SPLASH: G: Incremental Scannerless Generalized LR Parsing

Maarten P. Sijm
Delft University of Technology
Delft, The Netherlands
mpsijm@acm.org

Abstract

We present the Incremental Scannerless Generalized LR (ISGLR) parsing algorithm, which combines the benefits of Incremental Generalized LR (IGLR) parsing and Scannerless Generalized LR (SGLR) parsing. The ISGLR parser can reuse parse trees from unchanged regions in the input and thus only needs to parse changed regions. We also present incremental techniques for imploding the parse tree to an Abstract Syntax Tree (AST) and syntax highlighting. Scannerless parsing relies heavily on non-determinism during parsing, negatively impacting the incrementality of ISGLR parsing. We evaluated the ISGLR parsing algorithm using file histories from Git, achieving a speedup of up to 25 times over non-incremental SGLR.

CCS Concepts • **Software and its engineering → Incremental compilers**; **Parsers**.

Keywords incremental, scannerless, parsing, GLR, IGLR, SGLR, ISGLR, imploding, syntax, Spoofax

ACM Reference Format:
Maarten P. Sijm. 2020. SPLASH: G: Incremental Scannerless Generalized LR Parsing. In *ACM Student Research Competition Grand Finals, 2020.* ACM, New York, NY, USA, 5 pages.

1 Introduction

Background Scannerless Generalized LR (SGLR) parsing combines the lexical and context-free phases of parsing. The terminals in the grammar are single characters instead of tokens. This has several advantages: it removes the need for a separate lexing (or scanning) phase, supports modelling the entire language syntax in one single grammar, and allows composition of grammars for different languages. One notable disadvantage is that the SGLR parsing algorithm of Visser [9] is a batch algorithm, meaning that it must process each entire file in one pass. This becomes a problem for software projects that have large files, as every small change requires the entire file to be parsed again.

Incremental Generalized LR (IGLR) parsing is an improvement over batch Generalized LR (GLR) parsing. Amongst others, Wagner [10] and TreeSitter [8] have created parsing algorithms that allow rapid parsing of changes to large files. However, these algorithms use a separate incremental lexical analysis phase which complicates the implementation of incremental parsing [11] and does not directly allow language composition.

Contributions In Section 2, we present the Incremental Scannerless Generalized LR (ISGLR) parsing algorithm that combines the benefits of incremental and scannerless GLR parsing. The algorithm only considers changed parts of the input and includes a test that prevents unchanged parse nodes from being reused incorrectly when a change in the context would require them to be parsed differently.

We explain the impact of non-determinism on the ISGLR parsing algorithm in Section 3. This effect explains that the combined algorithm cannot reuse as much from previous results as the non-scannerless IGLR parsing algorithm.

In Section 4, we discuss the integration of the ISGLR parsing algorithm in the Spoofax language workbench [3]. It is implemented as an extension to the Java implementation of SGLR (JSGLR2) [2]. Specifically, we focus on imploding to an Abstract Syntax Tree (AST) and on syntax highlighting.

We have evaluated the algorithm on input from the Git version control system, as shown in Section 5. Regardless of the non-determinism issue, the ISGLR parsing algorithm performs up to 25 times faster than batch parsing for small changes to large files.

2 Incremental Scannerless GLR Parsing

We present the ISGLR parsing algorithm, which combines the benefits of IGLR parsing and SGLR parsing. We will discuss the main ideas of our parsing algorithm below.

Input Preprocessing The input to the ISGLR parser consists of two parts: a list of changes between the previous and the current input strings, and the parse tree that resulted from the previous parse. The changes can be deletions or insertions, and having both at the same time represents a replacement. From the previous parse tree, the parser removes parse nodes that fall within a deleted region and creates a new (temporary) parse node for every inserted region, which contains the inserted characters as children.

Parse nodes store the width of their subtree to make this preprocessing step efficient. The width corresponds to the number of characters of the input represented by the subtree. With the exact position of a change, the subtrees requiring changes can be found using a traversal from the root node and recursively picking the child that contains this position.

