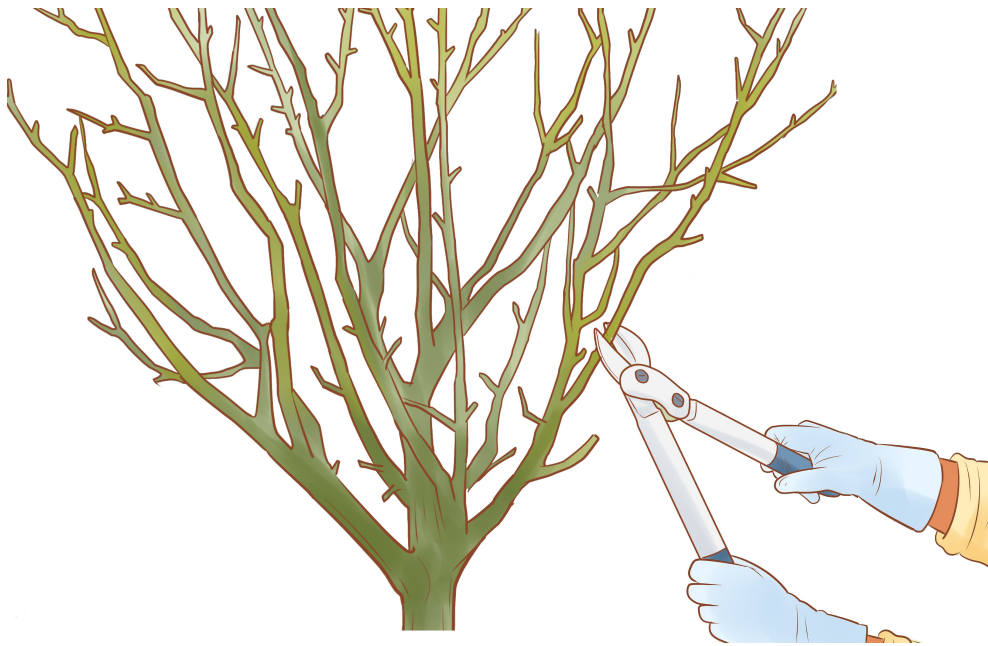


Incremental Scannerless Generalized LR Parsing

Master's Thesis



Maarten Peter Sijm

This page is intentionally left blank.

Incremental Scannerless Generalized LR Parsing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Maarten Peter Sijm
born in Honselersdijk, The Netherlands

To be defended publicly on
the 14th of July, 2021 at 10:00



Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.tudelft.nl/ewi
pl.ewi.tudelft.nl

© 2021 Maarten Peter Sijm, CC-BY 4.0.

Cover picture: Pruning a tree.

Source: <https://www.wikihow.com/Prune-a-Fruit-Tree>, CC BY-NC-SA 3.0.

All URLs have been archived at <https://web.archive.org/> on 2021-07-07.

Incremental Scannerless Generalized LR Parsing

Abstract

The Scannerless Generalized LR (SGLR) parsing algorithm supports the development of composed languages seamlessly but does not support incremental parsing. The Incremental Generalized LR (IGLR) parsing algorithm, on the other hand, does not support the seamless composition of languages. This thesis presents the Incremental Scannerless Generalized LR (ISGLR) parsing algorithm and investigates the effects of combining the SGLR and IGLR parsing algorithms. While the algorithmic differences are orthogonal, the fact that scannerless parsing relies on non-deterministic parsing for disambiguation has a negative impact on incrementality. Nonetheless, we show that the ISGLR parsing algorithm performs better than the batch SGLR parsing algorithm in typical scenarios. On average, the ISGLR parser can reuse 99% of a previous parse result. When parsing from scratch, the ISGLR parser has a 24% run time overhead compared to the SGLR parser, but when parsing incrementally for changes that are smaller than 1% of the input size on average, it has a 9× speedup.

Keywords

incremental, scannerless, parsing, GLR, IGLR, SGLR, ISGLR, imploding, syntax, Spoofox

Thesis committee

Chair:	Prof. Dr. E. Visser,	Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Bach Poulsen,	Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. T. van der Storm,	CWI / University of Groningen
Committee Member:	Prof. Emeritus P.D. Mosses,	Swansea University / TU Delft
University Supervisor:	J. Denkers MSc,	Faculty EEMCS, TU Delft

This page is intentionally left blank.

Preface

This thesis marks the end of my seven years of studying at the TU Delft, of which the final two-and-a-half years I should have worked on my thesis. There have been many distractions for me during this final period, but they have also helped me through it, in one way or another. The list of people I want to thank is long, but I will make an attempt to mention most of them.

Firstly, I would like to thank Eelco Visser and Jasper Denkers for supervising me throughout my thesis. Our many meetings have helped me reflect on and sort out ideas for my research. The hallway conversations with other members of the Programming Languages (PL) group have also played a big part in this. I am thankful that Eelco suggested presenting my work at the SPLASH 2019 conference in Athens (see Appendix B), where I got to meet other researchers who were also interested in parsing and liked to discuss ideas. Also, thanks to the PL group for letting me use one of their servers to run the evaluation scripts (for which Danny and Daniël were a great help) and for providing helpful feedback on my enhancements to Spoofox (see Appendix C).

One of the larger “distractions” was being a teaching assistant (TA) for the Bachelor courses of Computer Science and Engineering at the TU Delft. I would like to thank the many people I have worked with, both my colleague TAs and the teachers. The lab sessions and grading sessions provided some welcome social interactions.

I would also like to thank my friends and rubber ducks¹ for listening to my ramblings about parsing, providing suggestions for improvements and commenting on drafts of this thesis. My friends from the PL group, Bram, Chiel, Jeff, Luka, and Taico in particular, have provided some well-appreciated social distractions at the fourth floor and online. In addition, big thanks to Mathias and Mitchell, my buddies from the Westland with whom I have worked together since day one at the university, and hope to work with for many years to come at our startup company with Roy and Ruben.

I would be nowhere near where I am today without my supporting family, especially my parents and sister. Talking with them about working with parse trees has earned me the jocular title *softwarehovenier* (“software gardener”). And last, but definitely not least, my biggest thanks go out to my girlfriend Davina, who can make me calm even in the most stressful times.

Maarten Peter Sijm
Honselersdijk, the Netherlands
7th July 2021

Disclaimer: no real trees were harmed during the making of this thesis.

¹<https://rubberduckdebugging.com/> or https://en.wikipedia.org/wiki/Rubber_duck_debugging

This page is intentionally left blank.

Contents

Preface	iii
Contents	v
List of Figures	vii
List of Tables	ix
1 Introduction	1
2 Background	3
2.1 LR Parsing	3
2.2 Generalized LR Parsing	6
2.3 Incremental Generalized LR Parsing	8
2.4 Scannerless Generalized LR Parsing	10
3 Incremental Scannerless Generalized LR Parsing	17
3.1 ISGLR Parsing by Example	17
3.2 Non-determinism in ISGLR Parsing	24
3.3 Valid Parse Node Reuse	29
3.4 ISGLR Parsing Algorithm	31
3.5 Implementation in Modular Architecture	36
4 Incremental Post-Processing	39
4.1 Imploding	39
4.2 Tokenization	43
5 Performance Evaluation	49
5.1 Setup	49
5.2 Measurements Results	52
5.3 Time Benchmark Results	54
5.4 Memory Benchmark Results	57
5.5 Threats to Validity	60
6 Related Work	61
6.1 Incremental LR Parsers	61
6.2 Incremental Recursive Descent Parsers	63
6.3 Incremental Parser Combinators	64

7 Conclusion	65
7.1 Future Work	65
Bibliography	67
Acronyms	71
Glossary	73
A Full Evaluation Results	77
A.1 Measurements	77
A.2 Time Benchmarks	92
B SPLASH Conference 2019 – ACM Student Research Competition	103
B.1 Extended Abstract	104
B.2 Poster	107
B.3 Grand Finals	108
C Spoofox Enhancements	113

List of Figures

2.1	A small expression grammar, G_{Exp} , and its corresponding parse table.	4
2.2	A parse tree for the input “ $a + a$ ”, parsed according to the grammar G_{Exp} from Figure 2.1.	4
2.3	The process of parsing the input “ $a + a$ ” according to the context-free grammar G_{Exp} from Figure 2.1.	5
2.4	A part of the process of parsing “ $a + a \times a$ ” using G_{Exp} from Figure 2.1.	7
2.5	A screenshot of Harmonia, an IDE that uses the IGLR parsing algorithm (Wagner 1998, Figure 2.1, enhanced).	8
2.6	A snapshot of the state of an IGLR parser after a programmer has inserted the character 'b' in an existing identifier “acd”.	9
2.7	A small expression grammar, specified using SDF3 syntax.	11
2.8	The normalization of optionals and lists in SDF3.	12
2.9	An SDF3 example that uses lexical disambiguation constructs.	13
2.10	The current parsing pipeline in the Spoofox IDE.	13
3.1	The SDF3 definition of a small grammar.	18
3.2	The parse tree for input “ $ab = 42 * 42$ ” according to the grammar in Figure 3.1(b).	20
3.3	The diff between version 1 and version 2.	21
3.4	The parse stack (left) and the input stack (right) during an incremental parse.	22
3.5	The resulting parse tree after first parsing “ $ab = 42 * 42$ ” and then incrementally parsing “ $ans = 42 * 42$ ”.	23
3.6	The active parse stacks after parsing “ $a = x + y$ ”, according to the grammar of Figure 3.1.	25
3.7	The active parse stacks at two points during the parsing of “ $x = 3 y = 4 z = 5$ ”, according to the grammar of Figure 3.1.	27
3.8	Two production rules, added to the grammar of Figure 3.1, specified using SDF3 syntax.	28
3.9	The active parse stacks at two points during the parsing of “return ab”, according to the grammar of Figure 3.1, extended with the production rules of Figure 3.8.	28
3.10	A small expression grammar defined using kernel syntax.	30
3.11	The active parse stack and remaining input stack, after parsing “ $x*$ ”, according to the grammar of Figure 3.10.	30
3.12	The parsing pipeline in Spoofox, updated from Figure 2.10 for incremental parsing.	37
4.1	The imploded AST corresponding to the parse tree of Figure 3.2.	40
4.2	An example of the FLATTENLISTS function of Algorithm 12.	42
4.3	The tokens that are generated from the parse tree of Figure 3.2.	43
4.4	An example token tree, generated from the parse tree of Figure 3.2.	45

5.3	Parse times of the last 16 versions of several files and repositories from GitHub, excluding imploding and tokenization.	54
5.5	Parse times of the last 16 versions of several files and repositories from GitHub, including imploding and tokenization.	56
5.7	Size of memory allocations during parsing for three variants of the JSGLR2 parser.	58
5.9	Size of the cache after parsing a file for three variants of the JSGLR2 parser. . . .	59
C.1	A screenshot of Spoofox in Eclipse, showing a language that uses Unicode characters.	113
C.2	A screenshot of Spoofox in Eclipse, using a dark theme.	114

List of Tables

5.1	Corpus used to evaluate the performance of the ISGLR parser.	51
5.2	Measurements for parse nodes and breakdowns for the different languages in the evaluation corpus.	52
5.4	Average parse times for the different languages in the evaluation corpus, excluding imploding and tokenization.	54
5.6	Average parse times for the different languages in the evaluation corpus, including imploding and tokenization.	56
5.8	Average memory allocations for the different languages in the evaluation corpus.	57
5.10	Average memory size of the cache for the different languages in the evaluation corpus.	57

This page is intentionally left blank.

Chapter 1

Introduction

The development of a new programming language follows a predefined set of stages. The language needs a parser, static semantics, and a translation to some other language or machine code to run programs written in the language. In addition, an advanced editor for a programming language has features like incremental parsing, syntax highlighting, refactoring actions, and identifier lookup. Developing all these features for a new programming language takes a lot of effort, and this makes experimenting with language development cumbersome.

Using a tool that generates a parser from a declarative grammar specification simplifies the first stage of developing a programming language. These generated parsers can be integrated into an editor, which then supports writing programs in the new language. One such tool is Syntax Definition Formalism 3 (SDF3) (Vollebregt, Kats, and Visser 2012; de Souza Amorim and Visser 2020), which generates parsers that make use of the Scannerless Generalized LR (SGLR) parsing algorithm by Visser (1997). While traditional parsers require both a lexing and a parsing phase, SGLR parsing merges these two phases by using grammars that are defined in terms of single characters. Any disambiguation that the lexer normally performs is encoded in the SDF3 grammar and is consequently handled by the SGLR parser. The main advantage of this parsing technique is that it reduces the complexity of composing grammars for different languages into a single grammar. This simplifies the development of languages that have a different language embedded in them, such as the markup language HTML, which has CSS and JavaScript embedded into the language.

However, SDF3 and the SGLR parsers it generates do not support incremental parsing. Wagner and Graham (1997b) created an Incremental Generalized LR (IGLR) parsing algorithm that aims to reuse as much as possible from a previous parse result when reparsing a changed file. The time it requires to reparse a file containing a textual change is proportional to the size of this change. Just like SGLR parsing, the IGLR parsing algorithm is based on the Generalized LR (GLR) parsing algorithm by Tomita (1985) and Rekers (1992). We provide the background on all these parsing techniques in Chapter 2.

In this thesis, we apply Wagner’s incremental parsing technique to Visser’s SGLR parsing algorithm. The different disambiguation methods of SGLR might interfere with incremental parsing in how much of the parse result can be reused. Therefore, in this thesis, we answer the following research question:

What are the Effects of Combining Scannerless and Incremental GLR Parsing?

To answer this research question, we have investigated the differences of both SGLR parsing and IGLR parsing compared to GLR parsing. In terms of their algorithms, these differences are orthogonal to each other and can be combined without difficulties. We present the resulting Incremental Scannerless Generalized LR (ISGLR) parsing algorithm in Chapter 3. While the algorithmic differences are orthogonal, we show that there exist non-trivial interactions

between these two techniques: scannerless parsing introduces non-determinism in parsing that would not occur in non-scannerless parsing. This has a negative impact on the incrementality of the ISGLR parser, meaning that it can reuse less of the previous parse result than the non-scannerless IGLR parser.

To simplify the integration of the SGLR parser into an editor, its parse result is passed through two post-processing tasks: imploding removes irrelevant details from the parse result, and tokenization generates a list of tokens from the parse result. To our knowledge, these two tasks have not yet been described in the literature, so we provide their full description and algorithm in Chapter 4 before presenting an incremental algorithm for each post-processing task.

We have evaluated the performance of the ISGLR parsing algorithm using an automated benchmark suite that performs measurements and benchmarks on the ISGLR parser, based on a configuration file that describes the languages and sources used for the evaluation. In Chapter 5, we show that the ISGLR parser can reuse 99% of a previous parse result on average. We benchmark the run time performance using two scenarios: parsing files from scratch and parsing files incrementally. We show that ISGLR has a 24% run time overhead compared to SGLR when parsing from scratch, but when parsing incrementally for changes that are smaller than 1% of the input size on average, it has a 9× speedup.

We compare our approach with existing incremental parsing algorithms in Chapter 6. In Chapter 7, we conclude this work and state possibilities for future research.

Context We have developed the ISGLR parsing algorithm in the context of Spoofox (Kats and Visser 2010; MetaBorg 2016), a language workbench that simplifies the development of programming languages by generating many of the common features for a language based on declarative specifications. Besides generating an SGLR parser based on a grammar specification written in SDF3, it also generates a type checker based on a declarative specification of static semantics rules written in Statix (van Antwerpen et al. 2018), a compiler based on declarative transformation rules written in Stratego (Bravenboer et al. 2008), and an editor plugin that contains all of these features. Recent developments in Spoofox include incremental type checking (Aerts 2019) and incremental compilation (Smits, Konat, and Visser 2020), but it still lacks incremental parsing.

Contributions In summary, this thesis presents the following contributions.

- The Incremental Scannerless Generalized LR (ISGLR) parsing algorithm (Chapter 3).
- The negative impact of disambiguation in SGLR on incremental parsing (Chapter 3).
- A description and algorithm for imploding and tokenization (Chapter 4).
- Incremental algorithms for these two post-processing tasks (Chapter 4).
- An evaluation suite for measuring and benchmarking the performance of incremental parsers (Chapter 5).
- Insight into the performance of the ISGLR parser on representative inputs (Chapter 5).

Chapter 2

Background

This chapter provides background for the Incremental Scannerless Generalized LR (ISGLR) parsing algorithm. Section 2.1 explains the Left-to-right Rightmost-derivation (LR) parsing algorithm by Knuth (1965) and Section 2.2 explains the Generalized LR (GLR) parsing algorithm by Tomita (1985) and Rekers (1992). Based on GLR parsing, Section 2.3 explains the Incremental Generalized LR (IGLR) parsing algorithm by Wagner and Graham (1997b) and Section 2.4 explains the Scannerless Generalized LR (SGLR) parsing algorithm by Visser (1997).

2.1 LR Parsing

The LR(k) parsing algorithm by Knuth (1965) is a bottom-up approach for parsing *input streams* into *parse trees*. This algorithm parses an input stream according to a given language, where any language can be specified using *context-free grammars*. Regarding context-free grammars, we will use the following terminology, adapted from Sipser (2012):

A context-free grammar is a 4-tuple (N, Σ, R, S) , where

1. N is a finite set called the *non-terminals*;
2. Σ is a finite set, disjoint from N , called the *terminals*;
 - $N \cup \Sigma$ is called the set of *symbols*;
 - $\$$ is a special terminal called the End-of-File (EOF) symbol, used to indicate the end of the input;
3. R is a finite set of *production rules*, each of the form $A \rightarrow \alpha$ where A is a non-terminal and α is a string of symbols; and
4. $S \in N$ is the *start symbol*.

2.1.1 Lexical Analysis

The input to the parser is a stream of text characters. Since the terminals of an LR grammar typically correspond to several characters in the text, a *lexer* will analyse the stream of text characters and transform it into a stream of *tokens* that can be used as terminals. The lexer will skip any *layout* characters, as they are just used as separation between tokens.

Depending on the lexical analysis system, language designers can define tokens using regular expressions (W. L. Johnson et al. 1968) or using an entirely different formalism. As an example, when they use YACC (S. C. Johnson 1975) to describe an LR grammar, they will usually define the tokens using Lex (Lesk and Schmidt 1975).

2.1.2 Parse Table Generation

Before a context-free grammar can be used for LR parsing, a *parse table generator* generates a finite push-down automaton that will represent the LR parser. A two-dimensional *parse table* represents the possible state transitions for every *state* of the automaton, as shown in Figure 2.1. A parse table consists of two parts: an *action table* that maps state–terminal pairs to *actions* and a *goto table* that maps state–non-terminal pairs to other states.

In the action table, not all entries contain a valid action; trying to look up an empty entry will result in an error in the parser. In the bottom-left of the table in Figure 2.1(b), some entries contain multiple actions; these are called *conflicts* and will also result in an error.

$G_{\text{Exp}} =$	state	actions				gotos	
($\{S, E\}, \{a, +, \times\}, R, S$)		+	×	a	\$	S	E
The set of rules, R , is:	0			$S(2)$			1
$S \rightarrow E$	1	$S(3)$	$S(4)$		<i>Accept</i>		
$E \rightarrow E + E$	2	$R(E_a)$	$R(E_a)$		$R(E_a)$		
$E \rightarrow E \times E$	3			$S(2)$			5
$E \rightarrow a$	4			$S(2)$			6
	5	$S(3)/R(E_+)$	$S(4)/R(E_+)$		$R(E_+)$		
	6	$S(3)/R(E_\times)$	$S(4)/R(E_\times)$		$R(E_\times)$		

(a) The formal definition of G_{Exp} . (b) The parse table of G_{Exp} . Shift actions are abbreviated to 'S' and Reduce actions to 'R'. The actions will be explained in Section 2.1.3.

Figure 2.1: A small expression grammar, G_{Exp} , and its corresponding parse table. The parse table is generated using <http://jsmachines.sourceforge.net/machines/lr1.html>.

2.1.3 Parsing Algorithm

The two inputs to the LR parsing algorithm are a parse table and an input stream of tokens. The output of the parser is a parse tree, an example of which is shown in Figure 2.2. In this tree, the leaf nodes correspond to the terminals of the grammar and the non-leaf nodes, also called *parse nodes*, correspond to the production rules.

During parsing, the parser maintains a *parse stack*. The parse stack stores references to states in the parse table, always having the start state at the bottom of the stack. The current state of the parser is always at the top of the parse stack. Each link between two stack nodes stores a parse node or terminal.

The parser uses the current state and the first terminal from the input stream to get an action from the action table, as shown in Figure 2.3. The three different types of actions decide how the parser should continue:

Shift(s) Consume a terminal t from the input stream, push state s onto the parse stack, and store t on the created link to the previous stack node.

Reduce($A \rightarrow \alpha$) Pop $|\alpha|$ items from the parse stack and create parse node n corresponding to production rule $A \rightarrow \alpha$ with the popped nodes as children. Get state s' from the goto table using the state on top of the stack and non-terminal A , push state s' onto the parse stack, and store n on the created link to the previous stack node.

Accept Finish parsing and return the parse tree that is stored on the link between the only two remaining nodes on the parse stack.

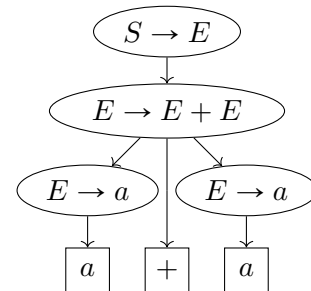


Figure 2.2: A parse tree for the input “ $a + a$ ”, parsed according to the grammar G_{Exp} from Figure 2.1.

