Table 22: Parse time results for incremental parsing without imploding for the *elib-utils* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 185.880497 | 1.392006 | 164.17817 | 6.416399 | 39821 | 0 | 39821 | 17 |
| 1 | 29.848188 | 0.580249 | 27.442882 | 0.520305 | 40480 | 169 | 828 | 59 |
| 2 | 20.325259 | 0.137952 | 19.03698 | 0.097517 | 40589 | 22 | 131 | 12 |

Table 23: Parse time results for incremental parsing with imploding for the *elib-utils* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 597.320512 | 12.151184 | 508.846789 | 3.885016 | 159255 | 0 | 159255 | 52 |
| 1 | 100.804936 | 1.009478 | 94.64607 | 1.140683 | 159227 | 67 | 39 | 9 |
| 2 | 118.199089 | 0.825666 | 109.924011 | 0.126597 | 159948 | 0 | 721 | 45 |

Table 24: Parse time results for incremental parsing without imploding for the *yellowgrass* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 722.803985 | 4.700597 | 628.611049 | 5.700614 | 159255 | 0 | 159255 | 52 |
| 1 | 111.337703 | 0.86559 | 103.228556 | 1.741948 | 159227 | 67 | 39 | 9 |
| 2 | 131.855874 | 0.88965 | 122.386898 | 0.774526 | 159948 | 0 | 721 | 45 |

Table 25: Parse time results for incremental parsing with imploding for the *yellowgrass* git repository.

## F Extended incremental benchmark

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 1041.617352 | 26.515388 | 948.722174 | 15.175868 | 395896 | 0 | 395896 | 1 |
| 1 | 165.039814 | 1.872445 | 161.60821 | 2.305434 | 395842 | 64 | 10 | 9 |
| 2 | 155.547885 | 2.333521 | 151.772748 | 0.581446 | 395842 | 7 | 7 | 4 |
| 3 | 94.261832 | 0.154905 | 91.858521 | 0.244288 | 397158 | 0 | 1316 | 1 |
| 4 | 212.660421 | 0.950212 | 205.596192 | 0.329347 | 397176 | 0 | 18 | 3 |
| 5 | 308.772296 | 3.751749 | 299.202277 | 0.380217 | 397184 | 0 | 8 | 4 |
| 6 | 274.551614 | 1.369688 | 269.479605 | 0.690448 | 397048 | 136 | 0 | 11 |
| 7 | 144.950832 | 1.949517 | 141.421847 | 1.995579 | 397077 | 0 | 29 | 26 |
| 8 | 418.512521 | 3.306794 | 412.012262 | 2.954598 | 396792 | 754 | 469 | 78 |
| 9 | 280.034265 | 0.482315 | 277.686069 | 3.554332 | 396796 | 80 | 84 | 7 |

Table 26: Parse time results for incremental parsing without imploding for the *apache.commons.lang.stringutils* git repository.

| | Incremental | | Inlined | | Source change | | | |
|---|---|---|---|---|---|---|---|---|
| Version | Score | Error | Score | Error | Size (bytes) | Removed | Added | Changes |
| 0 | 1209.303984 | 66.720214 | 1090.736034 | 35.312431 | 395896 | 0 | 395896 | 1 |
| 1 | 168.047289 | 1.027854 | 220.091318 | 4.821185 | 395842 | 64 | 10 | 9 |
| 2 | 160.458725 | 0.966742 | 214.285399 | 7.320482 | 395842 | 7 | 7 | 4 |
| 3 | 95.864898 | 0.844084 | 156.309689 | 16.292616 | 397158 | 0 | 1316 | 1 |
| 4 | 215.584354 | 0.475625 | 271.973207 | 17.445074 | 397176 | 0 | 18 | 3 |
| 5 | 311.056554 | 0.548062 | 360.331869 | 20.797911 | 397184 | 0 | 8 | 4 |
| 6 | 283.265738 | 2.559716 | 334.584718 | 18.238101 | 397048 | 136 | 0 | 11 |
| 7 | 148.377498 | 1.976207 | 201.147121 | 5.477651 | 397077 | 0 | 29 | 26 |
| 8 | 435.207436 | 5.982853 | 478.961763 | 6.805869 | 396792 | 754 | 469 | 78 |
| 9 | 286.401615 | 0.362552 | 337.628681 | 8.813776 | 396796 | 80 | 84 | 7 |

Table 27: Parse time results for incremental parsing with imploding for the *apache.commons.lang.stringutils* git repository.

## References


[1] *Spoofax language workbench*. [Online]. Available: `http://www.metaborg.org/en/latest/index.html`, (accessed: 10.06.2021).

[2] E. Visser, "Scannerless generalized-LR parsing," Programming Research Group, University of Amsterdam, Tech. Rep. P9707, Jul. 1997.

[3] J. Denkers, "A modular sglr parsing architecture for systematic performance optimization," M.S. thesis, Delft University of Technology, 2018. [Online]. Available: `http://resolver.tudelft.nl/uuid:7d9f9bcc-117c-4617-860a-4e3e0bbc8988`.

[4] T. A. Wagner and S. L. Graham, "Efficient and flexible incremental parsing," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 5, pp. 980–1013, Sep. 1998, ISSN: 0164-0925. DOI: `10.1145/293677.293678`. [Online]. Available: `https://doi-org.tudelft.idm.oclc.org/10.1145/293677.293678`.

[5] K. B. M Fowler, *Refactoring: improving the design of existing code*. Boston: Addison-Wesley, 2019.

[6] D. E. Knuth, "On the translation of languages from left to right," *Information and Control*, vol. 8, no. 6, pp. 607–639, 1965, ISSN: 0019-9958. DOI: `https://doi.org/10.1016/S0019-9958(65)90426-2`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0019995865904262`.

[7] R. W. Floyd, "Syntactic analysis and operator precedence," *J. ACM*, vol. 10, no. 3, pp. 316–333, Jul. 1963, ISSN: 0004-5411. DOI: `10.1145/321172.321179`. [Online]. Available: `https://doi-org.tudelft.idm.oclc.org/10.1145/321172.321179`.

[8] *Spoofax language workbench*. [Online]. Available: `http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/`, (accessed: 10.06.2021).

[9] *Jsglr2 evaluation suite*. [Online]. Available: https://github.com/metaborg/jsglr2evaluation/tree/43b5b30e04cf5520acf2892175948002f8d1c2dd, (accessed: 10.06.2021).

[10] N. Kapitonenko, "Undoing software engineering: Demodularization of a sglr parser for performance gains," Bachelor's Thesis, Computer Science Bachelor Project, TU Delft, 2021.

[11] J. de Ruiter, "Optimizing sglr parser performance," Bachelor's Thesis, Computer Science Bachelor Project, TU Delft, 2021.

[12] C. Prehofer, "Feature-oriented programming: A fresh look at objects," *Lecture Notes in Computer Science*, vol. 1241, Oct. 1997. DOI: 10.1007/BFb0053389.