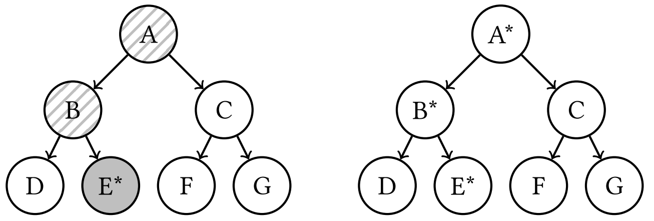
Any new parse nodes created in this process are marked as *irreusable*, signalling to the parser that they are not valid for reuse. This includes the parse nodes created because of insertions and replacements. In addition, all ancestors of any changed nodes are marked irreusable, simply because they have one or more descendants that cannot be reused, like the nodes A and B in the example of Figure 1a.

Parsing Instead of a stream of characters, the input to the parsing algorithm is a stream of *parse nodes*. These parse nodes can either be internal nodes (corresponding to grammar productions) or terminal nodes (corresponding to characters).

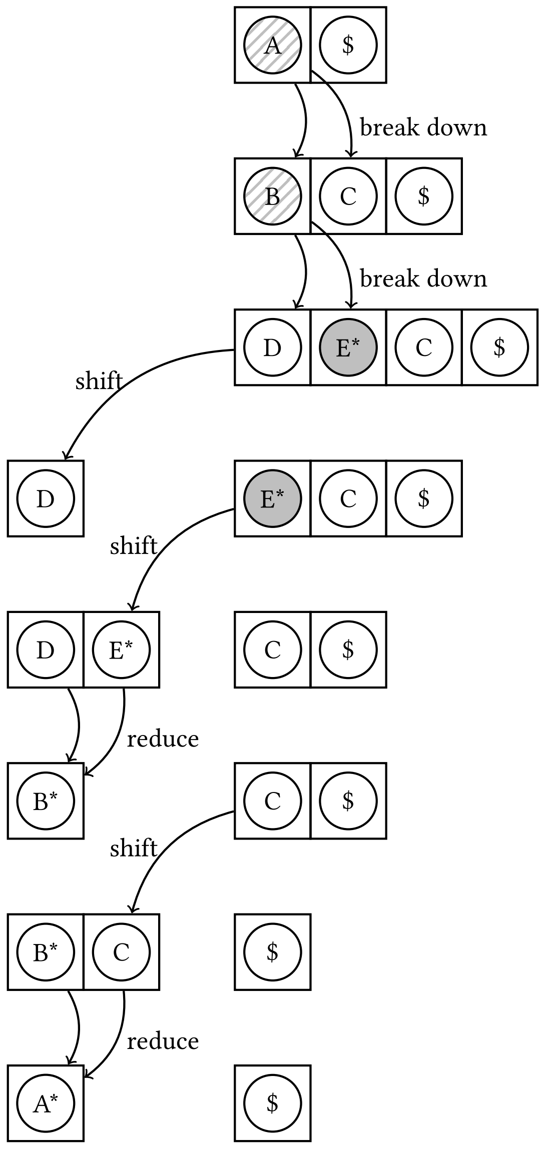
When parsing starts, the input stream consists only of the preprocessed parse tree and the end-of-file marker. When the parser encounters an irreusable parse node in the input stream, it is broken down, meaning that its child nodes will become part of the input stream instead. Ultimately, the parser will break down all parse nodes on the spines from the root to the changed regions. An example of this is shown in Figure 1b.

State Matching The state matching test is an extra check to decide whether an unchanged internal node from the input stream can be reused or not. This prevents the parser from blindly reusing a parse node that needs to be parsed differently because a part of the input has changed before the current position.

To accomplish this, the parser will store in each parse node the topmost state of the parse stack that it was pushed onto, so that it can be used for the state matching test during a subsequent incremental parse. During parsing, if the current state of the parser is equal to the state stored in the next node of the input stream, this node can be reused; else, it must be broken down like other irreusable parse nodes.