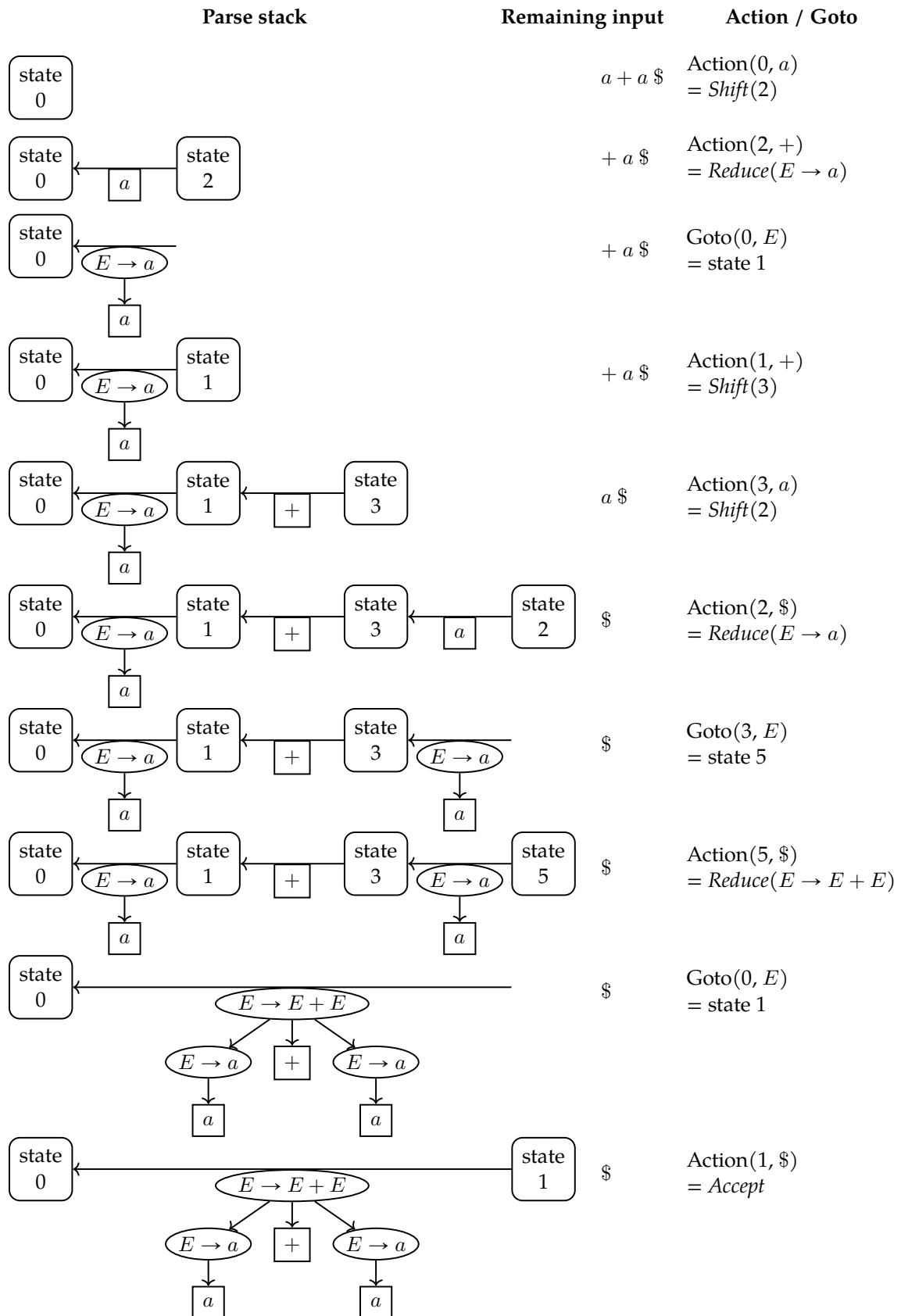


Figure 2.3: The process of parsing the input "a + a" according to the context-free grammar G_{Exp} from Figure 2.1. The EOF symbol $\$$ is appended to the input before parsing. In this figure, *Reduce* actions are processed in two steps: the first step shows popping items from the parse stack and the second step shows pushing a new state, determined via the goto table.

2.1.4 Bounded Versus Unbounded Lookahead

The k in “LR(k) parsing” stands for the number of terminals that the parser is allowed to look ahead when determining the next action from the parse table. The parser that we covered so far only looks at the first terminal in the input stream to determine the next action, so it is an LR(1) parser. In general, we will assume that a parser has a lookahead of one terminal when k is not specified.

All LR(k) parsers (with $k \geq 1$) are equally powerful because an LR(k) grammar can always be transformed into an LR(1) grammar (Mickunas, Lancaster, and Schneider 1976). However, LR(k) parsers cannot handle all context-free grammars. We say that a grammar requires a parser with *unbounded lookahead* if there exists no LR(k) parser that can parse all strings in the language of the grammar. For example, parsing C++ requires unbounded lookahead: the statement `int (a), b, c, d, ...;` can be interpreted either as a variable declaration list (the first being redundantly parenthesized) or as a comma-separated list of expressions (the first being a cast to `int`), depending on the final expression in the list. Since there can be arbitrarily many identifiers preceding this final expression, the parser theoretically needs infinite lookahead to disambiguate (Willink 2001, p. 147).

2.2 Generalized LR Parsing

Rekers (1992) created the GLR parsing algorithm based on the natural language parsing algorithm by Tomita (1985). Rekers’ algorithm overcomes the limitation of LR parsing by simulating unbounded lookahead, so it can handle all context-free grammars.

GLR parsing makes use of existing LR parse table generators. Whereas parse table entries with multiple actions generated an error for plain LR parsing, GLR parsing will instead explore all possible actions. The parser will immediately apply any possible *Reduce* actions, while it saves up *Shift* actions for the different stacks and only applies them once it has processed all *Reduce* actions. Effectively, this means that all active parse stacks are synchronized at the same location in the input stream.

Rekers uses a Graph-Structured Stack (GSS) to represent the parse stacks. The active parse stacks each have a different top in the GSS. All stacks have the same root, represented by the start state, just like the linear parse stack used in LR parsing. Some examples of GSSs are shown in Figure 2.4. Besides the regular pushing to and popping from the stack, three other things can happen during GLR parsing:

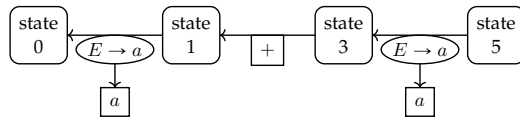
Forking The parser forks new parse stacks from an existing one whenever it can execute more than one action. This happens when it encounters a conflict in the parse table, as shown in Figure 2.4(a). The parser also forks the parse stacks when a *Reduce* action can be applied over multiple paths in the GSS, as shown in Figure 2.4(d).

Merging The parser merges parse stacks back together when actions from different parse stacks result in the same state. Examples of this are shown in Figures 2.4(b) and 2.4(e).

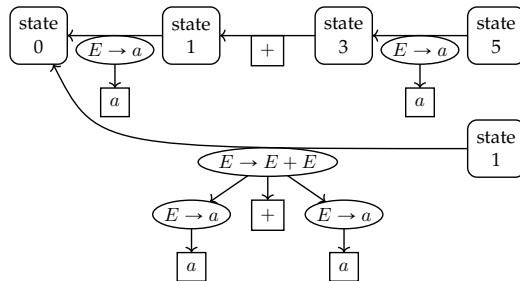
Discarding The parser discards a parse stack when it has no available actions. The parser halts and returns an error when all parse stacks have been discarded.

If the GLR parser needs to merge parse stacks and does not discard the resulting stack until it finished parsing, an *ambiguity* remains in the final result. This ambiguity is represented in a parse node by storing multiple *derivations* in it. When a parse tree contains (or may contain) ambiguities, it is referred to as *parse forest* since multiple non-ambiguous parse trees could be constructed from it.

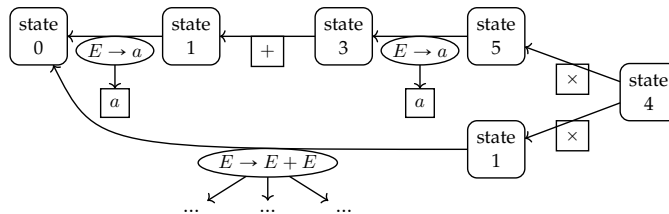
When a parser needs to fork the parse stack during parsing to achieve unbounded lookahead, we say that it is *parsing non-deterministically*. This does not imply that the final result contains ambiguities or that the grammar is *ambiguous* at all.



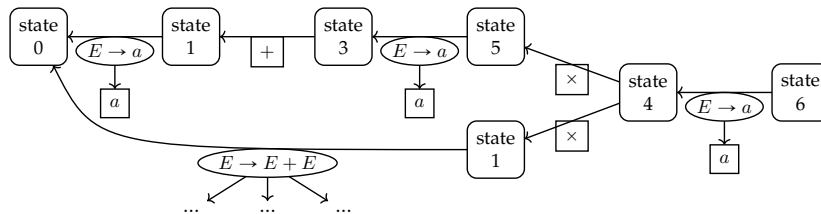
(a) The next terminal in the input stream is \times and $\text{Action}(5, \times) = \{\text{Shift}(4), \text{Reduce}(E \rightarrow E + E)\}$. The *Reduce* action is applied immediately using $\text{Goto}(0, E) = \text{state 1}$ and the parse stack forks. The *Shift*(4) action for the original stack will be applied after the *Reduce* action has been processed.



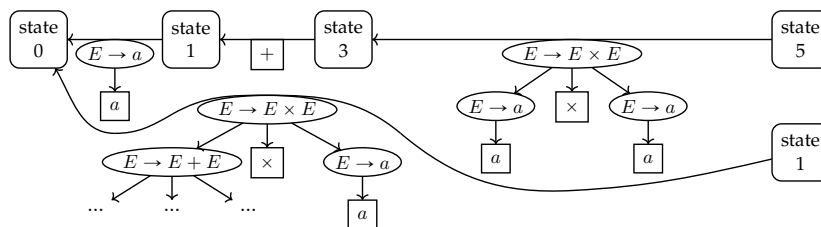
(b) For the new parse stack, $\text{Action}(1, \times) = \text{Shift}(4)$. The *Shift* actions for both stacks can now be applied. Since the new state is 4 in both cases, the stacks merge.



(c) After processing $\text{Action}(4, a) = \text{Shift}(2)$, $\text{Action}(2, \$) = \text{Reduce}(E \rightarrow a)$, and $\text{Goto}(4, E) = 6$, the parse stack looks as follows:



(d) With $\text{Action}(6, \$) = \text{Reduce}(E \rightarrow E \times E)$, there is only one possible action. However, there are two possible paths to reduce over: $3 - 5 - 4 - 6$ and $0 - 1 - 4 - 6$. Therefore, two new stacks are created using $\text{Goto}(3, E) = \text{state 5}$ and $\text{Goto}(0, E) = \text{state 1}$.



(e) The first stack has $\text{Action}(5, \$) = \text{Reduce}(E \rightarrow E + E)$ left to process and with $\text{Goto}(0, E) = \text{state 1}$, the two stacks are merged again. The parse node on the only remaining edge will then contain two derivations. At this point, $\text{Action}(1, \$) = \text{Accept}$ and the parser returns this parse node. The second derivation is not shown, as it is symmetric with the first derivation. Note that if the *Accept* action would be executed first, the other stack would still be processed, to get both derivations.

Figure 2.4: A part of the process of parsing “ $a + a \times a$ ” using G_{Exp} from Figure 2.1.

2.3 Incremental Generalized LR Parsing

Wagner (1998) introduced the IGLR parsing algorithm, which extends the GLR parsing algorithm by Tomita (1985) and Rekers (1992). Boshernitsan (2001) applied this parsing algorithm in the interactive setting of an Integrated Development Environment (IDE) called Harmonia (a screenshot is shown in Figure 2.5). When a software developer that uses Harmonia makes a small change to a file that they are working with, the incremental parser only performs work for the changed regions of the input. The reduced parsing time greatly speeds up the development process.

We will call subsequent invocations of the parser in interactive settings *incremental parses*. On the other hand, we use the term *batch parse* for a single invocation of the parser on a file without reusing previous results.

Just like LR and GLR parsing, the IGLR parser has two phases: a lexer and a context-free parser.

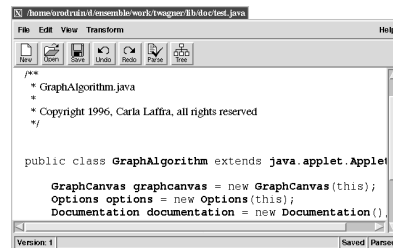


Figure 2.5: A screenshot of Harmonia, an IDE that uses the IGLR parsing algorithm (Wagner 1998, Figure 2.1, enhanced).

Incremental Lexical Analysis Wagner and Graham (1997a) describe an incremental lexer that maintains a mapping between the input character stream and the corresponding token stream. When the user changes the input, only the tokens nearby the changed regions need an updated lexical analysis to restore the mapping’s consistency.

The incremental lexer updates the tokens directly in the leaf nodes of the previous version of the parse tree. Therefore, the time to find which tokens need to be updated is proportional to the height of the parse tree for every changed region of the input. In this process, all parse nodes that are ancestors of the changed token nodes can be marked as changed.

Incremental Parsing After the incremental lexer has updated the tokens in the leaf nodes of the parse tree, the IGLR parser by Wagner and Graham (1997b) performs a traversal along this tree, starting at the root node. For every parse node that the parser encounters, it checks whether it is unchanged and valid for reuse. If this is the case, it shifts this node onto the parse stack, determining the new state from the goto table using the non-terminal symbol on the left-hand side of the production rule of the shifted parse node. The parser then performs one of the following two movements:¹

Descend If the parser *can not* reuse the current parse node, it moves to the first child node of the current parse node. In other words, the parser “breaks down” the current parse node to check its children.

Next If the parser *can* reuse the current parse node, it moves to the right sibling of the current parse node. If the current parse node is the last child in its parent, it moves to the right sibling of the closest ancestor that is not the last child in its parent.

Figure 2.6 shows an IGLR parser during such a traversal after a programmer has inserted the character ‘b’ in an existing identifier “acd” (Wagner 1998, Figure 6.1). It shows how the parser has already traversed the left part of the parse tree and has reused the parse nodes that have not changed, with the top of the parse stack (TOS) currently right before the changed token.

¹Wagner (1998) named these movements `leftBreakdown` and `popLookahead`, respectively.

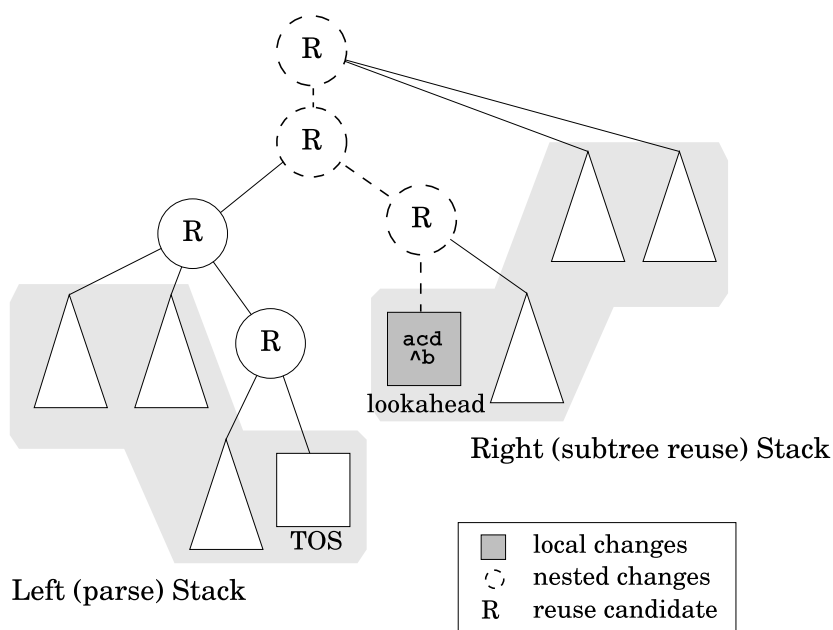


Figure 2.6: A snapshot of the state of an IGLR parser after a programmer has inserted the character 'b' in an existing identifier “acd” (Wagner 1998, Figure 6.1). The grey area on the left shows which parse nodes are on the parse stack right before the parser encounters the changed identifier, with “TOS” as the top of this stack. The grey area on the right shows which subtrees will be reused. This is not an explicit stack: the IGLR parser uses the *Next* movement to traverse over this part of the parse tree.

Valid Parse Node Reuse The IGLR parser can only reuse an unchanged parse node if the context surrounding this node allows this. To this end, it performs two tests, checking the context to the left and the context to the right. In other words, the parse node must be allowed to follow the part that has already been parsed, and the remaining input following this parse node must not have changed, respectively. We will give a more detailed example for both tests in Section 3.3 since the ISGLR parser uses the same tests.

Firstly, the IGLR parser uses a *state matching* test (Jalili and Gallier 1982) to check the context to the left, because the parse states capture this context by construction of the parse table. To implement this test, each parse node stores a reference to the state that was at the top of the parse stack *before* the node is pushed onto this stack. In a subsequent incremental parse, a parse node can only be reused if the state reference stored in the parse node is the same as the current parse state.

Secondly, the parser tests if the lookahead of the current parse node has changed, to check the context to the right. If this lookahead did change, the parser might need to take a different action from the parse table and cannot simply reuse this node. In a deterministic setting, only one² token following the current parse node needs to be checked. However, during non-deterministic parsing, the number of lookahead tokens is unbounded, as explained in Section 2.2. In a subsequent incremental parse, there could be changes following a non-deterministic region that would cause a different parse stack to survive. Therefore, parse nodes that are created while multiple parse stacks are active, are marked as *irreusable*. Wagner and Graham (1997b) implement this by not storing a reference to a parse state in them. In this way, the state matching test will always fail for this parse node, which causes the parser to *Descend* from the node to check its children.

²The parser only needs to check one token ahead in the case of an LR(1) parser. An LR(k) parser would have to check the following k tokens to see if they changed.

2.4 Scannerless Generalized LR Parsing

Visser (1997) introduced the SGLR parsing algorithm, which is based on the scannerless variant of the Noncanonical Simple LR (NSLR) parsing algorithm by Salomon and Cormack (1989) combined with the GLR parsing algorithm by Tomita (1985) and Rekers (1992).

Unlike the other parsing approaches discussed so far, scannerless parsing uses only one parsing phase: it does not use a lexer. All elements of a language can be described using a single context-free grammar. This reduces the complexity of composing grammars for different languages into a single grammar, such as embedding a grammar for JavaScript or CSS into the grammar of HTML. Referencing symbols from another grammar is enough to merge one grammar into another. In contrast, composing grammars for a two-phase parser would also entail writing a lexer that produces tokens for both grammars. We will further compare language composition with non-scannerless parsers in Section 6.1.1.

Specification of Character-Level Grammars LR grammars are *token-level grammars* because they use tokens as the terminals of the context-free grammar. SGLR parsing is scannerless, meaning that it does not use a lexer, so it uses *character-level grammars*. Language designers can describe character-level grammars using characters as terminals. The resulting parse tree has *character nodes* instead of tokens as leaves. We will use the *metalanguage* Syntax Definition Formalism 3 (SDF3) (Vollebregt, Kats, and Visser 2012; de Souza Amorim and Visser 2020) to describe character-level grammars. See Figure 2.7(a) for an example grammar specified using SDF3. Grammars written in SDF3 can be converted to an LR parse table after a preprocessing step called *grammar normalization*, as described in Section 2.4.1.

Context-Free Versus Lexical Syntax SDF3 distinguishes between **context-free syntax** and **lexical syntax** sections of a grammar. The **context-free syntax** sections describe production rules that are analogous to the rules defined in a token-level grammar. The **lexical syntax** sections describe the syntax of *lexical elements* in the language, e.g., identifier names, numbers, strings, and operators. Language designers can describe the lexical elements of the grammar in a context-free way, just like the context-free part of the grammar. Constructs to disambiguate lexical elements will be discussed in Section 2.4.2.

Parser Implementation The JSGLR parser presented in Chapter 3 is implemented as an extension to the JSGLR2 implementation of the SGLR parsing algorithm. This implementation is written in Java and integrated into the Spoofox language workbench (Kats and Visser 2010; MetaBorg 2016). The JSGLR2 parser uses parse tables generated from SDF3 for parsing. The implementation of JSGLR2 is described in more detail in Section 2.4.3.

2.4.1 Grammar Normalization

A preprocessing step called grammar normalization converts the SDF3 specification to a minimal subset of SDF3 to simplify the parse table generator. Because of grammar normalization, SDF3 can have high-level features that do not need to be supported by the parse table generator, thus simplifying the grammar specification process for language designers. Using Figure 2.7, we will discuss some of the steps that grammar normalization executes.

Kernel Syntax Grammar normalization transforms all production rules in **context-free syntax** and **lexical syntax** sections to rules in a third type of section: kernel syntax. The kernel syntax section is denoted in SDF3 using the keyword **syntax**. In kernel syntax, the *sort* symbols (representing the non-terminals) are annotated with **-CF** and **-LEX** to remember the origin of the symbol. An *injection* is added for sorts used in both syntax sections, stating that the context-free sort is equal to the lexical sort. In the grammar of Figure 2.7, this happens for the `Int` sort, among others.

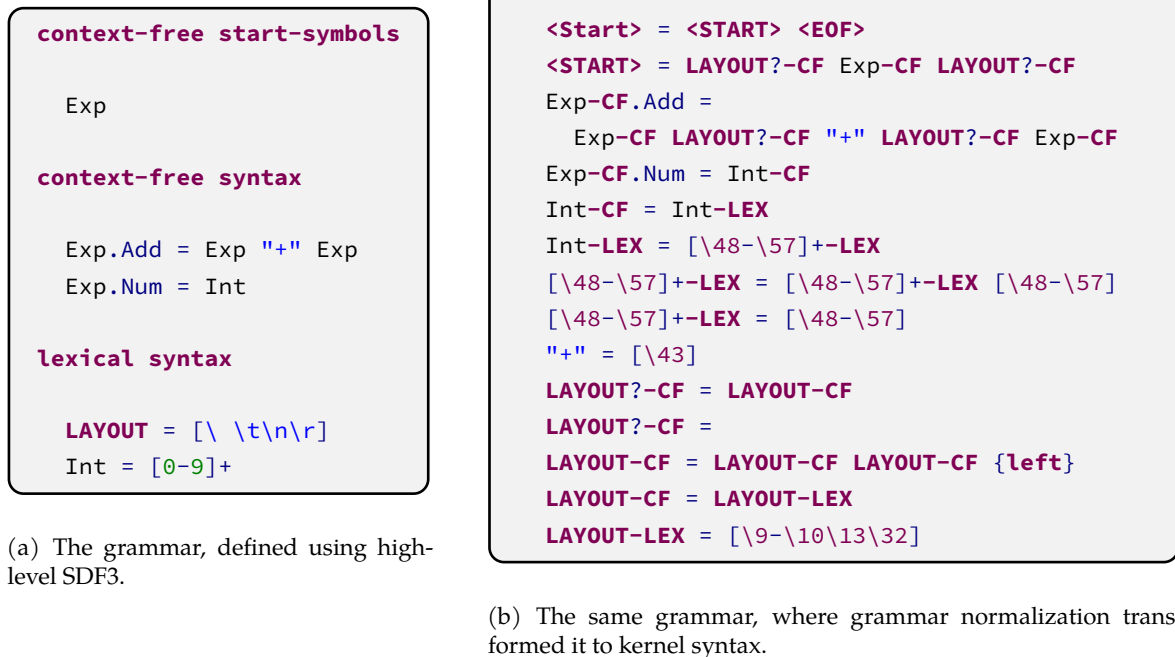


Figure 2.7: A small expression grammar, specified using SDF3 syntax. The grammar allows expressions (with the sort symbol `Exp`) to be either an addition or a number. Numbers can only be integers that consist of one or more numeric characters.

