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 parser preprocesses the input by modifying the previously saved parse forest. This allows the input to the parser to be a stream of parse nodes, instead of a stream of characters. 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, GLR, IGLR, SGLR, ISGLR, parsing, Spoofax

ACM Reference Format:
Maarten P. Sijm. 2019. Incremental Scannerless Generalized LR Parsing. In Proceedings of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion ’19), October 20–25, 2019, Athens, Greece. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3359061.3361085

1 Background
Visser introduced Scannerless Generalized LR (SGLR) parsing, which combines the lexical and context-free phases of Generalized LR (GLR) parsing. [5] The terminals in the grammar are single characters instead of tokens. This has several advantages: it removes the need for a separate lexical (or scanning) phase, supports modelling the entire language syntax in one single grammar, and composing grammars for different languages. One notable disadvantage is that the SGLR parsing algorithm is a batch algorithm: it processes each input file in its entirety. This becomes a problem for software projects that have large files, as every small change requires the entire file to be parsed again.

Wagner [6] and TreeSitter [4], amongst others, have introduced Incremental Generalized LR (IGLR) parsing algorithms that improve upon batch GLR parsing by incrementally parsing changes to large files. However, these algorithms use a separate incremental lexical analysis phase which complicates the implementation of incremental parsing and does not directly allow language composition.

2 Incremental Scannerless GLR Parsing
We present the Incremental Scannerless Generalized LR (ISGLR) parsing algorithm, which combines the benefits of IGLR parsing and SGLR parsing. We implemented the algorithm as part of the Spoofax language workbench [2] as a modular extension to the Java implementation of SGLR (JSGLR2). [1] We will discuss the main ideas of our parsing algorithm.

Input Preprocessing
After successfully parsing an input file, the parser saves both the input string and the resulting parse forest. When reparsing the same file, it calculates the difference between the previous and the new input strings. The changes can be deletions or insertions, or both at the same time. From the previous parse forest, 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. Changed parse nodes will no longer be valid, which will be fixed during parsing.

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 pre-processed parse forest and the end-of-file marker. When the parser encounters an invalid 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 1.
State Matching  A state matching test decides whether an unchanged internal node from the input stream can be reused or not. The parser will store in all parse nodes the top-most state of the parse stack that it was pushed onto. If the current state of the parser is equal to the state stored in the next node of the input stream, it can be reused; else, it must be broken down.

Non-determinism  When there are multiple possible actions, GLR parsers will split into multiple stacks and run the parsing algorithm concurrently on these stacks, synchronizing on shift actions. [3] 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. With a reparse, a change right after a non-deterministic region can cause a different parse stack to survive. As a result, any parse node that was created during non-deterministic parsing must be broken down.

SGLR parsing relies heavily on the fact that the parser is non-deterministic because character-level grammars frequently need arbitrary length lookahead. [5] Unfortunately, this means that the number of parse nodes that can be reused is a lot less than for IGLR parsing. It is not yet clear how to reduce non-determinism in character-level grammars.

3 Evaluation
We evaluated the ISGLR parsing algorithm with Git repositories, using the file differences between commits as input to the parser. Preliminary results show that the incremental parser is on average 13% slower than the JSGLR2 parser when parsing a file from scratch, but achieves a speed-up when parsing the files incrementally. The speedup of ISGLR over JSGLR ranges from 15% faster (for parsing all versions of all files in a repository1) to 25 times faster (for a single file of 90 kilobytes that has changes averaging 700 bytes2).

4 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. However, in typical use cases, the ISGLR parsing algorithm will still perform better than the non-incremental variant.

References
[1] 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:7d9f9bce-117c-4617-860a-4e3e0b8c8988

1 https://github.com/metaborg/mb-rep/tree/e33de32a766a1df6cebf79f6e93e3ebab22eef6e0
2 https://github.com/AnySoftKeyboard/AnySoftKeyboard/blob/165708103a492188687ad074679c74a914291aa2/app/src/main/java/com/anysoftkeyboard/keyboards/views/AnyKeyboardViewBase.java

Figure 1. An example of how the input stream and parse stack are behaving during incremental parsing.
[2] 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

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

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

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

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