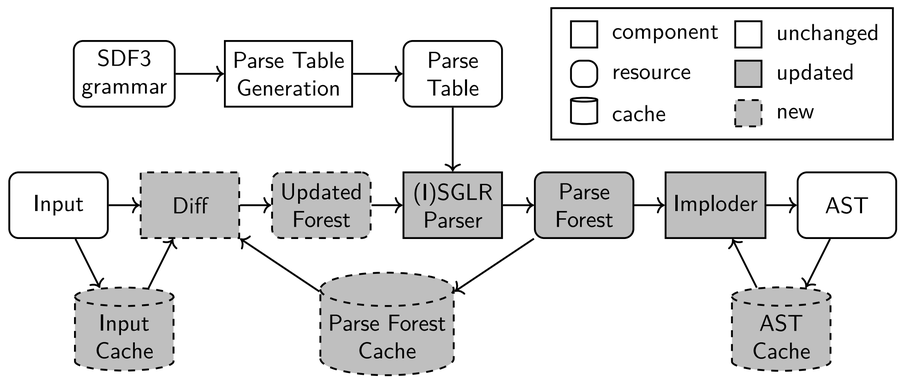
Incremental Scannerless Generalized LR Parsing

Maarten P. Sijm*

*Delft University of Technology, Programming Languages group mpsijm@acm.org

Introduction

We present the Incremental Scannerless Generalized LR (ISGLR) parsing algorithm, which combines the benefits of Incremental Generalized LR (IGLR) parsing [4] and Scannerless Generalized LR (SGLR) parsing [3]. 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 achieve a major speedup compared to JSGLR2 when parsing files incrementally.



The incremental parsing pipeline architecture.
Top row: *executed during language development.*
Middle row: *executed every time that a file is parsed.*
Bottom row: *caches that are maintained between parses.*

Processing Changes (*Diff*)

AnyKeyboardViewBase.java

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

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

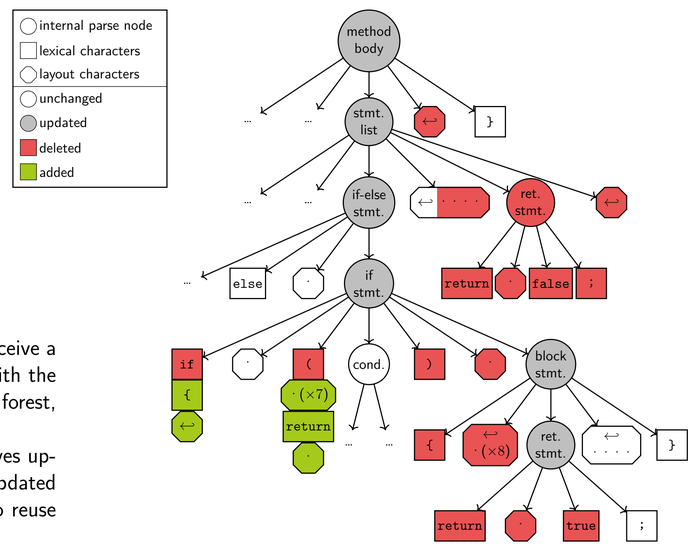
change

AnyKeyboardViewBase.java

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

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

```
····}·else·if·(Character.isHighSurrogate(hs))·{↩
········return true;↩
····}↩
····return false;↩
}↩
```

```
····}·else·{↩
·········return Character.isHighSurrogate(hs);↩
····}↩
}↩
```

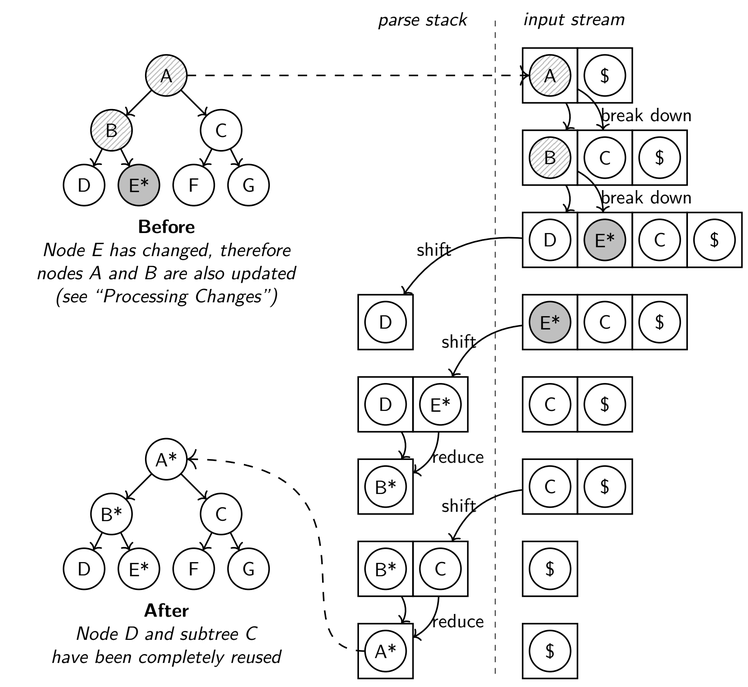


Upon a change by the user in the editor, the *Diff* component of the parser will receive a new version of the input file and computes a character-by-character difference with the previous version. These changes are then applied to the previously saved parse forest, producing the *Updated Forest*, as shown on the right.
Since parse nodes are immutable in our implementation, a parse node that receives updates to its children will be recreated (represented by the gray nodes in the updated forest). At parse time, the parser will break down any changed nodes and try to reuse unchanged nodes (see "Parsing Algorithm").