Character Classes The terminal symbols of an SDF3 grammar are *character classes*, which describe a set of characters in a compact notation.³ Grammar normalization will transform all character classes to use decimal Unicode values. For example, in the grammar of Figure 2.7(a), the `Int` sort is described by the character class `[0-9]`, which gets normalized to `[\48-\57]`.

Literal Expansion The *literals* of the grammar, consisting of non-varying lexical elements like keywords and operators, are normalized to a list of character classes corresponding to the characters of the literal. In the example of Figure 2.7, the `"+"` literal that is used as an operator in the addition rule gets normalized to the rule `"+" = [\43]`. A literal keyword like `"if"` would get `"if" = [\105] [\102]` as normalized production rule.

Layout Insertion A scannerless parser processes the input character by character. Therefore, the full grammar must also model the layout (whitespace characters) between grammar symbols. Manually adding this layout explicitly for every production rule would greatly decrease the readability of a grammar specification. Therefore, grammar normalization will insert optional layout symbols (denoted using `"LAYOUT?-CF"`) between the symbols on the right-hand side of production rules that are defined in **context-free syntax** sections. For example, the addition rule (`Exp.Add`) gets explicit `LAYOUT?-CF` symbols around the `"+"` symbol in Figure 2.7(b).

³<https://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/reference.html#character-classes>

Lists and Optionals SDF3 allows for three operators that indicate the allowed number of occurrences of a sort: `?` is used for “zero or one”, `*` is used for “zero or more”, and `+` is used for “one or more”. Each of these operators is normalized to two rules, as shown in Figure 2.8. When the list operators are used in a **context-free syntax** section, optional layout is also inserted between two sorts on the right-hand side of the normalized production rules.

```

X? = X
X? =

Y* = Y* Y
Y* = Y

Z+ = Z+ Z
Z+ = Z

```

Figure 2.8: The normalization of optionals and lists in SDF3.

Layout Production Rules To let the optional context-free layout symbol (**LAYOUT?-CF**) make use of the **LAYOUT** rule as defined in Figure 2.7(a), grammar normalization adds the production rules shown at the bottom of Figure 2.7(b). The normalization of **LAYOUT?-CF** is a combination of the optional and list normalizations as discussed in the previous paragraph. Also, the injection between the context-free and lexical is added here to make the final connection with the lexical definition of **LAYOUT** before normalization.

Start Symbol Grammar normalization adds two global start symbols to the normalized grammar. The first (**<start>**) is used in the first production rule in Figure 2.7(b), which has the second start symbol (**<START>**) and the EOF symbol (**<EOF>**) on the right-hand side. For the second start symbol, one rule is added for each start symbol defined in the **start-symbols** sections. Additionally, for start symbols defined in a **context-free start-symbols** section, this rule will also allow optional layout before and after the start symbol.

2.4.2 Lexical Disambiguation Constructs

Character-level grammars contain certain lexical ambiguities that do not occur in token-level grammars because these ambiguities are usually resolved by the lexer. In SDF3, *reject* rules and *follow-restriction* rules can be used to resolve two types of lexical ambiguities.

Reject Rules One type of lexical ambiguity that can occur, is that a string of tokens can be interpreted as different lexical elements that are all valid at that position. A typical example of this is that keywords (like **if** and **return**) can also be parsed as an identifier. To solve this, language designers using SDF3 can mark production rules with the keyword **reject**, causing the parser to reject these productions for the non-terminal on the left-hand side of the rule. Figure 2.9 shows an example of this.

Follow-Restriction Rules Another common ambiguity that arises in character-level grammars is that single identifiers could be split up into several identifiers, while there is no layout between those parts. To solve this, follow-restriction rules (denoted with **-/-**) can be used to prefer the longest match for lexical elements. Figure 2.9 shows an example with a follow-restriction rule which states that an identifier cannot be followed by a letter because that would mean that this letter should be part of that identifier.

2.4.3 Modular Parser Implementation

The JSGLR2 parser in the Spoofox IDE implements the SGLR parsing algorithm in a modular way (Denkers 2018). This allows the implementation of extensions to the parser with as little code duplication as possible, increases the maintainability of the code, and allows composing variants if their modules do not have any overlapping changes.

The architecture of the JSGLR2 parser is modular in two dimensions. First, the parser can be configured to use variants of the runtime data structures for parsing. Second, the parsing algorithm is spread over modules that each contain an implementation of one or more of the subroutines of SGLR. We will refer to Denkers (2018, §2.5) for the full SGLR algorithm.

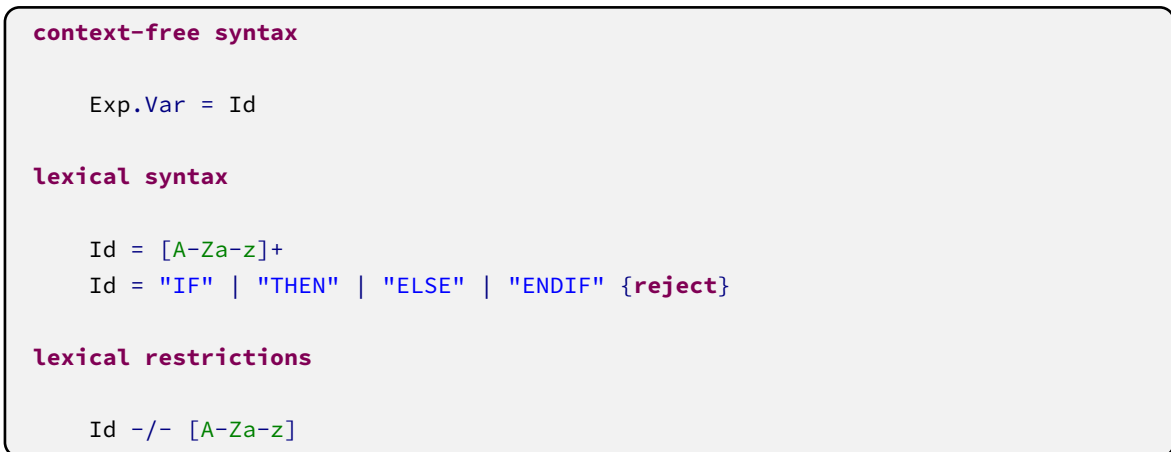


Figure 2.9: An SDF3 example that uses lexical disambiguation constructs. The rule annotated with `{reject}` forbids that an identifier is equal to one of the strings in that rule. The follow-restriction rule in the **lexical restrictions** section makes sure that an identifier cannot be followed by another letter.

Runtime Data Structures The JSGLR2 parser can use variants of the runtime data structures as described below. Constructing a variant of the parser requires passing factories for these data structures to the constructor of the `Parser` class. These factories will then instantiate the data structures during a parse.

ParseForest, ParseNode, Derivation, and CharacterNode are the data structures that make up the parse forest that the parser produces for a successful parse. Both `ParseNode` and `CharacterNode` are implementations of `ParseForest`. A `ParseNode` stores a list of `Derivations`, each of which stores a list of `ParseForests` again. A `Derivation` also stores for which production rule it was created. A `CharacterNode` stores which character it represents.

StackNode and StackLink make up the parse stack. A `StackNode` stores a state reference and a list of outgoing `StackLinks`. A `StackLink` stores references to the two `StackNodes` that it connects between, a `ParseForest`, and whether it is marked as rejected.

ParseState stores the global variables used during parsing: *active-stacks*, *accepting-stack*, *for-actor*, *for-actor-delayed*, *for-shifter*, and the input stream.

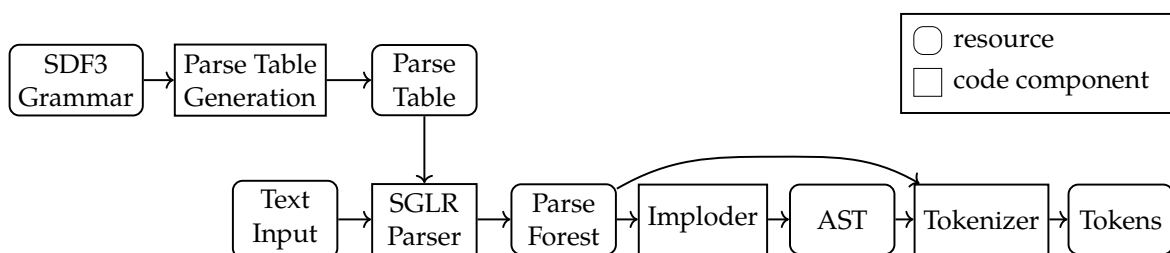


Figure 2.10: The current parsing pipeline in the Spoofox IDE. The top row of the pipeline is executed during language development, while the bottom row is executed every time a file is parsed.

Parser Modules The Java class `JSGLR2Implementation` is the entry point to the complete parsing pipeline. It ties the SGLR parsing algorithm, the *imploder*, and the *tokenizer* together. An overview of the Java classes that implement parts of the parsing pipeline is given below. Additionally, Figure 2.10 shows a schematic overview of the complete parsing pipeline from input to output. Note that the parse table generator in the top row of this figure is implemented in the SDF module in Spoofox, i.e., it is not a part of JSGLR2.

Parser contains the main parse loop, deferring several parts of the algorithm to other components. It has methods for the following procedures of the SGLR algorithm:

`PARSE`, calls `PARSECHARACTER` for all characters in the input stream;

`PARSECHARACTER`, calls `ACTOR` for every active parse stack and calls `SHIFTER` after that;

`ACTOR`, fetches an action from the parse table and acts upon it; and

`SHIFTER`, pushes the current character onto all remaining active stacks.

ReduceManager has methods for the `DoREDUCTIONS` and `DoLIMITEDREDUCTIONS` procedures, which calculate the possible reduction paths in the GSS and call `REDUCER` for every valid path. It also has a method that covers a large part of the `REDUCER` procedure. This method determines which branch of the `Reducer` to call, depending on other active parse stacks and their links to other stacks.

Reducer contains the code for the three branches of the `REDUCER` procedure: the case where both the new stack node and stack link already exist, the case where only the new stack node already exists, and the case where neither already exist.

StackManager handles the creation of stack nodes and stack links.

ParseForestManager handles the creation of parse nodes and derivations, and adding derivations to parse nodes.

Imploder transforms a parse forest to an Abstract Syntax Tree (AST). More details will be given in Section 4.1.

Tokenizer transforms the parse forest to a list of tokens and attaches these tokens to the AST. More details will be given in Section 4.2.

Parser Variants The current implementation of JSGLR2 supports the following parser variants, which have been implemented as modular extensions to the standard parser:

Elkhound McPeak and Necula (2004) introduced Elkhound parsing, an optimization to GLR parsers where it can fall back to the more efficient LR parsing algorithm when there is only one active parse stack. Denkers (2018, §5.2) describes the implementation of Elkhound in the JSGLR2 parser.

Optimized Parse Forest Denkers (2018) introduced a variant of the JSGLR2 parser that skips the creation of parse nodes when they are not needed for the final AST. This includes parse nodes that would be created for lexical production rules and **reject** rules.

Data-Dependent de Souza Amorim, Steindorfer, and Visser (2018) extend SDF3 by allowing symbols to be parameterized by data and allow arbitrary computation at parse time. They use this extension to solve deep priority conflicts, which are priority conflicts between parse nodes that are not directly connected, but instead are separated by arbitrarily many levels of parse nodes, so regular priority declarations cannot be used to resolve these conflicts.

Layout-Sensitive de Souza Amorim, Steindorfer, Erdweg, et al. (2018) extend SDF3 with layout constraint annotations, which allows parsing languages that are sensitive to layout. For example, Python uses indentation to indicate nesting in blocks of code, requiring this layout-sensitive parsing approach.

Error Recovery de Jonge et al. (2012) add error recovery to SGLR and implemented this in JSGLR1, the previous version of the SGLR implementation in Spoofox. The implementation of this extension in JSGLR2 is a work in progress.

We implemented the ISGLR parser presented in Chapter 3 in the modular architecture of JSGLR2. Details of the implementation in this architecture will be given in Section 3.5.

This page is intentionally left blank.

Chapter 3

Incremental Scannerless Generalized LR Parsing

This chapter presents and analyses the Incremental Scannerless Generalized LR (ISGLR) parsing algorithm to answer the research question stated in Chapter 1. This parsing algorithm combines Scannerless Generalized LR (SGLR) parsing and Incremental Generalized LR (IGLR) parsing. The differences of these two parsing algorithms with respect to GLR parsing are orthogonal to each other and are combined without difficulties. While the algorithmic differences are orthogonal, we show that there exist non-trivial interactions between these two techniques.

The ISGLR parsing algorithm works as follows. First, it calculates a list of changes (called a *diff*) between the old and the new version of the input string. Then, starting at the root node, the parser tests whether parse nodes can be reused in the current context, and if not, breaks them down and continues with their children. The parser will always break down parse nodes that contain any of the changed positions as calculated by the *diff*. When the parser encounters irreusable parse nodes, which were created while the parser was parsing non-deterministically (see Section 2.3), it will break those down as well.

In Section 3.1, we apply the ISGLR parsing algorithm to an example to show how it works in practice. We show that increased non-determinism caused by scannerless parsing results in less reuse during an incremental parse using more examples in Section 3.2. The ISGLR parsing algorithm uses the same state matching test and lookahead test as Wagner and Graham (1997b) (see Section 2.3) to test whether reusing a parse node is valid, as we show in Section 3.3.

We write out the entire parsing algorithm in detail in Section 3.4. In particular, we highlight the changes required to the SGLR parser algorithm by Visser (1997). In Section 3.5, we describe the required changes to the JSGLR2 implementation by Denkers (2018), as described in Section 2.4.3.

3.1 ISGLR Parsing by Example

In this section, we will give an intuition of the ISGLR parsing algorithm using an example. The example is based on a small grammar that allows a list of variable assignment statements of the shape `stmt.Assign = ID "=" Exp`. The allowed expressions are additions, multiplications, numbers and variables. The full SDF3 grammar is shown in Figure 3.1(a) and the normalized grammar in Figure 3.1(b).

In this example, we will consider the incremental parse of the input string “`ab =42 * 42`” followed by “`ans =42 * 42`” (hereafter called version 1 and version 2, respectively). The parse tree of the first input string is shown in Figure 3.2.

```
context-free start-symbols
Start

context-free syntax
Start = Stmt+

Stmt.Assign = ID "=" Exp

Exp.Add = Exp "+" Exp {left}
Exp.Mul = Exp "*" Exp {left}
Exp.Var = ID
Exp.Num = NUMBER

context-free priorities
Exp.Mul > Exp.Add

lexical syntax
ID = [a-z]+
NUMBER = [0-9]+

LAYOUT = [\ \t\r\n]

lexical restrictions
ID -/- [a-z]
NUMBER -/- [0-9]
```

(a) The grammar, defined using high-level SDF3.

Figure 3.1: The SDF3 definition of a small grammar that allows a list of statements that assign an expression to a variable. The allowed expressions are additions, multiplications, numbers, and variables.

syntax

```

<Start> = <START> <EOF>
<START> = LAYOUT?-CF Start-CF LAYOUT?-CF
Start-CF = Stmt+-CF
Stmt+-CF = Stmt-CF
Stmt+-CF = Stmt+-CF LAYOUT?-CF Stmt-CF
Stmt-CF.Assign = ID-CF LAYOUT?-CF "=" LAYOUT?-CF Exp-CF
Exp-CF.Add = Exp-CF LAYOUT?-CF "+" LAYOUT?-CF Exp-CF {left}
Exp-CF.Mul = Exp-CF LAYOUT?-CF "*" LAYOUT?-CF Exp-CF {left}
Exp-CF.Var = ID-CF
Exp-CF.Num = NUMBER-CF
ID-CF = ID-LEX
ID-LEX = [\97-\122]+-LEX
[\97-\122]+-LEX = [\97-\122]
[\97-\122]+-LEX = [\97-\122]+-LEX [\97-\122]
NUMBER-CF = NUMBER-LEX
NUMBER-LEX = [\48-\57]+-LEX
[\48-\57]+-LEX = [\48-\57]
[\48-\57]+-LEX = [\48-\57]+-LEX [\48-\57]
"=" = [\61]
"+" = [\43]
"*" = [\42]
LAYOUT?-CF = LAYOUT-CF
LAYOUT?-CF =
LAYOUT-CF = LAYOUT-CF LAYOUT-CF {left}
LAYOUT-CF = LAYOUT-LEX
LAYOUT-LEX = [\9-\10\13\32]

```

priorities

```

Exp-CF.Mul > Exp-CF.Add,
Exp-CF.Add left Exp-CF.Add,
Exp-CF.Mul left Exp-CF.Mul,
LAYOUT-CF = LAYOUT-CF LAYOUT-CF left LAYOUT-CF = LAYOUT-CF LAYOUT-CF

```

(b) The SDF3 grammar of Figure 3.1(a), normalized to kernel syntax.

Figure 3.1 (Continued)

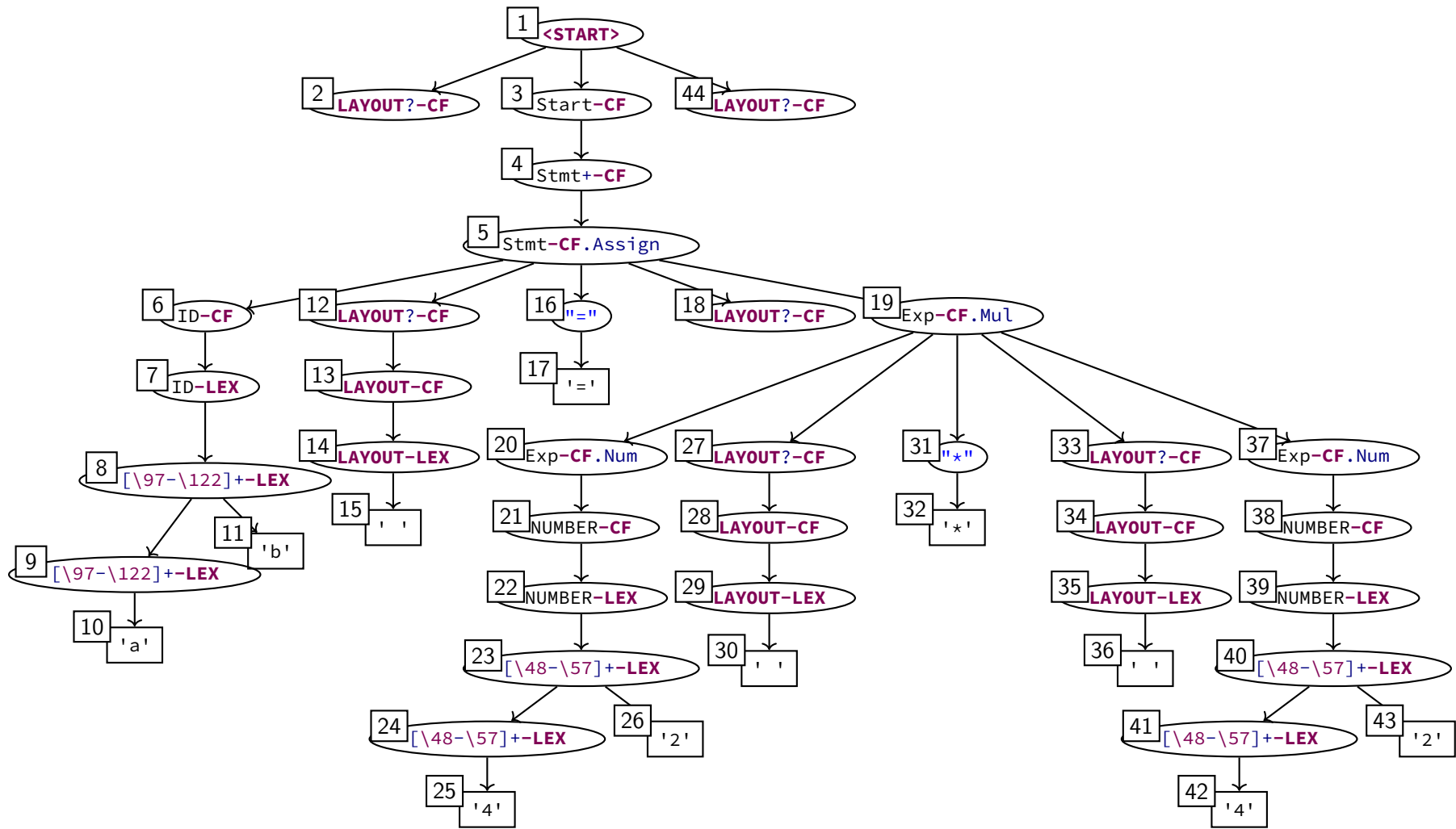


Figure 3.2: The parse tree for the input “ab =42 * 42” according to the grammar in Figure 3.1(b).

```

0 1 2 3 4 5 6 7 8 9 10 11
ab * 42

0 1 2 3 4 5 6 7 8 9 10 11 12
ans * 42

deletedStart = 1
deletedEnd = 2
inserted = "ns"

```

Figure 3.3: The diff between version 1 and version 2.

Calculate Updates Using Diff As a first step, we compute the character-by-character difference between the two versions of the input string, using an existing diff algorithm.¹ This algorithm gives a list of changes as a result. Calculating the diff between version 1 and version 2 yields a single change, as shown in Figure 3.3. This change indicates that the second character (the “b”) is deleted and replaced with “ns”.

In general, each text change is modelled using three values: the start and end positions of the deleted text, and the string that is inserted at the start position. This models text replacements, insertions, and deletions. For insertions, the start and end index are equal, because for this change no text is deleted. For deletions, the string that is inserted is left empty.

When the calculated diff contains changes that have different lengths for the deleted and inserted text, the positions in subsequent changes become outdated. We will abstract over this detail since it only requires simple arithmetic to update these positions. In the remainder of this chapter, we assume that the parser can always correctly detect whether a certain position contains a change, based on this abstraction.