(a) Left: a preprocessed parse tree, where node E has been changed. Because of this, its ancestors A and B are irreusable.
Right: the resulting parse tree after reparsing.



(b) The parse stack is on the left and the input stream is on the right. During parsing, the invalid nodes A and B are broken down. Node D and subtree C can be fully reused.

Figure 1. An example of how the input stream and parse stack are behaving during incremental parsing.

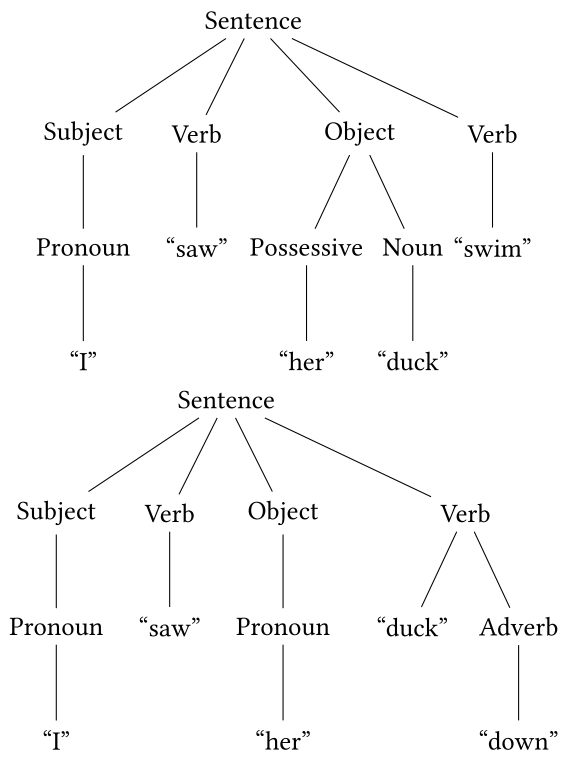


Figure 2. An example where non-determinism is used during parsing. In both sentences "I saw her duck swim" and "I saw her duck down", there is no ambiguity in the final result, but the word "her" has a different interpretation depending on the final word of the sentence. Therefore, the parser must explore both possibilities until it encounters the disambiguating final word.

3 Non-determinism

When a GLR parser reaches a point where multiple actions are possible, it will split the parse stack into multiple stacks and run the parsing algorithm concurrently on these stacks, synchronizing on shift actions [6]. Any stacks that have no applicable actions are discarded. As long as there are no ambiguities in the grammar, only one parse stack will remain.

Example To illustrate this, consider the example in Figure 2, showing two parse trees for two slightly different English sentences: "I saw her duck swim" and "I saw her duck down". Both sentences are not ambiguous in their meaning, but the words "her" and "duck" do have a different interpretation depending on the last word of the sentence. For the simplicity of this example, assume that these two sentences are the only possible valid sentences in English.

If these sentences were parsed using a GLR parser, the parser would split the parse stack when encountering the word "her". One parse stack would explore the possibility of "her" being a possessive pronoun, while the other would try to parse "her" as being a regular pronoun. This signals the start of a *non-deterministic* region in the input sentence: the parser can not directly know which interpretation is the correct one, so it explores all possibilities. Only when the parser reaches the final word of the sentence, the parser can discard the parse stack that has the incorrect interpretation and it continues with the single remaining parse stack, which ends the non-deterministic region in the input.

Impact on Incrementality Consider the previous example in an incremental setting, parsing one sentence after the other using an incremental parse. The changed word right after the non-deterministic region causes a different parse stack to survive. Even though the words "her" and "duck" have not changed, an incremental parser cannot blindly reuse the result of the previous parse.

As a result, any parse node that is created during non-deterministic parsing must be marked as *irreusable*, so that it will be broken down during the incremental parse even when it is unchanged. This forces the parser to explore all possible interpretations again. In the cases where, after reparsing, the parser chooses the same interpretation as before, it has effectively wasted some time reparsing that part.