Parsing Algorithm

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 a changed or 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.



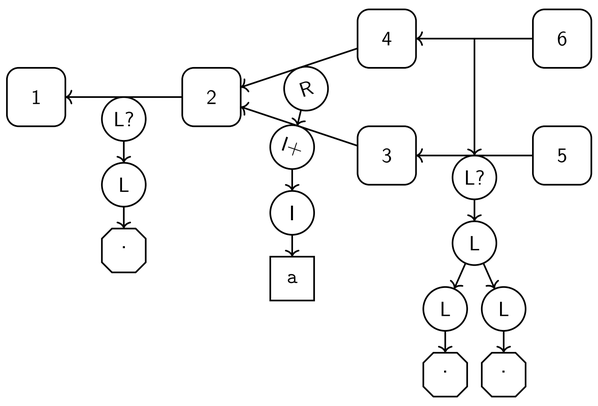
Non-determinism

The character-level grammars used for SGLR parsing frequently need arbitrary length lookahead [3]. Therefore, these grammars have a higher degree of non-determinism than token-level grammars. As an example, consider the grammar specification on the right. The graph-structured parse stack below shows a parser in states 5 and 6 after parsing the four characters "·a··". Now, two things can happen:

- If the parser encounters the end of the file, stack 6 reduces to the `Start` symbol.
- If the parser encounters an alphabetic character, it is shifted onto stack 5 and stack 6 is discarded.

In either case, this means that the created `Row` node can never be freely reused in a subsequent parse.

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.




```
context-free syntax

    Start = Row
    Row = Item+

lexical syntax

    Item = [a-z]
    LAYOUT = [\ ]
```

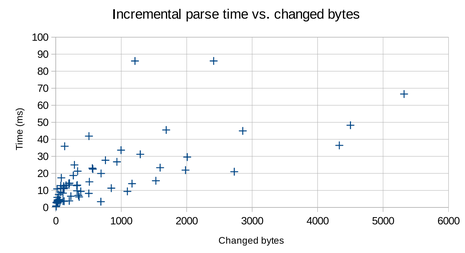
Above: *An example grammar written in SDF3.*

Below: *The same grammar as above, normalized.*

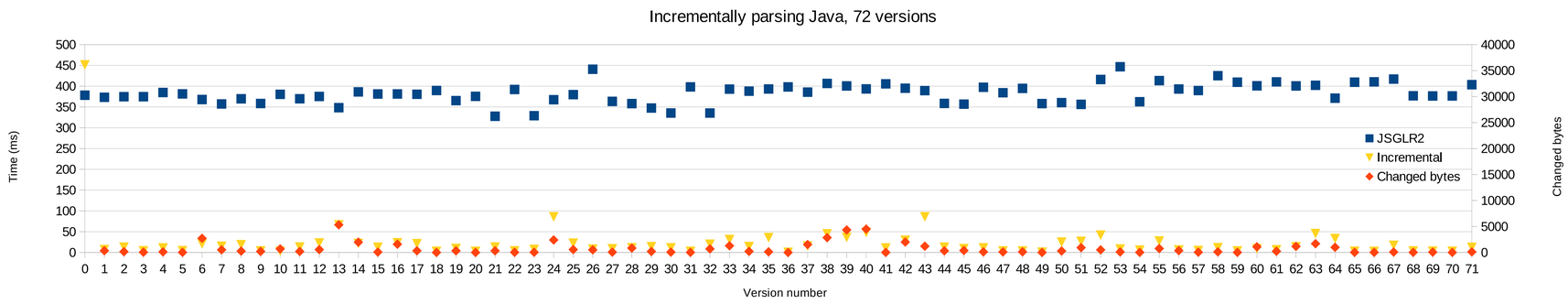
```
syntax

    Start = LAYOUT? Row LAYOUT?
    Row = Item+
    Item+ = Item+ LAYOUT? Item
    Item+ = Item
    Item = [a-z]

    LAYOUT? =
    LAYOUT? = LAYOUT
    LAYOUT = LAYOUT LAYOUT
    LAYOUT = [\ ]
```

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. ISGLR can be up to 25× faster than JSGLR2 when using files that are hundreds of kilobytes large.



The same incremental parsing times as in the plot below, showing the relation between change sizes and incremental parsing time.



Parsing times of the JSGLR2 parser and the ISGLR parser on a Java file of ± 100 kB.

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:7d9f9bcc-117c-4617-860a-4e3e0bbc8988

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

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

TU Delft

FAST — TU Delft | University Fund

Master thesis supervisors:
Prof. Dr. Eelco Visser
Jasper Denkers
Daniel A.A. Pelsmaeker

{S} spoofax