Incremental Input Stack The ISGLR parser uses an incremental *input stack* to consume input from. Whereas a regular input stream would store only characters, the incremental input stack can also store entire parse nodes. When the parse node at the top of the stack cannot be reused for whatever reason, it will be broken down and its children will be pushed back onto the stack in reverse order so that the first child ends up on top of the stack. The main reason for having to break down a parse node is when it contains a change from the calculated diff. Eventually, all parse nodes along the *spine* between the root node and the changes will be broken down. Other reasons for breaking down parse nodes will be discussed in the remainder of this chapter.

The approach of using an input stack differs from the IGLR parsing algorithm of Wagner and Graham (1998), which performs a walk along the parse tree after updating the tokens in the leaf nodes, as explained in Section 2.3. This requires the parse nodes to maintain a reference to their parent, which needs to be updated if a subtree is reused in a different node. Since we treat parse nodes as immutable, we use the incremental input stack to keep track of which parse nodes still need to be processed.

¹In particular, the diff algorithm implemented in JGit: <https://download.eclipse.org/jgit/site/5.6.0.201912101111-r/apidocs/org/eclipse/jgit/diff/package-summary.html>

Incremental Parsing Now, we start parsing version 2. In Figure 3.4, we will follow the contents of the incremental input stack and the parse stack during parsing. The incremental input stack is initialized with the root of the parse tree of Figure 3.2 (node 1) and an EOF marker \$.

Node 1 contains the change that was calculated by the diff, so the parser breaks it down and pushes its children (2, 3, and 44) onto the input stack. The parser then shifts node 2 onto the parse stack, since its symbol (LAYOUT?-CF) is in the goto table of the start state.

Following this, the parser has to break down multiple parse nodes successively, all the way until character node a (10) is on top of the input stack. This node can then be shifted onto the parse stack. Note that its parent node 9 is not part of the change, but it is directly followed by a change, so it should still be broken down (more on this in Section 3.3.2).

Now that character node b is at the top of the input stack, the update can be applied. Node b is removed from the stack, while two new character nodes n and s are pushed back onto it.

With character node n at the top of the input stack, the parser can reduce parse node a, creating node 47 with sort [\97-\122]+-LEX. On top of this new node, the parser shifts the n onto the parse stack.

The parser then reduces the two nodes at the top of the parse stack into node 48, which is another parse node with sort [\97-\122]+-LEX. After shifting the s node, it performs the same reduction, creating node 49.

Then, the parser performs two reductions, from parse node 49 to ID-LEX and from that node to ID-CF. Note that this is a node with the same sort as node 6, which was the first child of node 5 (with sort Stmt-CF.Assign) in the parse tree of the first version. Therefore, the parser can shift all other children of node 5 from the input stack onto the parse stack and reduce them all to a new parse node Stmt-CF.Assign.

Finally, the parser performs two more reductions, shifts parse node 44, and reduces the entire parse stack to the <START> symbol. With only the \$ node left in the input stack, the parser accepts the input. Figure 3.5 shows the resulting parse tree.

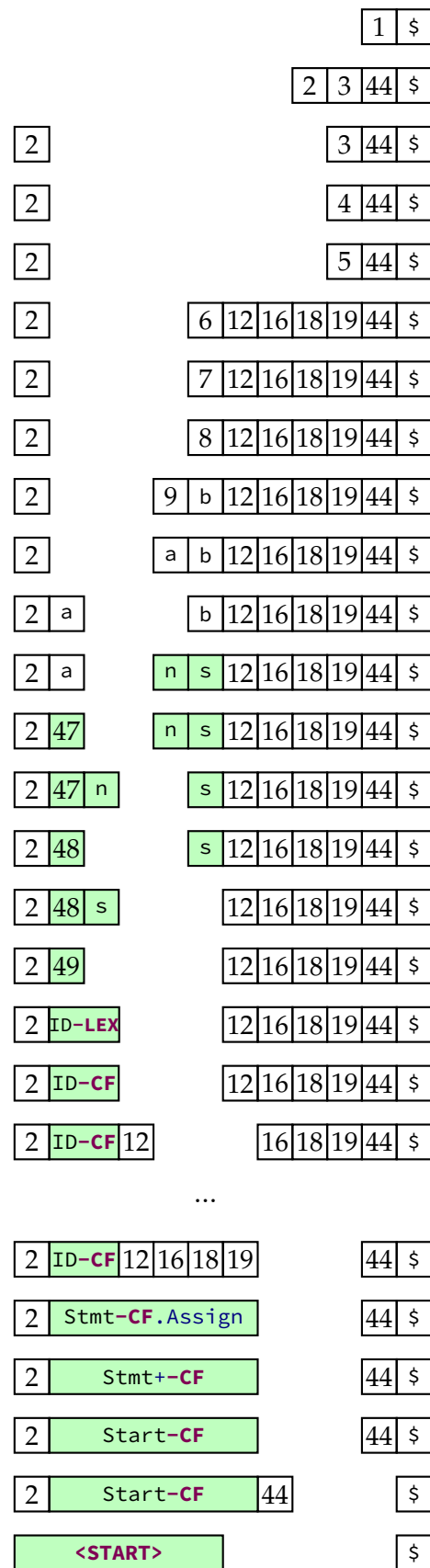


Figure 3.4: The parse stack (left) and the input stack (right) during an incremental parse, using the change from Figure 3.3.

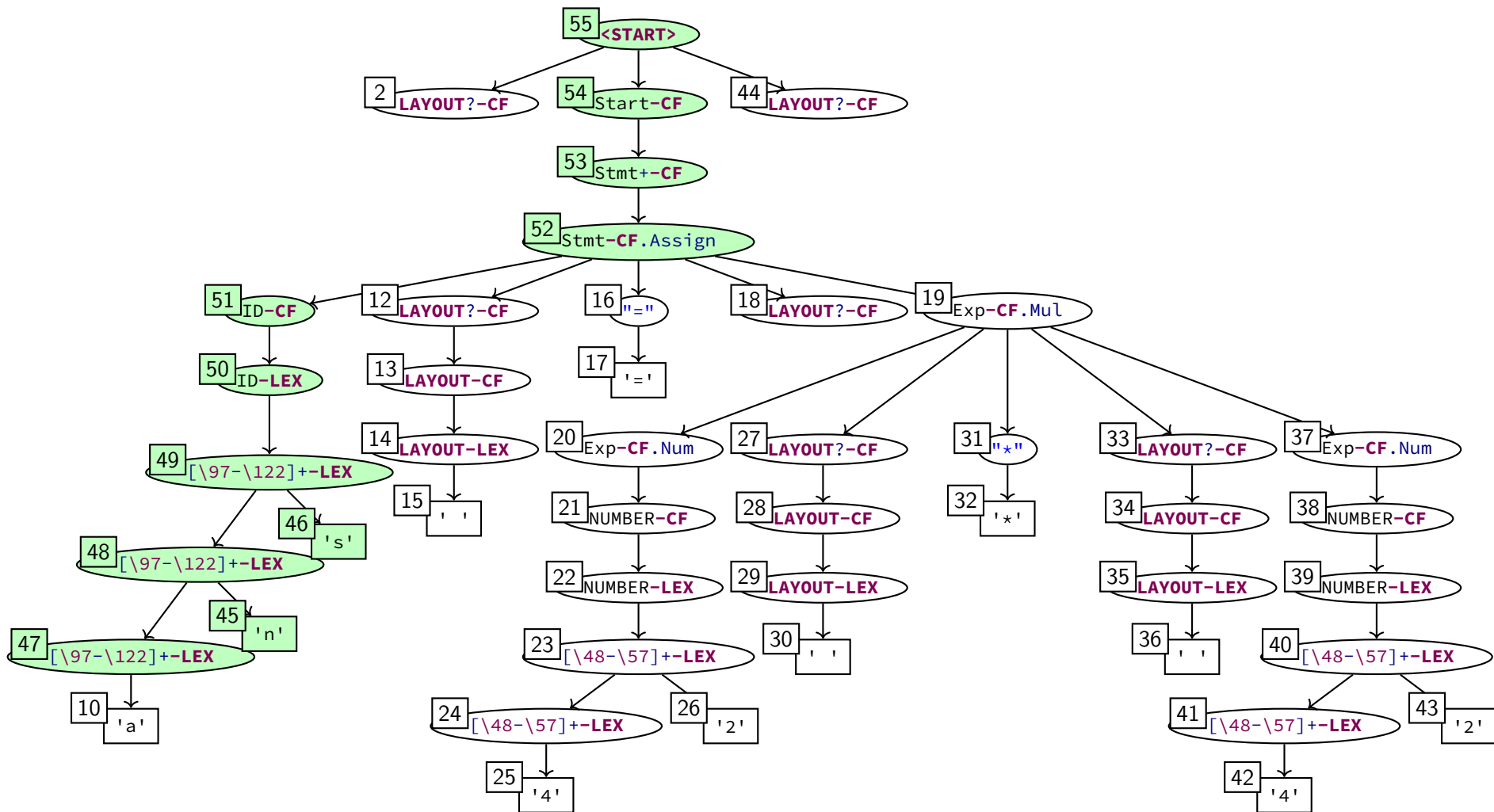


Figure 3.5: The resulting parse tree after first parsing “ $ab =42 * 42$ ” and then incrementally parsing “ $ans =42 * 42$ ”. The white nodes were created during the initial parse (see Figure 3.2) and the green nodes were created during the incremental parse (see Figure 3.4).

3.2 Non-determinism in ISGLR Parsing

As explained for IGLR parsing in Section 2.3, non-deterministic parsing results in irreusable parse nodes, which can never be reused in a subsequent incremental parse if they end up on top of the input stack. While Wagner and Graham (1997b, §5) showed that non-deterministic parsing rarely occurs for token-based parsers, the parsing of character-level grammars relies more on non-deterministic parsing for disambiguation (Visser 1997, §2.2). In this section, we show some examples of this, using the grammar of Figure 3.1.

3.2.1 Example With Layout Between Operators

Consider a batch parse of the input “a = x + y + z”. After parsing “a = x + y”, the parse stack contains one parse node for each character, as in stack (1) in Figure 3.6.

With ‘ ’ as the next character, the parser can take four successive *Reduce* actions. First, the top five nodes can be reduced to `Exp-CF.Add` (2), which can then be reduced together with four other parse nodes to `Stmt-CF.Assign` (3), which can, in turn, be reduced to `Stmt-CF+` (4) and further to `Start` (5). Figure 3.6 shows how these parse stacks look like, together with the parse nodes that are stored in the stack links. Of the five parse stacks, all of them can shift the ‘ ’ except for stack (3):

- (1) expects a ‘*’ following the space, since `Exp-CF.Mul` has priority over `Exp-CF.Add`.
- (2) expects a ‘+’ following the space, since `Exp-CF.Add` is left-associative.
- (4) expects another `stmt` following the space.
- (5) expects the End-of-File (EOF) following the space.

The reason why so many parse stacks stay active is that the parser can only use one character as lookahead. It cannot look after the space to see that it is followed by a ‘+’. After all, there could be an arbitrary number of spaces between the ‘y’ and the ‘+’, and with a different grammar, there could even be comments (which, in SDF3 grammars, are generally defined in a context-free way as part of the `LAYOUT`). Only after reducing the space via `LAYOUT-LEX` and `LAYOUT-CF` to `LAYOUT?-CF` and seeing that it is followed by a ‘+’, the parser can discard all parse stacks except stack (2).

In the resulting parse tree, the `Exp-CF.Add` node corresponding to “x + y” is an irreusable parse node because it was created when there were two parse stacks active, both expecting to shift layout (stacks (3)–(5) did not exist yet). The `Exp-CF.Add` node corresponding to “x + y + z” is also marked as irreusable. In this case, there was only one active parse stack, but the parser still had multiple actions: at the same time of creating this parse node, an empty `LAYOUT?-CF` node was created as well. This is because the latter parsing branch expects to find another ‘+’ following the `LAYOUT?-CF`.²

Now consider an incremental parse with input “a = x + y * z”, where the second ‘+’ changes to ‘*’. The `Exp-CF.Add` parse node corresponding to “x + y” will be exposed on the top of the input stack and the parser will break it down since it is irreusable. However, most of the children of this parse node can be directly reused (except for the parse node corresponding to the space between ‘x’ and ‘+’ because it was also irreusable, but that will be rebuilt), after which the parse stack looks exactly the same as stack (1). Before shifting the ‘ ’ preceding the ‘*’, the parser can take the same four *Reduce* actions again. However, this

²This was also the case for the example in Section 3.1, but we ignored this there for simplicity. Also, note that SDF3 currently uses an LR(0) parse table generator. An LR(1) parse table would be able to see the EOF that follows it, and in that case, there is only one possible *Reduce* action and the empty `LAYOUT?-CF` parse node is not created. However, the parser can always be “tricked” into non-determinism by adding extra layout before the EOF: in that case, it will still try to create a layout node, this time being non-empty.

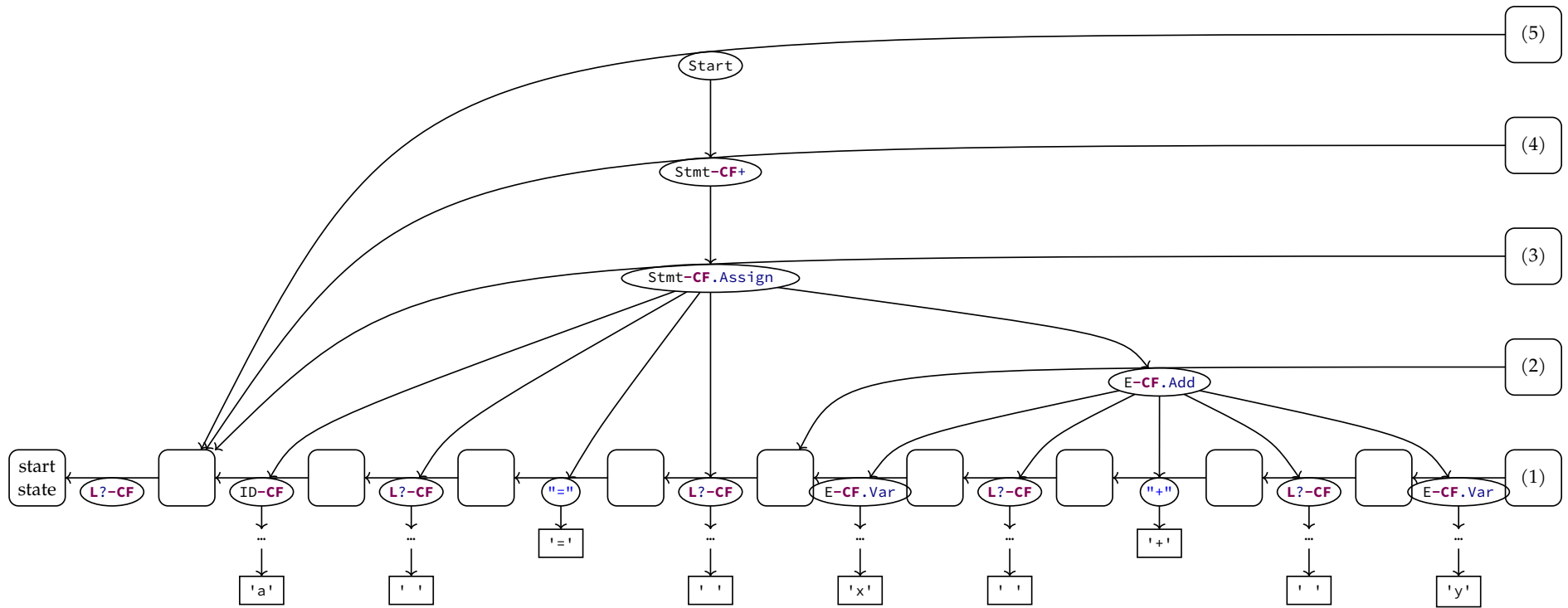


Figure 3.6: The active parse stacks after parsing “a = x + y”, according to the grammar of Figure 3.1. The stacks are displayed right before a ' ' is shifted onto stacks (1), (2), (4), and (5). In the parse nodes, E_{xp} is abbreviated to E and **LAYOUT** is abbreviated to L.

time, only stack (1) will survive, and after processing ' ' and 'z' it creates an `Exp-CF.Mul` node. Note that this `Exp-CF.Mul` node is *not* irreusable because there is only one *Reduce* action possible: the grammar does not allow any (optional) layout following an `Exp-CF.Mul` node since it has the highest priority of all `Exp` production rules.

3.2.2 Example With Layout Between List Items

Consider a batch parse with input “`x = 3 y = 4 z = 5`”. After parsing “`x = 3`”, the parser is in a similar situation as in the start of the example in Section 3.2.1, shown as stack (1) in Figure 3.7(a). With one parse node on the parse stack for every character parsed so far, the parser is about to shift a ' '. Just like in the previous example, the parser can first reduce the current parse stack (1) multiple times, to `stmt-CF.Assign` (2), `stmt-CF+` (3), and finally to `start` (4). At this point, all of these parse nodes are marked as irreusable, because stacks (1), (3), and (4) can also shift a space as second action:

- (1) expects a '*' or '+' following the space.
- (3) expects another `stmt` following the space (this parse stack survives).
- (4) expects the EOF following the space.

After parsing “`x = 3 y = 4`”, the parse stack will look like stack (5) in Figure 3.7(b). Here, a similar thing happens as with the first statement: the top five parse node are reduced to `stmt-CF.Assign` (6), after which the top three parse node are reduced to `stmt-CF+` (7), which is in turn reduced to `start` (8). Again, because a space follows next, stacks (5), (7), and (8) expect to shift it, so the parse nodes created from these reductions are again all irreusable.

The same thing happens when the parser creates the `stmt-CF+` parse node that spans the entire input, after also parsing the final “`z = 5`”. In the resulting parse tree, all parse nodes corresponding to `stmt-CF+` and `stmt-CF.Assign` production rules are marked as irreusable. Now consider an incremental parse of the input “`x = 3 y = 4 z = 7`”. Even though only the very last character has changed, all `stmt` nodes will be broken down because they are irreusable and exposed on top of the input stack.

Do note that adding semicolons (';') at the end of a `stmt` production rule makes sure that a `stmt-CF.Assign` parse node is *not* marked as irreusable.³ The semicolon cannot be parsed as anything else than the end of a statement, and since it always consists of exactly one character, the parser does not need to consume any further characters before it can create a parse node for the statement. Similarly, blocks of statements enclosed in curly braces ('{' and '}') are also not irreusable.

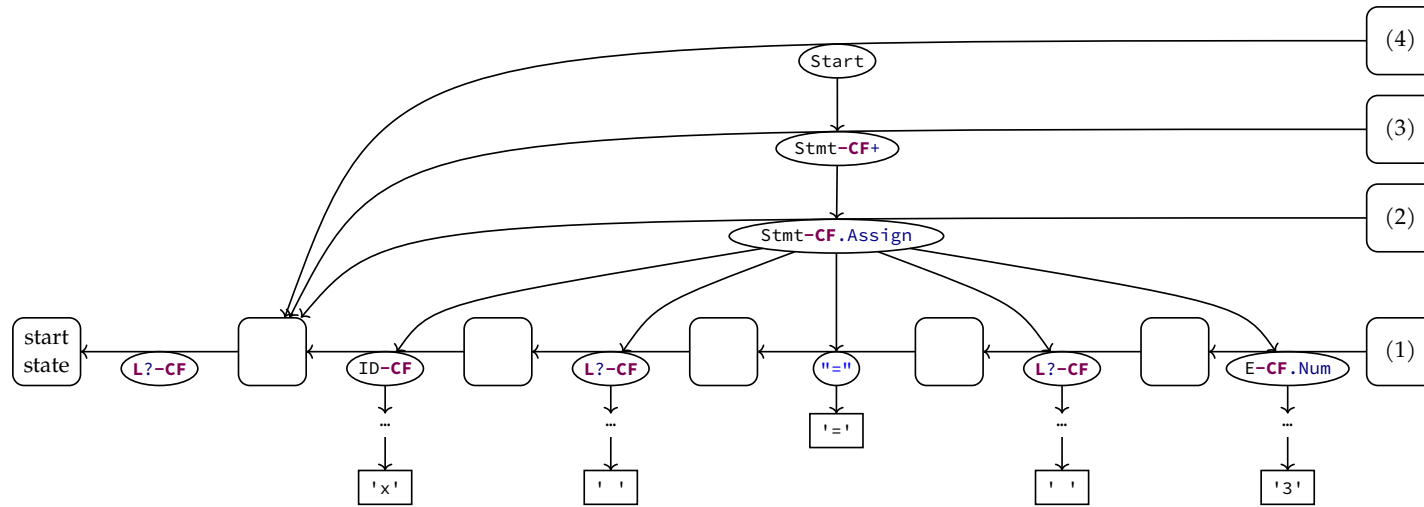
3.2.3 Example With Identifier Versus Keyword

For this example, we will extend the grammar of Figure 3.1 with two new production rules as shown in Figure 3.8. We add a return statement (`stmt.Return`) that consists of the keyword “`return`” and an expression (`Exp`). Note that this keyword could also be parsed as an identifier (`ID`), according to the rule `ID = [a-z]+`. Therefore, we also add a `{reject}` rule to disambiguate this.

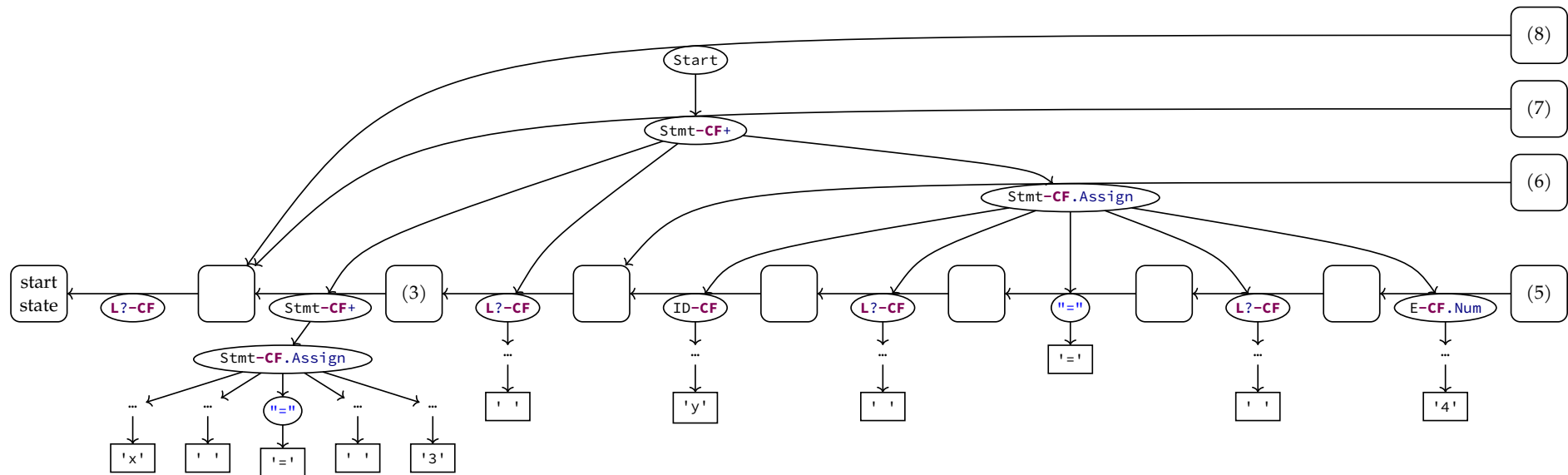
Consider a batch parse of the input “`return ab`”. The “`return`” part can be parsed in two ways: either as the keyword “`return`” (1) or as `[\\97-\\122]+-LEX` (2). Figure 3.9(a) shows these two parse stacks right after shifting the final 'n'.