Impact on Scannerless Parsing SGLR parsing relies heavily on the fact that the parser is non-deterministic because character-level grammars frequently need arbitrary length lookahead [9]. Unfortunately, this means that the number of parse nodes that ISGLR parsing can reuse is a lot less than for IGLR parsing. It is not yet clear how to reduce non-determinism in character-level grammars.

According to our measurements, about one-third of all parse nodes are marked as irreusable when parsing Java source code. However, on average only 2% of all parse nodes (of both kinds) were broken down during the experiment of Figures 4 and 5 in Section 5. The reason for this is as follows: while some irreusable nodes are exposed along the spine between the root and the changed regions, the majority of the irreusable parse nodes are not exposed and therefore the parser also does not need to break them down.

4 Editor Integration

Code editors can use the incremental result of an ISGLR parser to incrementally calculate the results of type checking and compilation, amongst others. In this research project, we focus on imploding the parse tree to an AST and on syntax highlighting.

We implemented the ISGLR parser in the Spoofax language workbench [3]. In Spoofax, language designers can specify programming languages declaratively by describing their syntax, static and dynamic semantics, and transformations. Spoofax then generates an editing environment for the language, including syntax highlighting and error checking. Our incremental parser adds to other recent work that incrementalizes the Spoofax pipeline, such as an incremental type checker [1], an incremental build system [4], and incremental compilation [7].

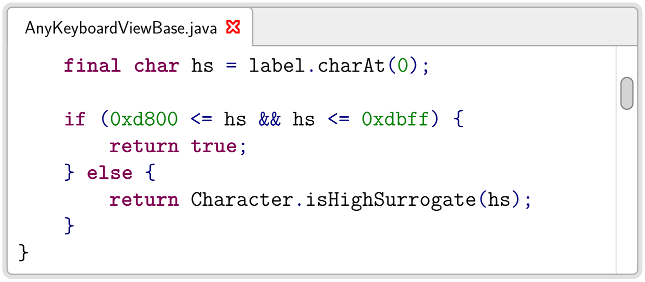
We implemented the ISGLR parsing algorithm as an extension to the JSGLR2 parsing algorithm [2]. The modular nature of JSGLR2 helped to only extend those parts of the parser that required changes to allow incremental parsing, resulting in about 1000 added lines of Java code.

Imploding The parse tree that the parser generates contains many details about the input program that are not relevant for further processing. Examples include whitespace and literal keywords (like `if` or `return`). The keywords are redundant information because they are always the same for their corresponding grammar rule and whitespace is redundant because it only contributes to the layout of the program, not the meaning of it.

In Spoofax, *imploding* is the post-processing step after parsing that removes this redundant information from the parse tree, producing an Abstract Syntax Tree (AST) that can be used in further processing. The baseline algorithm works in a top-down fashion: after processing a parse node, it processes the children of this node recursively.

In the design of the incremental imploding algorithm, we make use of two things. Firstly, we know that the parser only changed a small part of the parse nodes. All new parse nodes are reachable from the root of the parse tree via other changed parse nodes. Put differently, if a parse node is not changed, we can be certain that all its descendants are also not changed. Secondly, imploding is an operation that happens locally on parse nodes: no information of the parent node or any of the child nodes is required to process the current parse node. Because of this, we can store the resulting AST for each imploded parse node.

The incremental imploding algorithm also processes a parse tree recursively from the top down, with one key difference from the baseline algorithm: when encountering a parse node that already has a resulting AST, we can directly reuse this result. This ensures that only the parse nodes that are changed by the incremental parser are processed.



```
    final char hs = label.charAt(0);

    if (0xd800 <= hs && hs <= 0xdbff) {
        return true;
    } else {
        return Character.isHighSurrogate(hs);
    }
}
```

Figure 3. An editor showing syntax highlighting.