Following this *Shift* action, multiple *Reduce* actions follow, of which the final result is shown in Figure 3.9(b). First, the six character nodes of parse stack (1) are reduced to the

³Of course, before a parser can actually reuse them, they must still pass the state matching test, but that will only fail when the context changed.

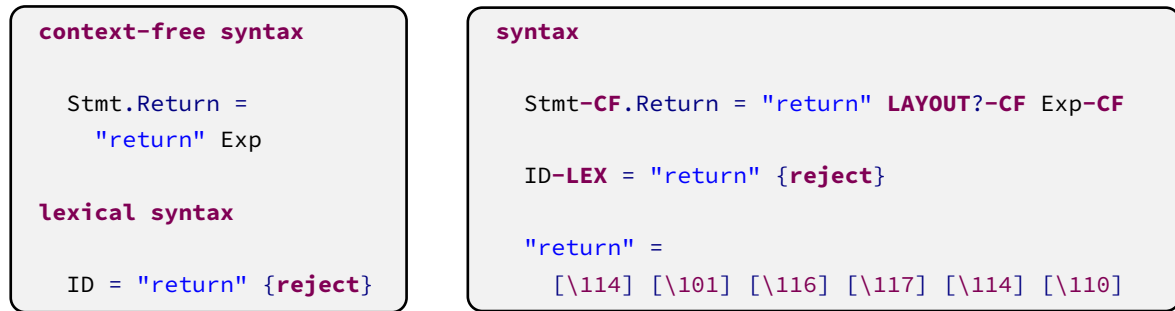


(a) The active parse stacks after parsing "x = 3". The stacks are displayed right before a ' ' is shifted onto stacks (1), (3), and (4).



(b) The active parse stacks after parsing "x = 3 y = 4". The stacks are displayed right before a ' ' is shifted onto stacks (5), (7), and (8).

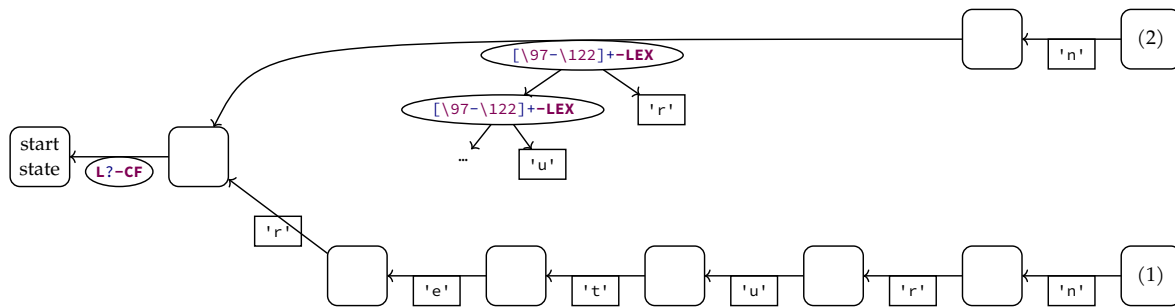
Figure 3.7: The active parse stacks at two points during the parsing of "x = 3 y = 4 z = 5", according to the grammar of Figure 3.1. In the parse nodes, Exp is abbreviated to E and LAYOUT is abbreviated to L.



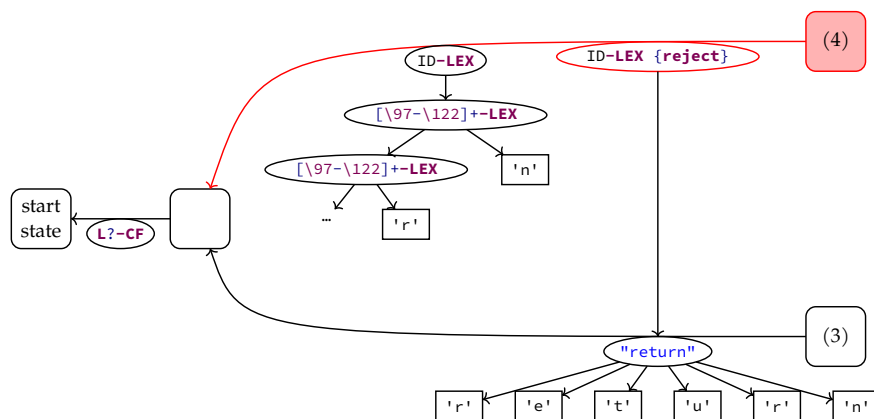
(a) The production rules, defined using high-level SDF3.

(b) The same production rules, where grammar normalization transformed them to kernel syntax.

Figure 3.8: Two production rules, added to the grammar of Figure 3.1, specified using SDF3 syntax. The first rule introduces a return statement and the second rule prevents the keyword "return" to be parsed as an identifier.



(a) The active parse stacks after parsing "return". The stacks are displayed right after shifting the 'n'.



(b) The active parse stacks after parsing "return". The stacks are displayed after processing all possible Reduce actions based on Figure 3.9(a). The stack link of stack (4) contains two possible derivations for the same sort. This link is drawn in red because the second derivation marked it as rejected. Since stack (4) has no remaining valid stack links, the parser will no longer consider it for applying new actions. Therefore, the following ' ' can only be shifted onto stack (3).

Figure 3.9: The active parse stacks at two points during the parsing of "return ab", according to the grammar of Figure 3.1, extended with the production rules of Figure 3.8. In the parse nodes, LAYOUT is abbreviated to L.

keyword "return" (3). Then, the $[\backslash 97-\backslash 122]^+-\text{LEX}$ parse node of stack (2) is reduced to ID-LEX (4). The "return" is also reduced to ID-LEX , because of the `{reject}` rule. Note how this last reduction merges into stack (4) since it is another way of parsing the sort ID-LEX . This marks the stack link of stack (4) as rejected, which means that the parser will no longer consider it as an active parse stack since it has no remaining valid stack links.

Following these *Reduce* actions, parse stack (3) continues by shifting a ' ' as part of the LAYOUT?-CF of the production rule for stmt-CF.Return . In the resulting parse tree, the parse node corresponding to the "return" keyword is marked as irreusable, because it was created while the parser was also exploring the possibility that it could have been an identifier.

Now consider an incremental parse for the input "return ans". The parser will definitely break down the stmt-CF.Return parse node because its last child contains changes, making the "return" parse node end up on top of the parse stack. Because this is an irreusable parse node, the "return" keyword will need to be parsed from scratch.

3.3 Valid Parse Node Reuse

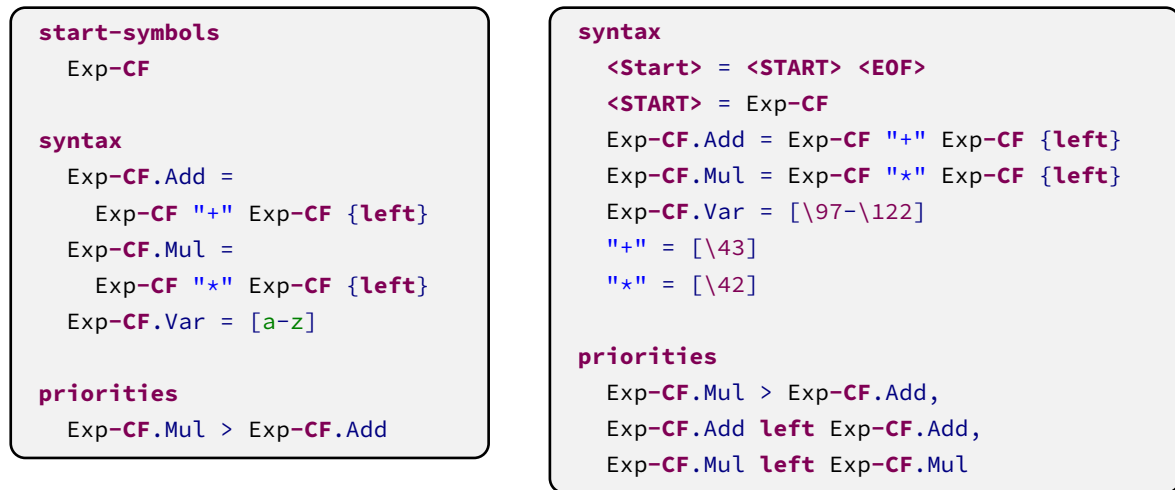
The ISGLR parser can only reuse an unchanged parse node if the context surrounding this node allows this. Following the approach of the IGLR parser by Wagner and Graham (1997b) as described in Section 2.3, we use a state matching test to check the context to the left and we test if the lookahead has changed to check the context to the right. For the lookahead test, we only need to test one character ahead in a deterministic setting. In order to also check the lookahead following a non-deterministic region, we will break down irreusable parse nodes that end up on top of the input stack to reconsider all possible ways of parsing that part of the input, as we discussed in Section 3.2.

In SDF3, it is possible to construct grammars that do not have optional layout inserted between context-free symbols, which avoids most occurrences of non-deterministic parsing. Figure 3.10 shows such a grammar, defined using kernel syntax, so that no optional layout is inserted during grammar normalization. In this expression grammar, the identifiers and operators always consist of exactly one character. This makes a scannerless parser behave exactly in the same way as a token-based parser would. We use this grammar to show two examples of how the state matching test (Section 3.3.1) and the lookahead test (Section 3.3.2) prevent invalid reuse of parse nodes in the ISGLR parsing algorithm.

3.3.1 State Matching Test

Consider a batch parse with input "x+y*z". The priorities in the grammar of Figure 3.10 make sure that the resulting parse tree has the Exp-CF.Mul node as the right child of the Exp-CF.Add node, i.e., it could be parenthesized as "x+(y*z)". Note that this input is parsed fully deterministically thanks to these priorities, so none of the parse nodes is marked as irreusable. This is regardless of the conflicts in the parse table, as we will explain in Section 3.3.3.

Now consider an incremental parse of the input "x*y*z". The "+" changes into a "*", so the Exp-CF.Add parse node from the previous parse is broken down. After parsing "x*", the parse stack contains one Exp-CF.Var node and one "*" node, as shown in Figure 3.11. Parse state 6 is on top of the stack, which is different from the first parse after the parsing "x+": the node for "y*z" (Exp-CF.Mul) was created on top of state 8 during the previous parse. Because this parse node fails the state matching test, it cannot be directly reused and it is broken down. The node for "y" (Exp-CF.Var) and the node for "*" ("*") are also broken down because of a failing state matching test. After that, incremental parsing continues as normal, with a resulting parse tree that can be parenthesized as "(x*y)*z".



(a) The grammar, defined using SDF3 kernel syntax.

(b) The same grammar, after grammar normalization.

state	actions					gotos					
	[+]	[*]	[a-z]	<EOF>	all	"+"	"*"	E.A	E.M	E.V	<START>
0			S(1)					3	2	2	4
1					R(E.V)						
2	S(7)	S(5)			R(<START>)	8	6				
3	S(7)				R(<START>)	8					
4				Accept							
5					R("*")						
6			S(1)							9	
7					R("+")						
8			S(1)						10	10	
9					R(E.M)						
10		S(5)			R(E.A)		6				

(c) The parse table of the grammar, as generated by SDF3. Shift actions are abbreviated to 'S' and Reduce actions to 'R'. The sort and constructor names have been abbreviated to single letters. Note that the parse table generator of SDF3 produces one goto state per production rule (Visser 1997, §5.4), rather than one per sort (as for LR generators). The Reduce actions are valid for any lookahead since SDF3 currently uses LR(0), and we will prefer a Shift action over a Reduce action to resolve the conflicts, as will be explained in Section 3.3.3.

Figure 3.10: A small expression grammar defined using SDF3 kernel syntax, so that it does not get optional layout (LAYOUT?-CF) in between production symbols or surrounding the start symbol during grammar normalization.

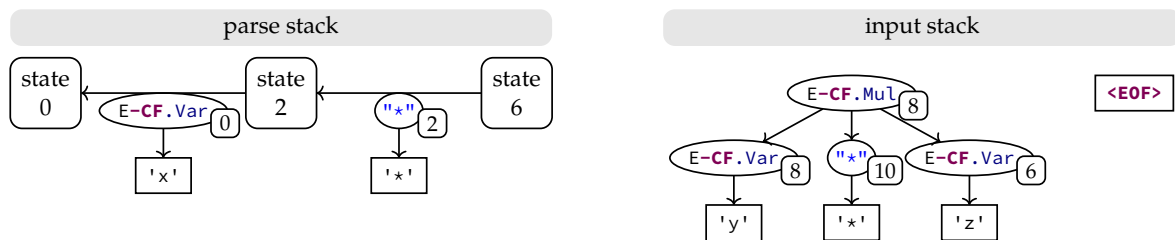


Figure 3.11: The active parse stack and remaining input stack during an incremental parse from "x+y*z" to "x*y*z", according to the grammar of Figure 3.10. The parse state references stored in parse nodes are shown as labels at the bottom right of a node. In the parse nodes, Exp is abbreviated to E.

3.3.2 Lookahead Test

Consider a batch parse of the input “ $x+y+z$ ”. Because the `Exp-CF.Add` production rule is left-associative, the resulting parse tree can be parenthesized as “ $(x+y)+z$ ”. Just as in the previous example, this input is parsed fully deterministically, so none of the parse nodes is marked as irreusable.

Now consider an incremental parse with input “ $x+y*z$ ”. The priorities in the grammar of Figure 3.10 dictate that the new input should be parsed as “ $x+(y*z)$ ”. After updating the second “ $+$ ” to “ $*$ ”, a parser without the lookahead test would see that the parse node corresponding to “ $x+y$ ” is not changed, and blindly reuse it. However, this parse node should actually be broken down because the lookahead changed, so the parser should take a different action this time. Instead of reducing to `Exp-CF.Add`, it should now shift the ‘ $*$ ’.

In the ISGLR parsing algorithm, we implement the lookahead test by considering the character preceding an edit as “changed” when calculating whether the parse node at the top of the input stack contains a change. In this example, this means that the parser considers the “ y ” preceding the new “ $*$ ” as changed. This will trigger the breakdown of the parse node corresponding to “ $x+y$ ” and allow the parser to pick a different action. For this approach, we do not need to remember what the previous lookahead was for every parse node, at the cost of potentially breaking down slightly more parse nodes than necessary.

3.3.3 Conflicts in This Parse Table

In this section, we have ignored the *Shift/Reduce* conflicts in the parse table of Figure 3.10(c). These conflicts are present because SDF3 currently uses an LR(0) parse table generator. In an LR(1) parse table, the *Reduce* actions would not overlap with the *Shift* actions.

For our examples, this means that the parser sometimes tries to reduce to `Exp-CF.Add` or `<START>` while it should only do a *Shift* action, so it temporarily creates a second parse stack, similar to the examples in Sections 3.2.1 and 3.2.2. However, because the *Shift* action can only be executed on the first parse stack, the other parse stack is immediately discarded. None of the irreusable parse nodes created during the small period where multiple parse stacks were active, ends up in the final result, because their parse stacks are discarded.

3.4 ISGLR Parsing Algorithm

In this section, we present the full ISGLR parsing algorithm, broken up into separate functions and procedures. The changes with respect to the SGLR parsing algorithm by Visser (1997) (and in particular, the implementation by Denkers (2018, §2.5)) are indicated using a vertical bar in the left margin of the page and will be explained per procedure. These changes mostly follow the ideas of Wagner and Graham (1997b), with some exceptions: we have a different way of applying changes to the parse tree (Algorithms 1 and 3) and we work around the fact that the parse table generated by SDF3 does not support directly indexing the action table using a production rule instead of a character (Algorithms 4 and 5).

The ISGLR parsing algorithm uses the functions `PEEK`, `POP`, and `BREAKDOWN`, that act on an incremental input stack. `PEEK` returns the node on top of the stack without removing it, `POP` removes the node on top of the stack and returns it, and `BREAKDOWN` breaks down the top node of the input stack and pushes back its children in reverse order.

The `PARSE` function in Algorithm 1 has a different signature than Visser’s algorithm: it now allows passing the input string and parse result of the previous version. These two new parameters are allowed to be empty, in which case we start a batch parse by initializing the input stack with only character nodes. In the case of an incremental parse, we calculate the `DIFF` between the previous and current version of the input string according to Figure 3.3.

We initialize the incremental input stack with the EOF marker (\$) and the root node of the previous parse tree. We break down the previous parse tree along the spine between the root node and the changes at the start of the PARSESTEP procedure of Algorithm 2. Additionally, we define a global boolean variable to remember whether the parser is in multiple states, i.e., whether it is parsing non-deterministically. Finally, the parse loop has been adapted to use the incremental input stack instead of iterating over all characters in the input.

Algorithm 1 The PARSE function of the ISGLR algorithm.

```

1: function PARSE(global parse-table, input, previous-input, previous-tree)
2:   if previous-input =  $\emptyset \vee$  previous-tree =  $\emptyset$  then
3:     global updates  $\leftarrow \emptyset$ 
4:     global input-stack  $\leftarrow$  [ $\$$ ] + reversed input            $\triangleright$  stack of only character nodes
5:   else
6:     global updates  $\leftarrow$  DIFF(previous-input, input)
7:     if updates =  $\emptyset$  then                                    $\triangleright$  list of changes
8:       return previous-tree                                    $\triangleright$  If there are no changes, return previous result
9:     global input-stack  $\leftarrow$  [ $\$,$  previous-tree]  $\triangleright$  stack of parse nodes and character nodes
10:    global accepting-stack  $\leftarrow \emptyset$ 
11:    init-stack  $\leftarrow$  new stack with start state of parse-table
12:    global active-stacks  $\leftarrow$  {init-stack}                  $\triangleright$  list of parse stacks
13:    global multiple-states  $\leftarrow$  false
14:    while PEEK(input-stack)  $\neq$   $\$$   $\wedge$  active-stacks  $\neq \emptyset$  do
15:      PARSESTEP()
16:    if accepting-stack  $\neq \emptyset$  then
17:      return the parse node on the only link of accepting-stack
18:    else
19:      return error

```

There are three changes to the PARSESTEP procedure in Algorithm 2. Firstly, the assignment to the variable *current-character* is removed, because we use the incremental input stack to get the lookahead instead. Secondly, the variable *multiple-states* is set to **true** if there is more than one active parse stack. Finally, we call the CHECKUPDATES procedure of Algorithm 3 before we start processing the *for-actor* stacks.

Algorithm 2 The PARSESTEP procedure of the ISGLR algorithm.

```

1: procedure PARSESTEP()
2:   multiple-states  $\leftarrow$  |active-stacks| > 1
3:   CHECKUPDATES()
4:   global for-actor  $\leftarrow$  active-stacks                    $\triangleright$  list of parse stacks
5:   global for-actor-delayed  $\leftarrow \emptyset$                     $\triangleright$  priority queue of parse stacks
6:   global for-shifter  $\leftarrow \emptyset$                         $\triangleright$  list of  $\langle$ parse stack, parse state $\rangle$  pairs
7:   while for-actor  $\neq \emptyset \wedge$  for-actor-delayed  $\neq \emptyset$  do
8:     if for-actor =  $\emptyset$  then
9:       for-actor  $\leftarrow$  {pop highest-priority stack in for-actor-delayed}
10:    for all st  $\in$  for-actor do
11:      if st has a link that is not rejected then
12:        ACTOR(st)
13:    SHIFTER()

```

The CHECKUPDATES procedure in Algorithm 3 checks whether the current lookahead contains any of the changes that were calculated by the diff. Firstly, it checks whether the current position of the parser is the start of a deletion. If this is the case, we pop all parse nodes that fall within the range of the deletion from the input stack. If the node at the top of the input stack covers the deletion only partially, we break it down and continue the loop for its children. After the deletion phase, we push all characters that are inserted by the change to the input stack. Secondly, in the case that the parse node on top of the input stack overlaps with any of the changes calculated by the diff, we break it down until this is no longer the case.

Algorithm 3 The CHECKUPDATES procedure of the ISGLR algorithm.

```

1: procedure CHECKUPDATES()
2:   if  $\exists update \in updates$  : current position = start position of deleted text of update then
3:     deleted-end  $\leftarrow$  end position of deleted text of update
4:     while current position < deleted-end do
5:       if width of PEEK(input-stack) + current position > deleted-end then
6:         BREAKDOWN(input-stack)
7:       else
8:         POP(input-stack)
9:       while width of PEEK(input-stack) = 0 do
10:        POP(input-stack)  $\triangleright$  Also pop null-yield parse nodes at position deleted-end
11:        Push inserted characters of update to input-stack in reverse order
12:   while PEEK(input-stack) contains a change  $\wedge$  it is not a character node do
13:     BREAKDOWN(input-stack)

```

The ACTOR procedure in Algorithm 4 has a different way of getting the actions from the parse table. Because the symbol of the top node in the incremental input stack can be either a terminal or not, the parse table can be indexed in different ways. This has been moved to a separate function BREAKDOWNUNTILVALIDACTIONS which is shown in Algorithm 5. If that function needs to break down the current lookahead, any elements that we already added to *for-shifter* need to be updated. It is still valid to shift the new lookahead onto the parse stacks in *for-shifter* by construction of the parse table, but the stacks will go to a different state. Finally, if there are multiple possible actions, the *multiple-states* variable is set to **true**.

Algorithm 4 The ACTOR procedure of the ISGLR algorithm.

```

1: procedure ACTOR(st)
2:   s  $\leftarrow$  state of st
3:   original-lookahead  $\leftarrow$  PEEK(input-stack)
4:   actions  $\leftarrow$  BREAKDOWNUNTILVALIDACTIONS(s)
5:   if original-lookahead  $\neq$  PEEK(input-stack) then
6:     Update GOTO states in for-shifter
7:   if |actions| > 1 then
8:     multiple-states  $\leftarrow$  true
9:   for all a  $\in$  actions do
10:    switch a do
11:      case Accept
12:        accepting-stack  $\leftarrow$  st
13:      case Shift(s')
14:        for-shifter  $\leftarrow$   $\{\langle st, s' \rangle\} \cup$  for-shifter
15:      case Reduce(A  $\rightarrow$   $\alpha$ )
16:        DOREDUCTIONS(st, A  $\rightarrow$   $\alpha$ )

```

The `BREAKDOWNUNTILVALIDACTIONS` function in Algorithm 5 determines the possible actions for the parser based on the current top of the incremental input stack (called *lookahead*). We use the first character of the lookahead to index the action table of the parse table. If the lookahead is a character node, we don't enter the loop and immediately return the list of actions. If the lookahead is a parse node, we enter the loop to check if we can reuse it.

If the state matching test succeeds (see Section 3.3.1), we substitute the normal *Shift* action with a new one that uses the state from the goto table which matches the production rule of the lookahead. We also do an optimization here in the case that the only *Reduce* action would create a parse node without children and the leftmost descendant of the lookahead has the same production rule as this *Reduce* action. In this case, we can remove this action to avoid setting *multiple-states* to **true**, because this action will eventually result in the same parse node as the one that we want to shift.

We break down the lookahead if it is an irreusable parse node or when there are *Shift* actions. Note that at this point, we know that the state matching test failed, so if we can do different *Shift* actions based on the first character in the lookahead, this lookahead possibly needs to be parsed in a different way. If the lookahead is not irreusable and we only have *Reduce* actions, we can safely return the current list of actions, which saves us from breaking down the lookahead.

If the lookahead that we broke down had no children, this is similar to popping a node from the input stack, so we need to do two more things. Firstly, we check if the new lookahead contains any changes and break it down again until this is no longer the case because the next parse node on the input stack might span a part of the input that has changed. Secondly, we empty *for-shifter* because the parse node that these parse stacks want to shift no longer exists. These parse stacks will be recreated by *Reduce* actions with arity 0. Note that this does not interfere with the optimization of line 10, because that optimization only occurs when the parser is not in multiple states, meaning that *for-shifter* must be empty by definition in that case.

Algorithm 5 The `BREAKDOWNUNTILVALIDACTIONS` function of the ISGLR algorithm.

```

1: function BREAKDOWNUNTILVALIDACTIONS(s)
2:   lookahead  $\leftarrow$  PEEK(input-stack)
3:   actions  $\leftarrow$  parse-table[s, first character in lookahead]
4:   while lookahead is not a character node do
5:     if state of lookahead = s then
6:       result  $\leftarrow$  Reduce actions in actions
7:       if multiple-states = false  $\wedge$  |result| = 1 then
8:         Reduce( $A \rightarrow \alpha$ )  $\leftarrow$  only action in result
9:         if  $|\alpha| = 0 \wedge$  leftmost descendant of lookahead has production  $A \rightarrow \alpha$  then
10:          result  $\leftarrow$   $\emptyset$   $\triangleright$  Prevents setting multiple-states to true
11:         add Shift(GOTO(s, production rule of lookahead)) to result
12:       return result
13:     if lookahead is irreusable  $\vee \exists$  Shift( $\_$ )  $\in$  actions then
14:       BREAKDOWN(input-stack)
15:     if lookahead does not have children then
16:       while PEEK(input-stack) contains a change do
17:         BREAKDOWN(input-stack)
18:       for-shifter  $\leftarrow$   $\emptyset$ 
19:       lookahead  $\leftarrow$  PEEK(input-stack)
20:     else
21:       break
22:   return actions

```

In `DoREDUCTIONS` and `DoLIMITEDREDUCTIONS` (Algorithms 6 and 8), the parser records that it is in multiple states when this reduction leads to more than one `GOTO` state. This can occur when multiple parse stacks have previously been merged into a single stack, and the multiple valid paths would make the `REDUCER` procedure add multiple new stacks to *for-actor*.

Algorithm 6 The `DoREDUCTIONS` procedure of the ISGLR algorithm.

```

1: procedure DoREDUCTIONS( $st, A \rightarrow \alpha$ )
2:   if there are multiple valid4 reduction paths, leading to different GOTO states then
3:      $multiple\text{-}states \leftarrow \mathbf{true}$ 
4:   for all valid paths from  $st$  to  $st'$  of length  $|\alpha|$  do
5:      $kids \leftarrow$  the parse nodes on the links of the path from  $st$  to  $st'$ 
6:     REDUCER( $st', \text{GOTO}(\text{state of } st', A \rightarrow \alpha), A \rightarrow \alpha, kids$ )

```

Algorithm 7 The `REDUCER` procedure of the ISGLR algorithm.

```

1: procedure REDUCER( $st, s, A \rightarrow \alpha, kids$ )
2:    $current\text{-}state \leftarrow \emptyset$  if  $multiple\text{-}states$ , else state of  $st$ 
3:    $rule\text{-}node \leftarrow$  new rule node with production  $A \rightarrow \alpha$ , child parse nodes  $kids$ 
4:   if  $\exists st' \in active\text{-}stacks : s = \text{state of } st'$  then
5:     if  $\exists$  a direct link  $l$  from  $st'$  to  $st$  then
6:        $symbol\text{-}node \leftarrow$  the parse node at  $l$ 
7:       add  $rule\text{-}node$  to the derivations of  $symbol\text{-}node$ 
8:       state of  $symbol\text{-}node \leftarrow \emptyset$ 
9:       if  $A \rightarrow \alpha$  is a reject production then
10:        mark link  $l$  as rejected
11:     else
12:        $symbol\text{-}node \leftarrow$  new symbol node for symbol  $A$  with  $rule\text{-}node$  as first derivation
13:       state of  $symbol\text{-}node \leftarrow current\text{-}state$ 
14:       add link  $l$  from  $st'$  to  $st$  with parse node  $symbol\text{-}node$ 
15:       if  $A \rightarrow \alpha$  is a reject production then
16:        mark link  $l$  as rejected
17:       for all  $st'' \in active\text{-}stacks$  do
18:         if  $st''$  has a link that is not rejected  $\wedge st'' \notin for\text{-}actor \cup for\text{-}actor\text{-}delayed$  then
19:           for all  $Reduce(B \rightarrow \beta) \in$ 
20:              $parse\text{-}table[\text{state of } st'', \text{first character in } PEEK(input\text{-}stack)]$  do
21:               DoLIMITEDREDUCTIONS( $st'', B \rightarrow \beta, l$ )
22:       else
23:          $st' \leftarrow$  new stack with state  $s$ 
24:          $symbol\text{-}node \leftarrow$  new symbol node for symbol  $A$  with  $rule\text{-}node$  as the first derivation
25:         state of  $symbol\text{-}node \leftarrow current\text{-}state$ 
26:         add link  $l$  from  $st'$  to  $st$  with parse node  $symbol\text{-}node$ 
27:          $active\text{-}stacks \leftarrow \{st'\} \cup active\text{-}stacks$ 
28:         if state of  $st'$  is rejectable then
29:            $for\text{-}actor\text{-}delayed \leftarrow \{st'\} \cup for\text{-}actor\text{-}delayed$   $\triangleright$  Priority unknown (Visser 1997)
30:         else
31:            $for\text{-}actor \leftarrow \{st'\} \cup for\text{-}actor$ 
32:         if  $A \rightarrow \alpha$  is a reject production rule then
33:           mark link  $l$  as rejected

```

⁴These “valid” paths are those that do not include rejected links. Neither Visser (1997) nor Denkers (2018) mention this in the SGLR parsing algorithm, but Denkers added this in the JSGLR2 implementation to fix a reported issue: <https://github.com/metaborg/jsglr/commit/5395a5870d51fd08578016fbbd0a756072799ed7>

The REDUCER procedure in Algorithm 7 has one change: we save the parse state of the current parse stack in the newly created *rule-node* (or no state if the parser is in multiple states, to mark it as irreusable).

Algorithm 8 The DOLIMITEDREDUCTIONS procedure of the ISGLR algorithm.

```

1: procedure DOLIMITEDREDUCTIONS( $st, A \rightarrow \alpha, l$ )
2:   if there are multiple valid5 reduction paths, leading to different GOTO states then
3:      $multiple\text{-}states \leftarrow \mathbf{true}$ 
4:   for all valid paths from  $st$  to  $st'$  of length  $|\alpha|$  going through  $l$  do
5:      $kids \leftarrow$  the parse nodes on the links of the path from  $st$  to  $st'$ 
6:     REDUCER( $st', \text{GOTO}(\text{state of } st', A \rightarrow \alpha), A \rightarrow \alpha, kids$ )

```

The SHIFTER procedure in Algorithm 9 has one main change. It no longer needs to create a new character node for the current character, but instead, we shift the character node or parse node that is at the top of the input stack.

Algorithm 9 The SHIFTER procedure of the ISGLR algorithm.

```

1: procedure SHIFTER()
2:    $active\text{-}stacks \leftarrow \emptyset$ 
3:    $lookahead \leftarrow \text{POP}(\text{input}\text{-}stack)$ 
4:   for all  $\langle s, st' \rangle \in \text{for}\text{-}shifter$  do
5:     if  $\exists st' \in active\text{-}stacks : s = \text{state of } st'$  then
6:       add a link from  $st'$  to  $st$  with parse node  $lookahead$ 
7:     else
8:        $st' \leftarrow$  new stack with state  $s$ 
9:       add a link from  $st'$  to  $st$  with parse node  $lookahead$ 
10:     $active\text{-}stacks \leftarrow \{st'\} \cup active\text{-}stacks$ 

```

3.5 Implementation in Modular Architecture

In this section, we describe the implementation of the ISGLR parsing algorithm in the modular JSGLR2 implementation by Denkers (2018) (see Section 2.4.3). Figure 3.12 shows the changes to the parsing pipeline of JSGLR2. This figure also shows the Imploder and Tokenizer components, which will be discussed in detail in Chapter 4.

The implementation of the ISGLR parser in the modular architecture of JSGLR2 requires extensions for several data structures (input stack, parse state, and parse forest) and code components (parser, reduce manager, and parse forest manager). We discuss each of these in the paragraphs below, as well as how we manage the caching of previous results between incremental parses.

Input Stack The ISGLR parser requires a specialization of the input stream because it consumes a stream of parse nodes instead of a stream of characters. Also, the input stream should allow breaking down parse nodes. Because of this, the implementation of this component can be better described as an input *stack*, rather than a stream.

The implementation of the incremental input stack also handles the processing of changes as shown in the CHECKUPDATES procedure in Algorithm 3. This encapsulation simplifies the other parsing procedures since they can always assume that the top of the input stack does not contain any changes.

⁵See footnote 4 in DoREDUCTIONS.

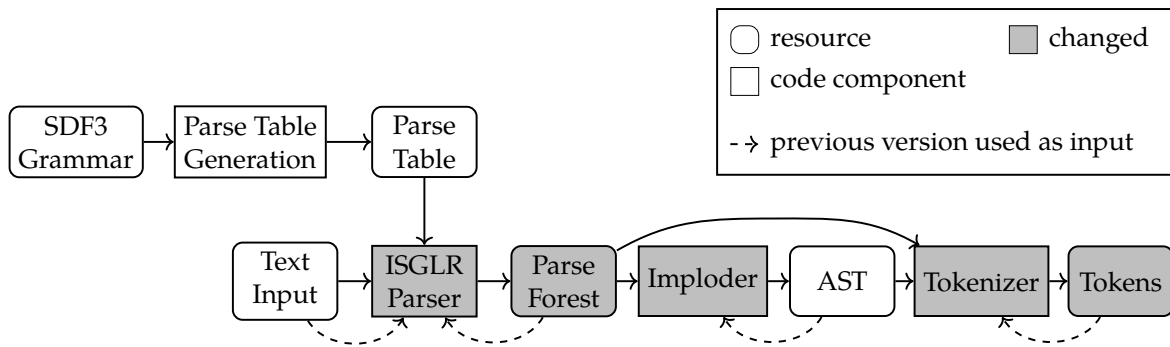


Figure 3.12: The parsing pipeline in Spoofox, updated from Figure 2.10 for incremental parsing. The top row of the pipeline is executed during language development, while the bottom row is executed every time a file is parsed. The dashed arrows indicate that a previous version of this resource is required as input to the incremental components.

Parse State The `ParseState` class for the ISGLR parser requires two modifications. Firstly, the basic input stream is replaced with the incremental input stack. Secondly, it needs to store the *multiple-states* global variable, which determines whether newly created parse nodes are irreusable or not.

Parse Forest (Manager) The `ParseForest` class and its subclasses have two modifications for the ISGLR parser. Firstly, each `ParseNode` stores the parse state that the parser was in during the creation, or no parse state if the parser was in multiple states, making the parse node irreusable. Secondly, we store the *width* of each `ParseForest`, i.e., the number of characters in its yield, to optimize the processing of updates and the imploding and tokenization steps. Because of these extensions to the `ParseForest`, the ISGLR parser also requires a specialized `ParseForestManager`, which makes sure that all the required data is passed to the constructor of the `ParseForest` classes.

Parser and Reduce Manager The changes for the ISGLR parsing algorithm with respect to SGLR, as indicated in Section 3.4, have been implemented in an extension of the `Parser` class. At the start of the `PARSE` procedure, it instantiates the new data structures. In the `ACTOR` procedure, getting actions from the parse table is extended to work if the current top of the input stack is a parse node instead of a character. Also, this procedure triggers the breakdown of the parse node that is currently on top of the input stack if it cannot reuse this parse node or if it finds no applicable actions in the parse table. Keeping track of the *multiple-states* variable requires changes in multiple places: at the start of the `PARSESYMBOL` procedure, after fetching the actions in the `ACTOR` procedure, and after calculating the number of valid reduction paths in the `Do(LIMITED)PRODUCTIONS` procedures (in the `ReduceManager` class).

Cache We set up the ISGLR parsing algorithm and the `Imploder` and `Tokenizer` components such that they take the previous result as input, i.e., they are agnostic of how these results are cached. Users of the JSGLR2 implementation are free to determine their caching strategy, but we also implemented a default caching strategy that can be used. This strategy maintains four `HashMap`s, one for each resource that needs to be cached (see the dashed arrows in Figure 3.12). These `HashMap`s are indexed using a file name, combined with some metadata of the parsing request. They are only updated when the parser produces a valid result, i.e., this caching strategy ignores parse failures.

This page is intentionally left blank.

Chapter 4

Incremental Post-Processing

After parsing, the JSGLR2 implementation performs two post-processing tasks, as shown in Section 2.4.3 and Figure 2.10, to simplify the integration of the parser into an editor. These tasks can also be performed incrementally, making use of the result of the Incremental Scannerless Generalized LR (ISGLR) parser, as shown in Section 3.5 and Figure 3.12. This chapter describes these tasks and shows how we made them incremental.

First, the imploder (Section 4.1) reduces the parse forest from a Concrete Syntax Tree (CST) into an Abstract Syntax Tree (AST). The AST only contains elements that are useful for further processing, discarding any layout and redundant lexical elements. The AST is used for type checking and code transformations. Because the imploding of a parse node does not require information about its context, the incremental imploding algorithm can directly reuse previous imploding results for unchanged parse nodes.

Second, the tokenizer (Section 4.2) transforms the parse forest to a list of tokens and attaches these tokens to the AST. It uses the production rule that was used to create a parse node to determine the type of the token. The tokens are used for syntax highlighting and error reporting. Similar to imploding, the incremental tokenization algorithm can reuse previous tokenization results for unchanged parse nodes. However, instead of storing absolute positions in tokens, we create a tree-shaped data structure that allows calculating the absolute positions on the fly.

4.1 Imploding

The parse forest that the parser produces is a Concrete Syntax Tree (CST), containing parse nodes for all kinds of production rules of the context-free grammar. However, many of these parse nodes are not necessary for further processing steps of the parse result, such as transformation and code generation in Stratego (Bravenboer et al. 2008; MetaBorg 2016) or type checking in Statix (van Antwerpen et al. 2018; MetaBorg 2016). For example, layout and literals do not add any semantic meaning to the program. Therefore, these elements are discarded when the imploder transforms the CST into an AST.

As an example, see the imploded AST of the parse tree in Figure 3.2 in Figure 4.1. In the CST of Figure 4.1(a), all parse nodes that are not needed to build the AST are left out. The remaining relevant nodes can be divided into three categories: those representing lists (node `4`), those containing constructors (nodes `5`, `19`, `20`, `37`), and characters that are not part of literals or layout (nodes `10`, `11`, `25`, `26`, `42`, `43`). As shown in Figure 4.1(b), the list nodes are converted into lists in the AST using square brackets (`[...]`), the parse nodes with constructors are converted into applications of their children, and the characters are imploded into strings. This last step is specific to scannerless parsing since parsing techniques that require a lexer would already have combined separate characters into larger tokens.

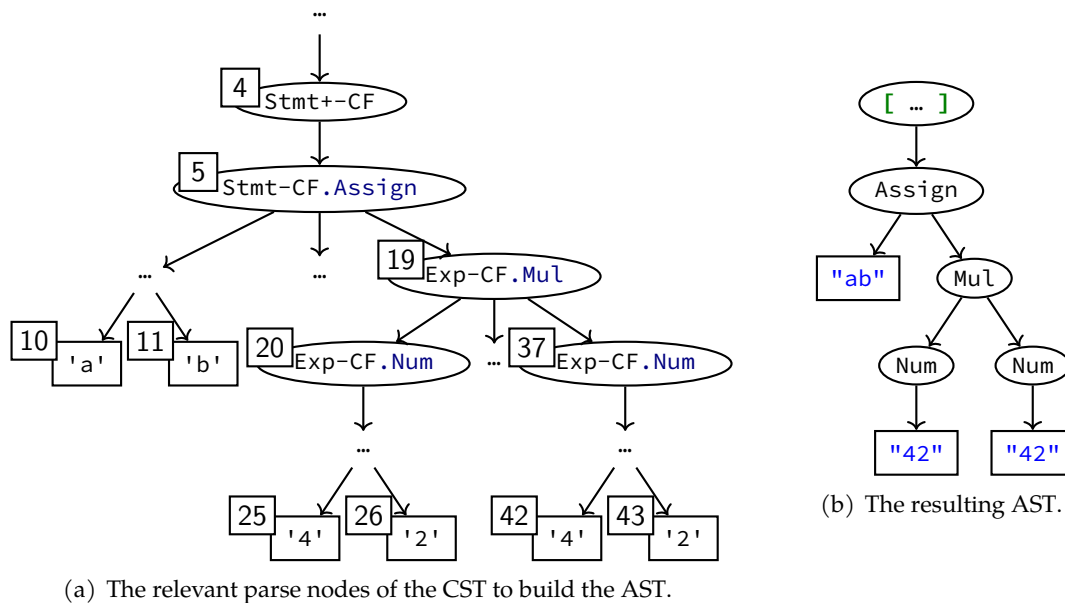


Figure 4.1: The imploded AST corresponding to the parse tree of Figure 3.2. In textual form, this AST would be written as `[Assign("ab", Mul(Num("42"), Num("42")))]`.

4.1.1 Imploding Algorithm

The current, non-incremental imploder in JSGLR2 works as shown in Algorithms 10 to 12.

The `IMPLODE` function of Algorithm 10 will decide how to implode the given *parse-tree* based on its grammar production. If the production is context-free, the imploder will recursively implode the children of *parse-tree*. In the case that *parse-tree* is an ambiguous parse node, the imploder will do this for all alternatives. If the production is not context-free, the imploder creates a leaf of the AST. This leaf is empty in the case that the production is a literal or layout production.

Algorithm 10 The `IMPLODE` function of JSGLR2.

```

1: function IMPLODE(parse-tree)
2:   production ← production of parse-tree
3:   if production is context-free then                                ▷ i.e., production contains -CF
4:     if parse-tree is ambiguous then                                ▷ i.e., has multiple derivations
5:       if production is a list then                                  ▷ i.e., has a + or * operator
6:         Reorder derivations of parse-tree so that ambiguities are pulled to the top
7:         alternatives ← empty list
8:         for all derivation ∈ derivations of parse-tree do
9:           Add IMPLODECHILDREN(production,
10:            FLATTENLISTS(production, child nodes of derivation)) to alternatives
11:        return amb(alternatives)
12:     else
13:       return IMPLODECHILDREN(production,
14:        FLATTENLISTS(production, child nodes of the only derivation of parse-tree))
15:   else
16:     if production is literal or layout then
17:       return ∅
18:     else
19:       return A new string constructed from all characters in parse-tree1

```

The `IMPLODECHILDREN` function of Algorithm 11 implodes all children of a parse node. All resulting ASTs that are not empty are added to the list of *child-asts*. Then, based on the production rule of the parent node, one of the possible types of AST nodes is created: this can be a constructor application, a list, an optional, or a tuple; or, if none of these cases applies, the production must be an injection and the only element of *child-asts* is returned.

Algorithm 11 The `IMPLODECHILDREN` function of JSGLR2.

```

1: function IMPLODECHILDREN(parent-production, parse-trees)
2:   child-asts  $\leftarrow$  empty list
3:   for all parse-tree  $\in$  parse-trees do
4:     child-ast  $\leftarrow$  IMPLODE(parse-tree)
5:     if child-ast  $\neq$   $\emptyset$  then
6:       Add child-ast to child-asts
7:   if parent-production has Constructor then
8:     return Constructor(child-asts)
9:   else if parent-production is a list then  $\triangleright$  i.e., has a + or * operator
10:    return [child-asts]  $\triangleright$  Construct a list from the child ASTs
11:   else if parent-production is an optional then  $\triangleright$  i.e., has a ? operator
12:     if  $|child-asts| = 0$  then
13:       return None()
14:     else
15:       return Some(child-asts)
16:   else if  $|child-asts| = 1$  then
17:     return the only element of child-asts
18:   else
19:     return (child-asts)  $\triangleright$  Construct a (possibly empty) tuple from the child ASTs

```

Finally, the `FLATTENLISTS` function of Algorithm 12 makes sure that the left-recursive lists in the parse tree are flattened out. This function will remove all parse nodes that represent a list production (in SDF3, this is a production rule containing a + or *) and return a list of all children of the removed parse nodes. Figure 4.2 shows an example of how this works. If the function receives parse nodes that did not belong to a list production, the original list of parse nodes is simply returned.