Syntax highlighting In most code editors, a program is displayed to the user using colours to give a visual indication about the different program elements in the code, as shown in Figure 3. This is also the case in Spoofax. Language designers can indicate which program elements get which colour and Spoofax will show the right colours in the editor.

Spoofax transforms the parse tree that resulted from the parser into a list of editor tokens, each given the correct colour based on the grammar rule that the parse node was created with. It might seem odd to create tokens when one of the features of scannerless parsing was that no tokens are required before parsing. However, there is one key difference with regular tokenization: for these tokens, information from the parser can be used to determine their type and colour. This means that the exact same word can be coloured differently based on context. Regular tokenizers like Lex only partially support this by allowing custom C code to be executed and having a mechanism called *start conditions* [5]. However, this can never be as powerful as a full context-free parser, simply because then a parser would be unnecessary.

For the incremental syntax highlighter, we store the tokens as leaves to the AST and link them together to allow iterating over the tokens in linear time. Updating tokens in updated parts of the AST is done in a way similar to imploding: subtrees that did not change can be directly reused and changed subtrees require reprocessing. In the case of reprocessing, the links between tokens on the boundaries of the change must be updated to make sure that iterating over all tokens uses the most recent tokens.

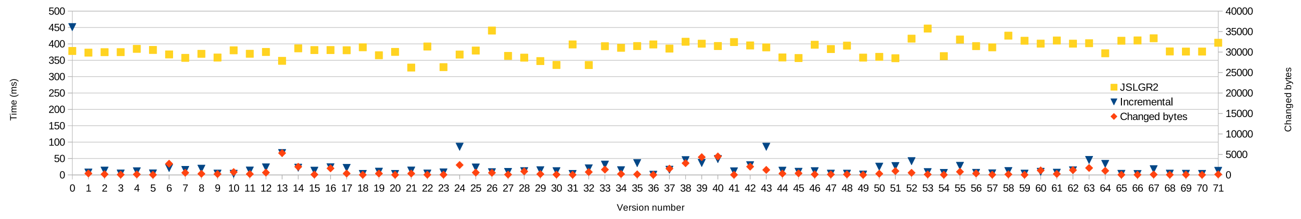


Figure 4. Parsing times of the JSGLR2 parser and the ISGLR parser on a Java file of almost 100 kB. The yellow squares and blue triangles indicate the parsing times (left y-axis) for these two parsers, respectively. The red diamonds indicate the number of changed bytes between each version (right y-axis).

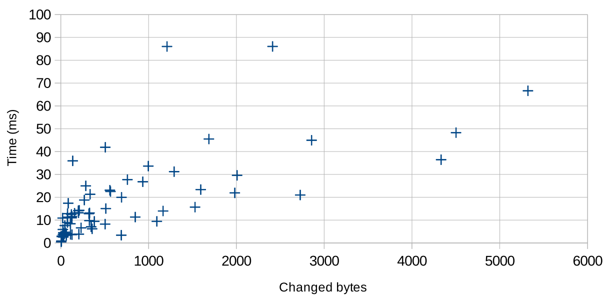


Figure 5. The same incremental parsing times as in Figure 4, showing the relation between change sizes and incremental parsing time.

5 Evaluation

We evaluated the runtime performance of the ISGLR parsing algorithm. As input to the parser, we use the file differences between commits in Git repositories. This experiment models how the parser would be used by a developer in their editor in the case that they would switch between commits in their local clone of the repository, for example, when they pull the latest changes committed by other developers. We are still working on experiments where user input is simulated as they would be working in the editor. The differences recorded in individual commits vary greatly in size and therefore cannot be directly used for this scenario.

It is important to distinguish between two types of results: those for *batch parsing* (where the full file is parsed from scratch) and for *incremental parsing* (where a new version of the file with a small change is parsed).

Preliminary results show that the ISGLR parser is on average 13% slower than the JSGLR2 parser when performing a batch parse. This slight slowdown can be attributed to the need to store more data to perform incremental updates later.