Algorithm 12 The `FLATTENLISTS` function of JSGLR2.

```

1: function FLATTENLISTS(parent-production, parse-trees)
2:   if parent-production is a list then  $\triangleright$  i.e., has a + or * operator
3:     flattened-list  $\leftarrow$  empty list
4:     for all parse-tree  $\in$  parse-trees do
5:       if parse-tree is ambiguous then  $\triangleright$  i.e., has multiple derivations
6:         Add parse-tree to flattened-list
7:       else
8:         sublist  $\leftarrow$  FLATTENLISTS(production of parse-tree, children of parse-tree)
9:         Add all items in sublist to flattened-list
10:    return flattened-list
11:   else
12:     return parse-trees

```

¹In the implementation of the imploder in Java, this string is taken as a substring from the original input string, using an *offset* that is incremented based on the widths of processed subtrees. This detail is left out to simplify the pseudocode.

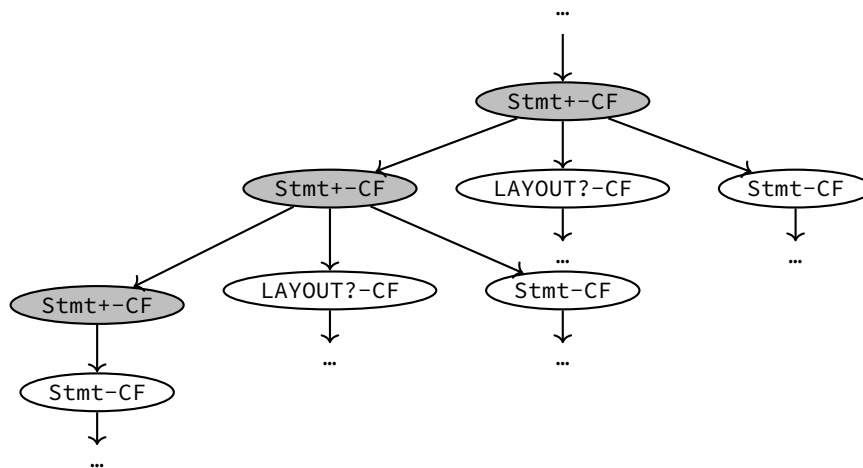


Figure 4.2: An example of the `FLATTENLISTS` function of Algorithm 12. This function takes the parse nodes that represent list productions (the grey nodes) and returns a list containing all their children (the white nodes). In this example, this list thus contains all `Stmt-CF` nodes, interleaved with the `LAYOUT?-CF` nodes.

4.1.2 Incremental Imploding

The incremental imploding algorithm builds upon the non-incremental algorithm outlined in Section 4.1.1. We make two observations that help in the design of the incremental imploding algorithm. Firstly, the result of Algorithm 10 does not depend on the ancestors or siblings of the parse node that is given as input. Secondly, we make use of the fact that parse nodes are immutable, meaning that their descendants cannot change. Therefore, if the imploder receives a parse node as input that has already been imploded before, the previous result can be reused.

The implementation of the incremental imploding algorithm is shown in Algorithm 13. This algorithm builds up a cache, using a dictionary that maps from CST nodes to AST nodes. If the input to the algorithm is already in this dictionary, we can directly return the stored result from it. Else, we call the `IMPLODE` algorithm from Algorithm 10 and store the result of that in the dictionary before returning it.

Algorithm 13 The `IMPLODE` function of incremental JSGLR2, extending the `IMPLODE` function of Algorithm 10 (referenced using `super.IMPLODE`).

```

1: cache ← dictionary from CST nodes to AST nodes
2: function IMPLODE(parse-tree)
3:   if parse-tree ∉ cache then
4:     cache[parse-tree] ← super.IMPLODE(parse-tree)
5:   return cache[parse-tree]

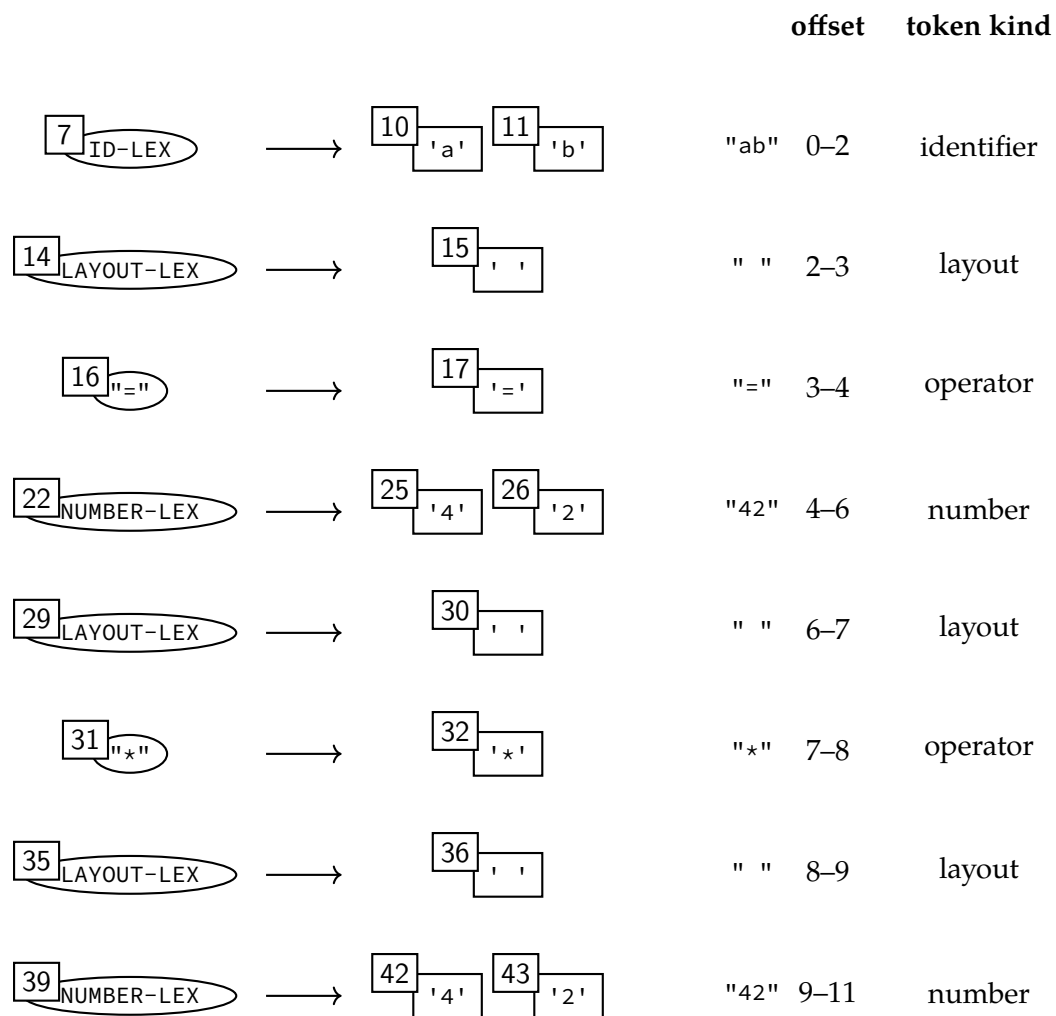
```

There are two implementation details worth noting about this algorithm. Firstly, because of how Java subclassing and dynamic dispatch works, this new `IMPLODE` function is executed when `IMPLODECHILDREN` recursively calls `IMPLODE`, instead of the original function. Secondly, we implemented the *cache* dictionary using a `WeakHashMap`,² allowing the Java garbage collector to delete entries from the dictionary from which the parse nodes no longer exist. Therefore, this *cache* is not exposed in the interface of the `IMPLODE` function.

²<https://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html>

4.2 Tokenization

Most IDEs (Spoofox included) use syntax highlighting to make the code they display easier to read, giving different types of elements in the code different colours. An IDE generates these colours using a list of tokens generated by the tokenizer. The tokenizer creates this list based on the information stored in the CST and the AST, as shown in Figure 4.3. The tokens are similar to those that a lexer would create before parsing token-level grammars, in the sense that they contain information about their location in the input and their type. However, creating the tokens *after* parsing has the advantage that the token types can be calculated based on their context, making this calculation more precise, especially when grammars of multiple languages are composed together.



(a) The parse nodes of the CST that the tokenizer generates tokens from.

(b) The resulting tokens.

Figure 4.3: The tokens that are generated from the parse tree of Figure 3.2.

An IDE can also use the tokens to generate the locations of error messages. Other parts of the IDE (e.g., the type checker) can report an error or warning by referencing an AST node instead of having to look up the exact position themselves. To accommodate this, the tokenizer attaches two tokens to each node in the AST, indicating the leftmost and rightmost position of the AST in the source code.

4.2.1 Tokenizer Algorithm

The current, non-incremental tokenizer in JSGLR2 works as shown in Algorithms 14 and 15.

The `TOKENIZE` function of Algorithm 14 generates a list of tokens based on the input parse tree. Just like the imploder, this algorithm recursively generates tokens for all context-free parse nodes.

To make sure that all AST nodes get a left and right token, so-called *empty tokens* are created for AST nodes that do not correspond to any characters in the input (see line 12 in Algorithm 14 and lines 4–5 in Algorithm 15). These empty tokens span a width of 0 characters but still contain location and production information.

The tokenizer also keeps track of the positions of tokens. Positions of characters are composed of three integers: *offset*, the zero-based index of a character in the entire input; *line*, the one-based index of the line that the character appears on; and *column*, the one-based column index of the character on its line.³ The begin position of a token is the position of the first character of this token, while its end position is the position *after* the last character of the token. This means that for empty tokens, their begin position is equal to their end position.

The initial value for the *start-position* parameter is a position with offset 0, line 1, and column 1. While iterating over the child trees, the position is advanced based on the position ranges of the subtrees (see line 11 in Algorithm 14 and line 2 in Algorithm 15).

Algorithm 14 The `TOKENIZE` function of JSGLR2.

```

1: function TOKENIZE(parse-tree, start-position)
2:   if production of parse-tree is context-free then           ▷ i.e., production contains -CF
3:     all-tokens ← ∅                                           ▷ list of tokens
4:     for all derivation ∈ derivations of parse-tree do
5:       tokens ← ∅                                           ▷ list of tokens
6:       pivot-position ← start-position
7:       for all child-tree ∈ child nodes of derivation do
8:         sub-tokens ← TOKENIZE(child-tree, pivot-position)
9:         Add all tokens in sub-tokens to tokens
10:        if sub-tokens ≠ ∅ then
11:          pivot-position ← end position of the last token in sub-tokens
12:        if tokens = ∅ then           ▷ This implies that derivation spans 0 characters
13:          token ← CREATE_TOKEN(parse-tree, pivot-position)
14:          Set token as left and right token of AST of derivation
15:          Add token to tokens
16:        else
17:          Set first token in tokens as left token of AST of derivation
18:          Set last token in tokens as right token of AST of derivation
19:          Add all tokens in tokens to all-tokens
20:        return all-tokens
21:   else
22:     if parse-tree spans 0 characters ∧ parse-tree does not have an AST node then
23:       return ∅
24:     else
25:       token ← CREATE_TOKEN(parse-tree, begin-position)
26:       Set token as left and right token of AST of parse-tree
27:       return [token]

```

³The pseudocode does not show this level of detail to make it more readable. We assume that the + operator can add two position(-range)s together.

Algorithm 15 The `CREATE_TOKEN` function of JSGLR2.

```

1: function CREATE_TOKEN(parse-tree, start-position)
2:   end-position  $\leftarrow$  start-position + position range of parse-tree
3:   Use begin-position and end-position when creating a token in this function
4:   if parse-tree spans 0 characters then
5:     return empty token
6:   production  $\leftarrow$  production of parse-tree
7:   if production is LAYOUT then
8:     return layout token
9:   if production is a literal then
10:    if production is an operator literal then
11:      return operator token
12:    return keyword token
13:  if production represents a string then  $\triangleright$  i.e., it contains the character class [\ ]4
14:    return string token
15:  if production represents a number then  $\triangleright$  i.e., it contains the character class [0-9]
16:    return number token
17:  return identifier token

```

4.2.2 Incremental Tokenization

There is one main challenge related to incremental tokenization: storing the absolute positions of tokens would require updates in many tokens for even a small update in the input. As an example, when the input changes from “ab =42 * 42” to “ans =42 * 42”, an incremental tokenizer does not only need to replace the first token (representing “ab”) by a new token (representing “ans”), but it would also need to increment the positions of all following tokens by one.

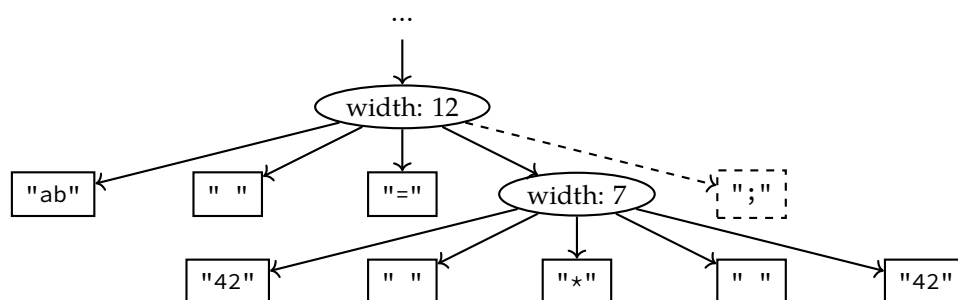


Figure 4.4: An example token tree, generated from the parse tree of Figure 3.2. An imaginary semicolon has been added for illustration purposes.

To solve this challenge, we turn the list of tokens into a tree-shaped data structure, with the tokens as leaf nodes. In this tree, every node stores its width in terms of the number of characters, the number of lines, and the number of columns in the last line. Absolute positions of tokens can then be calculated on the fly by summing up the widths of all siblings to the left along the spine of a token to the root. An example of such a token tree is shown in Figure 4.4. When calculating the start offset of the token “*”, we sum up the widths of its left siblings ($2 + 1 = 3$) and the left siblings of its parent node ($2 + 1 + 1 = 4$), arriving at

⁴If the right-hand side of the production rule contains the space character (and it is not layout), it must be enquoted in some way, so it represents a string in its language.

a start offset of 7. When calculating the start offset of the imaginary ";" token, summing up the widths of its left siblings gives a start offset of $2 + 1 + 1 + 7 = 11$.

Using a tree-shaped data structure to store the tokens has the advantage that we can make the tokenizer incremental similar to the incremental `IMPLODE` function, as shown in the `TREETOKENIZE` function in Algorithm 16. If the tokenizer has already processed a parse tree, it can reuse that result from the cache. Note that the same implementation details apply as for the incremental `IMPLODE` function in Algorithm 13.

Algorithm 16 The `TREETOKENIZE` function of incremental JSGLR2, extending the `TREETOKENIZE` function of Algorithm 17 (referenced using `super.TREETOKENIZE`).

```

1: cache ← dictionary from CST nodes to token tree nodes
2: function TREETOKENIZE(parse-tree)
3:   if parse-tree ∉ cache then
4:     cache[parse-tree] ← super.TREETOKENIZE(parse-tree)
5:   return cache[parse-tree]

```

The non-incremental `TREETOKENIZE` function in Algorithm 17 works roughly the same as the non-incremental `TOKENIZE` function in Algorithm 14. The main difference is that the `TREETOKENIZE` function does not return a list of tokens but a token tree node. Each token tree node stores its width (equal to the sum of the widths of its children), its left- and rightmost tokens, and a reference to its parent.

Algorithm 17 The `TREETOKENIZE` function of JSGLR2.

```

1: function TREETOKENIZE(parse-tree)
2:   if production of parse-tree is context-free then           ▷ i.e., production contains -CF
3:     all-token-trees ← ∅                                       ▷ list of token trees
4:     for all derivation ∈ derivations of parse-tree do
5:       token-subtrees ← ∅                                       ▷ list of token trees
6:       for all child-tree ∈ child nodes of derivation do
7:         token-subtree ← TREETOKENIZE(child-tree)
8:         if token-subtree ≠ ∅ then
9:           Add token-subtree to token-subtrees
10:      if token-subtrees = ∅ then           ▷ This implies that derivation spans 0 characters
11:        token ← CREATETREETOKEN(parse-tree)
12:        Set token as left and right token of AST of derivation
13:        Add token to token-subtrees
14:      else
15:        Set first token in token-subtrees as left token of AST of derivation
16:        Set last token in token-subtrees as right token of AST of derivation
17:        token-tree ← new token tree node from token-subtrees
18:        Add token-tree to all-token-trees
19:      return new token tree node from all-token-trees
20:   else
21:     if parse-tree spans 0 characters ∧ parse-tree does not have an AST node then
22:       return ∅
23:     else
24:       token ← CREATETREETOKEN(parse-tree)
25:       Set token as left and right token of AST of parse-tree
26:       return new token tree node from token

```

The `CREATETREETOKEN` function in Algorithm 18 has one main difference compared to the creation of traditional tokens in the `CREATETOKEN` function in Algorithm 15: instead of storing a begin- and end-position, a token in the token tree only stores its width.⁵

Algorithm 18 The `CREATETREETOKEN` function of JSGLR2.

- 1: **function** `CREATETREETOKEN`(*parse-tree*)
 - 2: *width* \leftarrow position range of *parse-tree*
 - 3: Use *width* when creating a token in this function
 - 4: **return** a token with kind as calculated in `CREATETOKEN` (Algorithm 15, from line 4)
-

⁵Just like how the Java implementation of the imploder needs the original input string to do its calculations more efficiently (see footnote 1), the same goes for calculating the width of a tree token. Ironically, this calculation also requires knowing the absolute position of a token in its original input string, so the Java implementation of the tokenizer needs to keep track of this. However, since the tree tokens do not store their absolute position directly, this does not affect the incrementality of the algorithm.

This page is intentionally left blank.

Chapter 5

Performance Evaluation

We evaluate the ISGLR parser in two different ways: measurements and benchmarks. All evaluations use software repositories on GitHub¹ as input to the parser, making use of the recorded history of the files. We describe the evaluation setup and corpus in Section 5.1.

We measure statistics for certain events happening during parsing and the parse nodes in the resulting parse forest in Section 5.2. We find that irreusable parse nodes make up 27% of the average parse forest, which causes the majority of the breakdown events during incremental parsing. Languages like Java and WebDSL, which use fence characters like curly brackets and semicolons, have less irreusable parse nodes in their parse forests and have more parse node reuse than a language like SDF3, which does not have such fences. The majority of the irreusable parse nodes is not exposed on top of the input stack during a subsequent incremental parse, so on average, the ISGLR parser can reuse 99% of a previous parse tree.

In Sections 5.3 and 5.4, we evaluate the run time and memory performance of the ISGLR parser, respectively. We consider both the scenario of batch parses and the scenario of incremental parses. In batch mode, we compare the performance of the ISGLR parser with two existing JSGLR2 parser variants: the *Standard* parser and the *Elkhound* variant (see Section 2.4.3). The run time/memory performance of the ISGLR parser in batch mode is 24% slower/23% higher than the Standard parser and 60% slower/68% higher than the Elkhound variant. In incremental mode, for changes that are smaller than 1% of the input size on average, the run time performance of the ISGLR parser is 9× faster than the Standard parser in batch mode.

In Section 5.5, we discuss possible threats to the validity of this evaluation and to what extent we countered them.

5.1 Setup

We executed the evaluation on a server machine with two 32-core AMD EPYC processors with a base frequency of 2.3 GHz and 256 GB RAM provided by sixteen DDR4-2933 modules. The used operating system is Ubuntu 20.04 (kernel version 5.4.0-77), using OpenJDK version 1.8.0_275-b01 and Apache Maven 3.6.3.

We ran a series of evaluation scripts, written in Scala 2.13 unless noted otherwise, inside a Docker container² using Docker version 20.10.7. The scripts are configurable using a YAML³ file describing the list of languages and sources to evaluate the parser with.

¹<https://github.com/>

²The Docker container, including the used scripts, are publicly available on GitHub:
<https://github.com/metaborg/jsglr2evaluation>

³<https://yaml.org/>

We separated the evaluation scripts into the following steps:

Pull Languages Pull the latest version of the Spoofox languages used for evaluation from GitHub and build these projects using Maven.

Pull Sources Pull the latest versions of the sources in the evaluation corpus from GitHub. This only pulls versions from Git commits that have changes in source files with the file extension that belongs to the language, e.g., `.java` for the Java language.

Preprocessing Validate the evaluation corpus in several ways, see below.

Measurements Perform measurements while parsing and compare parse forests of subsequent versions, see below.

Time Benchmarks Execute benchmarks that measure the run time of the parsers, see below.

Memory Benchmarks Execute benchmarks that measure the memory usage and cache size of the parsers, see below.

Post-Processing Process the Comma-Separated Values (CSV) files that resulted from the various benchmarks, add extra data like file sizes and change sizes, and combine all this into larger CSV files.

Generate Figures Generate plots with Matplotlib,⁴ in Python 3.8.5.

Publish Generate a web page from the evaluation results and publish it to the website <https://www.spoofox.dev/jsglr2evaluation-site/>.

Preprocessing We validate the evaluation corpus in several ways.

For the batch scenario, we ensure the validity of inputs and parsers by parsing all inputs with all parser variants. Some files can be invalid due to, for example, a mismatch between language grammar versions and are automatically removed from the batch scenario corpus. We inspect those files manually to judge whether they are actually invalid or that the language grammar was incorrect. To validate consistency between the JSGLR2 variants, we check that all variants produce the same AST in batch mode.

For the incremental scenario, we validate that an incremental parse of a file has the same result as a batch parse of the same file. Naturally, this is only done for files that have a previous version in the evaluation corpus. If this validation fails, it would mean that the JSGLR parser is incorrect and the entire evaluation is aborted.

Measurements We performed measurements while parsing the input from the evaluation corpus. We counted the number of parse nodes in the resulting parse forests, recording their type (parse node or character node) and whether they are reusable or not.

We also compared parse forests between subsequent versions. During an incremental parse, we count how many parse nodes are reused or broken down. We also count the number of parse nodes that are *rebuilt*, i.e., that were broken down during parsing, but were recreated with the exact same children as in the parse forest of the previous version. Of the parse nodes that are broken down, we record the reason for the breakdown.

Time Benchmarks The benchmarks that measure the run time of the parsers make use of the Java Microbenchmark Harness (JMH),⁵ version 1.25.2. The setup phase of a JMH benchmark initializes the parse table and parser for the given language. Also, when doing a benchmark of an incremental parse, the setup phase parses the first version of the input to populate the cache.