However, for incremental parses, the ISGLR parser has a speedup over JSGLR2 ranging from 15% faster (for parsing all versions of all files in a repository[1]) to 25 times faster (for a single file of 90 kilobytes that has changes averaging 700 bytes, with a standard deviation of 1100 bytes[2]). The results for the last experiment are shown in Figures 4 and 5. These figures show a slight correlation between the size of a change and the time needed by the ISGLR parser to perform an incremental parse on this change.

So far, our experiments have focused on evaluating just the parsing algorithm. In upcoming experiments, we will also evaluate the performance of the incremental imploding algorithm and incremental syntax highlighting.

[1]https : / / github . com / metaborg / mb - rep / tree / e33de52a766a1df6cbef79f069c3ebab822ef6e0

[2]https : / / github . com / AnySoftKeyboard / AnySoftKeyboard / blob / 16570810a492188687ad074679c74a9114291aa2/app/src/main/java/com/anysoftkeyboard/keyboards/views/AnyKeyboardViewBase.java

6 Conclusion

Our main contribution is the ISGLR algorithm, which combines SGLR parsing with IGLR parsing. An open challenge for this algorithm is that typically fewer parse nodes can be reused than with IGLR parsing due to the non-deterministic nature of scannerless parsing. However, in typical use cases, the ISGLR parsing algorithm will still perform better than the non-incremental variant.

Within this research project, we implemented incremental algorithms for imploding the parse tree to an AST and for syntax highlighting. We are still in the process of evaluating the performance of these editor integration services.

References

[1] Taico Aerts. 2019. *Incrementalizing Statix: A Modular and Incremental Approach for Type Checking and Name Binding using Scope Graphs.* Master's thesis. Delft University of Technology, Delft, The Netherlands. Advisor(s) Eelco Visser, Hendrik van Antwerpen, Neil Yorke-Smith, Robbert Krebbers. http://resolver.tudelft.nl/uuid:3e0ea516-3058-4b8c-bfb6-5e846c4bd982

[2] Jasper Denkers. 2018. *A Modular SGLR Parsing Architecture for Systematic Performance Optimization.* Master's thesis. Delft University of Technology, Delft, The Netherlands. Advisor(s) Eelco Visser, Michael Steindorfer, Eduardo de Souza Amorim. http://resolver.tudelft.nl/uuid:7d9f9bcc-117c-4617-860a-4e3e0bbc8988

[3] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10).* ACM, New York, NY, USA, 444–463. https://doi.org/10.1145/1869459.1869497

[4] Gabriël Konat, Michael J Steindorfer, Sebastian Erdweg, and Eelco Visser. 2018. PIE: A Domain-Specific Language for Interactive Software Development Pipelines. *Art, Science and Engineering of Programming* 2, 3 (2018), 1–31. https : / / doi . org / 10 . 22152 / programming - journal.org/2018/2/9

[5] M. E. Lesk. 1975. *Lex—A Lexical Analyzer generator.* Technical Report CS-39. AT&T Bell Laboratories, Murray Hill, N.J.

[6] Jan G Rekers. 1992. *Parser generation for interactive environments.* Ph.D. Dissertation. University of Amsterdam.

[7] Jeff Smits, Gabriël DP Konat, and Eelco Visser. 2020. Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System. *The Art, Science, and Engineering of Programming* 4, 3 (2020), 16–1.

[8] TreeSitter. 2019. *TreeSitter Documentation.* Retrieved May 23, 2019 from http://tree-sitter.github.io

[9] Eelco Visser et al. 1997. *Scannerless generalized-LR parsing.* Universiteit van Amsterdam. Programming Research Group.

[10] Tim A Wagner. 1997. *Practical algorithms for incremental software development environments.* Ph.D. Dissertation. University of California, Berkeley.

[11] Tim A. Wagner and Susan L. Graham. 1997. General Incremental Lexical Analysis. (1997).