⁴<https://matplotlib.org/>

⁵<https://github.com/openjdk/jmh>

JMH runs a predefined action in multiple so-called *iterations*, where each iteration runs the action as often as possible for 10 seconds. We run the time benchmarks using 10 *warmup* iterations and 10 regular iterations. These warmup iterations execute the action but do not measure its run time, to warm up the Java Virtual Machine (JVM).

We separated the time benchmarks into multiple executions of JMH, where each of these benchmark runs considers two consecutive versions of the evaluation corpus. In the setup phase of the benchmark, we save the results of the first version (parse forests, ASTs, tokens). We defined the benchmark action as a call to the parser with the second version as input, using the results of the first version for incrementality.

Memory Benchmarks We run two kinds of memory benchmarks: one that measures the *allocations* during parsing and one that calculates the *cache size* after a parser saves a result to the cache. For every language, the benchmark script sampled 100 files from the evaluation corpus to run the memory benchmarks with.

We measure memory allocations by instrumenting the JVM to count every object constructor called during parsing, using a custom allocation instrumenter library⁶ forked from Google. This library instruments all constructors of all loaded Java classes with a call to an observable. To this observable, we attach an observer that sums the sizes of all objects that are created during parsing.

We measure the cache size using the Java Object Layout library.⁷ This library can calculate the size of an object, including the size of all of its references, recursively. For every file in the benchmark, we subtract a measurement taken before parsing from a measurement taken after parsing to obtain the impact on the memory usage of the cache for that file.

5.1.1 Evaluation Corpus

Table 5.1 shows the languages and sources that we used to evaluate the performance of the ISGLR parser. We chose three languages with different characteristics: Java as a General-Purpose Language (GPL), SDF3 as Domain-Specific Language (DSL) that is used as a meta-language in Spoofox, and WebDSL as DSL that is used to generate websites. The grammars of these languages are publicly available on GitHub.

Table 5.1: Corpus used to evaluate the performance of the ISGLR parser. The numbers of files/lines and the (file/change) sizes are the average of all versions. The change sizes are equal to the number of removed characters plus the number of added characters.

Language	Source	Versions	Files	Lines	Size (B)	Mean file size (B)	Change size (B)
Java	StringUtils	16	1	9 635	396 297	396 297	850
	gson	16	204	37 163	1 265 001	6 189	988
	slf4j	16	238	26 632	893 078	3 756	8 606
WebDSL	builtin.app	10	1	3 403	98 147	98 147	650
	YellowGrass	16	53	6 019	166 767	3 171	1 168
	elib-utils	16	17	1 422	39 480	2 322	172
SDF3	NaBL	16	126	4 386	98 690	781	606
	DynSem	16	4	437	9 716	2 429	130
	FlowSpec	16	15	567	12 209	816	516
	Stratego	7	64	2 950	79 224	1 240	379
	WebDSL	16	26	3 252	86 032	3 308	114

⁶<https://github.com/mpsijm/simple-allocation-instrumenter>

⁷<https://github.com/openjdk/jol>

For two sources, we picked a single large file to investigate the effect of this on the performance of the parser. For Java, we chose the file `StringUtil.java`, a file of almost 400 kB from Apache’s Commons Lang library. For WebDSL, we chose the file `builtin.app`, a file of almost 100 kB from the WebDSL compiler project. For other sources, we picked entire repositories. All sources are publicly available on GitHub.

5.2 Measurements Results

In this section, we will discuss measurements about the parse nodes in the parse forest returned as result from the parser (Section 5.2.1) and the reasons for breaking down parse nodes (Section 5.2.2). Table 5.2 displays the most interesting results from the measurements, as an average of the measurements over all corpus sources of a language. See Appendix A.1 for the full measurement results.

Table 5.2: Measurements for parse nodes and breakdowns for the different languages in the evaluation corpus as shown in Table 5.1.

Language	Parse nodes (% of total nodes)				Breakdowns (% of total breakdowns)			
	Irre-usable	Reused	Broken down	Rebuilt	Contains Change	Irre-usable	No actions	Wrong state
Average	27.25%	99.12%	0.67%	0.48%	40.68%	56.23%	0.00%	3.08%
Java	19.08%	99.60%	0.17%	0.08%	51.35%	46.67%	0.00%	1.98%
WebDSL	21.32%	99.55%	0.28%	0.15%	51.35%	43.76%	0.00%	4.89%
SDF3	41.36%	98.21%	1.57%	1.21%	19.35%	78.28%	0.00%	2.38%

5.2.1 Parse Nodes

The left part of Table 5.2 shows four measurements as a percentage of the total number of parse nodes:

Irreusable is the percentage of parse nodes that is marked as irreusable because they were created while the parser was parsing non-deterministically.

Reused is the percentage of parse nodes that are reused during an incremental parse, i.e., parse nodes that the parser could shift directly together with all its descendent nodes.

Broken down is the percentage of parse nodes that the parser has to break down during an incremental parse.

Rebuilt is the percentage of parse nodes that are created during an incremental parse for the same production rule and with the same children as in the previous parse.

Note that the percentages “Reused” and “Broken down” do not fully sum up to 100% for any language, while intuitively, a parse node is either reused or broken down. However, the `CHECKUPDATES` procedure might skip larger parse nodes, as shown in line 8 of Algorithm 3. Any descendants of these skipped parse nodes are not counted in the “Broken down” percentage, because the parser does not actively break them down.

Java and WebDSL have over 99.5% parse node reuse, but SDF3 has only 98% parse node reuse. Similarly, parse forests for Java and WebDSL have around 20% irreusable parse nodes, while in parse forests for SDF3 they make up 41% of all parse nodes. After manually inspecting parse forests for the different languages, we found that the higher number of irreusable parse nodes for SDF3 can be explained by the lack of fencing characters in the language.

Languages like Java and WebDSL use curly brackets (`{}`) and semicolons (`;`) as fencing characters. Curly brackets indicate the nesting of parts of code, for example, the body of a function or the body of an `if`-statement. Similarly, semicolons terminate a statement within a block of code. However, in SDF3, code sections are bounded using keywords (like **context-free syntax**) and definitions of production rules do not have a terminating character. The parser will need to parse any word non-deterministically, as discussed in Section 3.2.2: a word can either be the next sort in the right-hand side of a production rule, the sort on the left-hand side of a new production rule, or a keyword that indicates the start of a new code section. Therefore, the parse nodes which correspond to code sections and production rules are marked as irreusable, which makes the ISGLR parser break down many of these parse nodes during an incremental parse when they end up at the top of the input stack. In languages that do use fencing characters, this does not happen, because the fences consist of a single character: when the parser encounters a fence, there is only one way of parsing the statement (block) that has this fence as the final character.

Of the parse nodes that are broken down, on average, 71% of them are rebuilt.⁸ This implies that the majority of the breakdowns are unnecessary in hindsight. However, because of the non-deterministic nature of scannerless parsing, these breakdowns are still necessary during parsing since the parser can only look one character ahead. Still, for changes that are smaller than 1% of the input size on average, we conclude that a reuse of 99% on average is acceptable for an incremental parser.

5.2.2 Breakdowns

The right part of Table 5.2 shows four measurements as a percentage of the total number of broken-down parse nodes:

Contains Change is the percentage of breakdowns that happen when that parse node contains a change as calculated by the diff.

Irreusable is the percentage of breakdowns that break down an irreusable parse node.

No actions is the percentage of breakdowns occurring when there are no valid actions in the parse table.

Wrong state is the percentage of breakdowns that break down a parse node because the state matching test failed, i.e., the parse state reference stored in the parse node is not the same as the current parse state.

The number of breakdowns caused by changed parse nodes is 41% on average. All other broken-down parse nodes are not located in the spine between the updated character nodes and the root of the parse forest. Ideally, an incremental parser would only break down parse nodes that are located in this spine, including the parse nodes that cannot be reused because the state matching test failed. In the evaluated corpus, another 3% of parse nodes fall in this latter category.

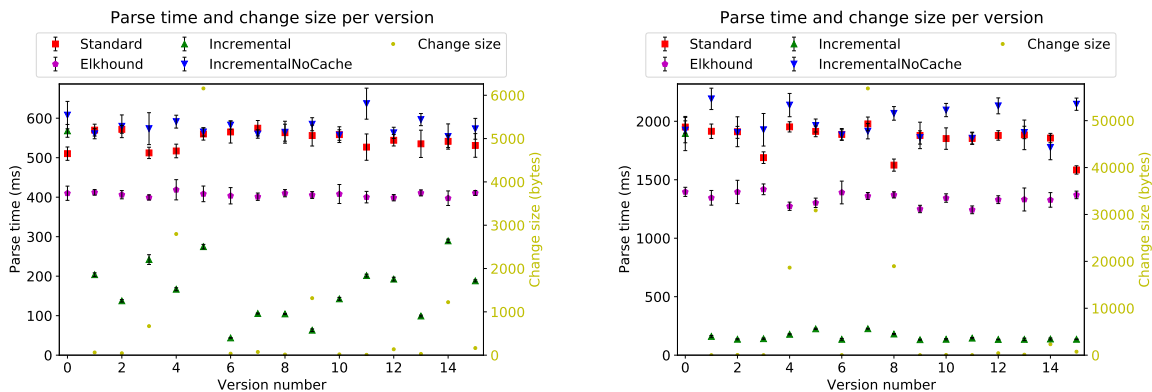
However, most of the breakdowns, 56% on average, are caused by irreusable parse nodes. This can be explained by the fact that a quarter of all parse nodes is irreusable, so many of them end up exposed at the top of the input stack, causing them to be broken down. This effect is stronger in the parsing of SDF3 than for the parsing of Java or WebDSL, for the same reasons as mentioned in Section 5.2.1.

None of the breakdowns is caused by the absence of valid actions in the parse table. This scenario does not occur in this evaluation, since we only consider input files that are valid according to the grammar of the language.

⁸This value is calculated from Table 5.2 by dividing the value in the “Rebuilt” column by the value in the “Broken down” column.

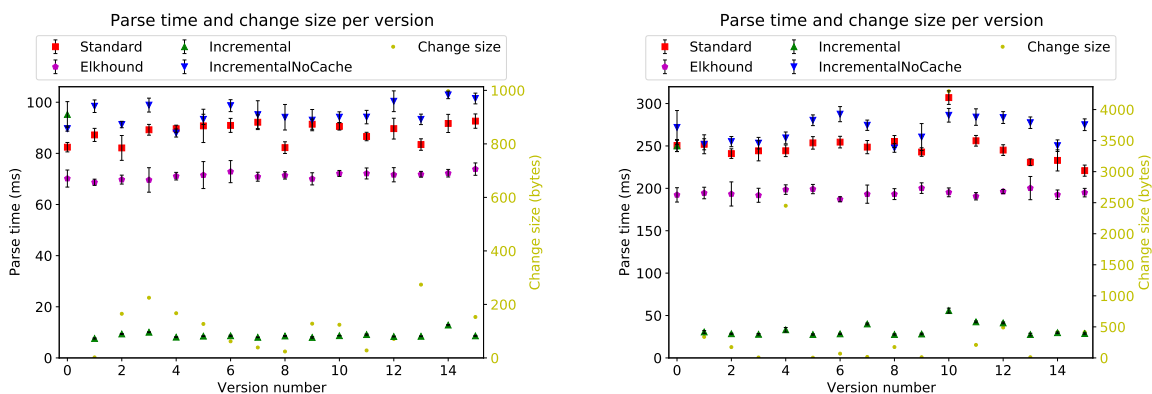
5.3 Time Benchmark Results

This section shows the performance results of the ISGLR parser in terms of run time, compared to other parser variants in JSGLR2 and Tree-sitter (Brunsfeld 2018). Tree-sitter is a parser based on the Incremental LR (ILR) parsing algorithm by Wagner and Graham (1998). We will describe Tree-sitter in more detail in Section 6.1.1.



(a) Parse times of `StringUtils.java` from Apache's commons-lang library.

(b) Parse times of the `slf4j` repository.



(c) Parse times of the `elib-utils` repository.

(d) Parse times of the `nabl` repository.

Figure 5.3: Parse times of the last 16 versions of several files and repositories from GitHub, excluding imploding and tokenization. The red squares represent the Standard JSGLR2 parser, the purple pentagons represent the Elkhound parser variant, the blue downward-pointing triangles represent the incremental parser in batch mode (i.e., without cache), and the green upward-pointing triangles represent the incremental parser in incremental mode. The yellow dots indicate the size of the changes in a particular version, which is the sum of the number of removed and inserted characters, on the second y-axis.

Table 5.4: Average parse times for the different languages in the evaluation corpus, as shown in Table 5.1, for three JSGLR2 parser variants. The values indicate the parse time in milliseconds, excluding imploding and tokenization.

Language	Standard	Elkhound	Incremental (no cache)	Incremental
Average	785.510	597.235	839.299	82.503
Java	1974.375	1497.341	2112.104	203.043
WebDSL	233.029	175.540	244.903	23.681
SDF3	149.127	118.824	160.888	20.784

All benchmark results in this section show the average run time performance over 10 benchmark iterations, where the error bars in the plots indicate the 99%-confidence interval. The full time benchmark results are shown in Appendix A.2.

We evaluate the run time performance separately with and without the imploding and tokenization steps, as shown in the bottom row of Figures 2.10 and 3.12. The Tree-sitter parser is only included in the second comparison, where for both parsers their full parsing pipeline is timed from start to end.

Parsing Only Figure 5.3 shows the parse times for four of the sources in the evaluation corpus, where the ISGLR parser is measured both in batch mode and in incremental mode. The parse times in this figure are measured for only parsing, i.e., excluding imploding and tokenization. The average parse times for the three languages of the evaluation corpus (Section 5.1.1) are shown in Table 5.4.

The overall result is that the ISGLR parser in incremental mode has a significant speedup compared to the Standard parser in batch mode of 9.5 \times , with changes that are smaller than 1% of the input size, on average. The ISGLR parser in batch mode is 7% slower than the Standard JSGLR2 parser variant, and 41% slower than the Elkhound variant.

Interestingly, Figure 5.3(a) shows some outliers where the ISGLR parser is significantly slower to parse incremental updates to `StringUtils.java` than for the other sources of the evaluation corpus. Remember from Section 5.1.1 that `StringUtils.java` is a Java file of almost 400 kB. Some of the slower incremental parsing times can be attributed to the size of the diff, like version 5. However, version 11 has only added eight characters with respect to version 10, yet the incremental parsing time is one of the longest for this figure. Specifically, these eight characters were added in four different places throughout the file. Further inspection using a Java profiler⁹ showed that executing the diff algorithm took 7 \times more time than the incremental parsing for this particular diff. We consider optimizing the diff algorithm out of scope for this thesis.

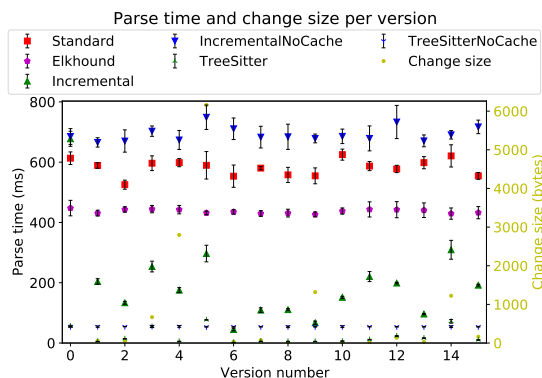
With Imploding And Tokenization Figure 5.5 shows the parse times for four of the sources in the evaluation corpus. The parse times of the JSGLR2 parser variants are measured for the full JSGLR2 parsing pipeline, which includes imploding and tokenization. The average parse times for the three languages of the evaluation corpus (Section 5.1.1) are shown in Table 5.6. The parse times for Tree-sitter (Brunsfeld 2018) are only measured for the Java language since no Tree-sitter grammars for WebDSL and SDF3 are available.

The overall result is that the ISGLR parser in incremental mode has a significant speedup compared to the Standard parser in batch mode of 9.0 \times , with changes that are smaller than 1% of the input size, on average. The ISGLR parser in batch mode is 24% slower than the Standard JSGLR2 parser variant, and 60% slower than the Elkhound variant. Comparing this with the results of Table 5.4, we see that the parsing phase takes up 97% and 95% of the full JSGLR parsing pipeline for the Standard and Elkhound parser variants, respectively. For the ISGLR parser, this percentage drops to 83% in batch mode and 91% in incremental mode. This indicates that the incremental imploding and tokenization algorithms have more overhead compared to the incremental parsing algorithm than the original algorithms.

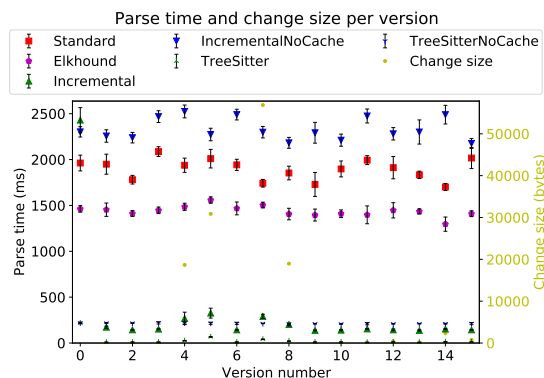
The ISGLR parser is roughly 11 \times slower than the Tree-sitter parser, both in batch mode and in incremental mode. It is almost impossible to conclude anything from this because Tree-sitter is written in the C programming language, while JSGLR2 is written in Java. Generally speaking, a program written in Java is slower than the same program written in C, but the slowdown factor varies heavily depending on the type of program (Fourment and Gillings 2008). However, the speedup of incremental parsing compared to batch parsing for Tree-sitter is similar, although slightly better (12 \times) compared to the ISGLR parser.

⁹<https://www.ej-technologies.com/products/jprofiler/overview.html>

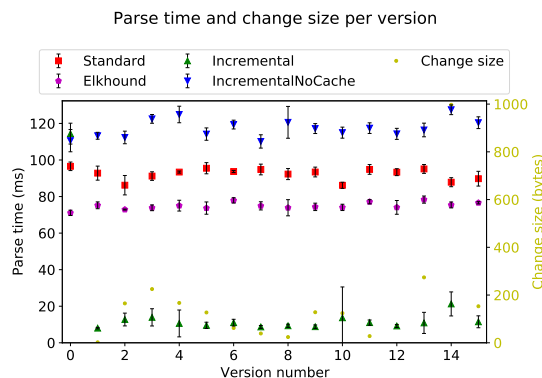
5. PERFORMANCE EVALUATION



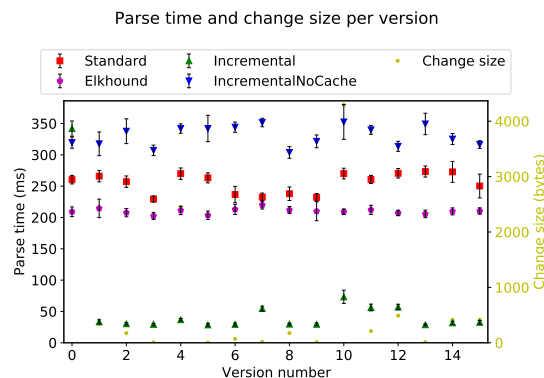
(a) Parse times of `StringUtils.java` from Apache's commons-lang library.



(b) Parse times of the `slf4j` repository.



(c) Parse times of the `elib-utils` repository.



(d) Parse times of the `nabl` repository.

Figure 5.5: Parse times of the last 16 versions of several files and repositories from GitHub, including imploding and tokenization. The red squares represent the Standard JSGLR2 parser, the purple pentagons represent the Elkhound parser variant, the blue downward-pointing triangles represent the ISGLR parser in batch mode (i.e., without cache), the green upward-pointing triangles represent the ISGLR parser in incremental mode, the blue downward-pointing three-pointed stars represent the Tree-sitter parser in batch mode, and the green upward-pointing three-pointed stars represent the Tree-sitter parser in incremental mode. The yellow dots indicate the size of the changes in a particular version, which is the sum of the number of removed and inserted characters, on the second y-axis.

Table 5.6: Average parse times for the different languages in the evaluation corpus, as shown in Table 5.1, for three JSGLR2 parser variants and Tree-sitter. The values indicate the parse time in milliseconds, including imploding and tokenization.

Language	Standard	Elkhound	Incremental (no cache)	Incremental	Tree-sitter (no cache)	Tree-sitter
Average	810.266	630.811	1 007.738	90.302	—	—
Java	2 033.161	1 579.048	2 518.010	217.591	233.929	19.715
WebDSL	241.125	185.905	303.368	27.597	—	—
SDF3	156.514	127.480	201.835	25.719	—	—

5.4 Memory Benchmark Results

This section shows the performance results of the ISGLR parser in terms of memory usage, compared to other parser variants in JSGLR2. All benchmark results in this section show the average memory performance over 10 benchmark iterations, where the error bars in the plots indicate the minimum and maximum measurement for that data point. Note that the error bars are barely visible because the memory usage was almost constant in all cases. For these memory benchmarks, we measured the full JSGLR2 parsing pipeline, which includes imploding and tokenization, as shown in the bottom row of Figures 2.10 and 3.12. Note that we only measured the memory usage when running the parsers in batch mode.

Memory Allocations Figure 5.7 shows the memory usage in terms of memory allocations. The memory usage of all parser variants tends to increase linearly with respect to the size of the input files for almost all files of the corpus. The average memory usage is shown in Table 5.8. On average, the ISGLR parser uses 23% more memory allocations than the Standard JSGLR2 parser variant and 67% more than the Elkhound variant.

We observe that Figure 5.7(a) shows some outliers where the memory usage is lower than for other files. After manual inspection, these Java files appeared to be abstract classes with a large number of documentation comments. This results in a smaller AST compared to a file of similar size that contains less documentation, therefore reducing memory usage in the imploder and tokenizer.

Table 5.8: Average memory allocations for the different languages in the evaluation corpus, as shown in Table 5.1, for three JSGLR2 parser variants. The values indicate the number of bytes allocated in the memory per character in the input, averaged over 100 input files.

Language	Standard	Elkhound	Incremental
Average	2 480	1 826	3 042
Java	2 379	1 713	2 878
WebDSL	2 287	1 653	2 782
SDF3	2 774	2 113	3 465

Cache Size Figure 5.9 shows the size of the parser cache after parsing, calculated as the difference between the memory footprints before and after parsing. We can see that the parsers that only support batch parsing indeed store no data that should be available during a subsequent parse. The incremental parser *does* store this data, the size of which seems to increase linearly with respect to the input file size, similar to the number of memory allocations. The average memory size of the cache is shown in Table 5.10.

Table 5.10: Average memory size of the cache for the different languages in the evaluation corpus, as shown in Table 5.1, for three JSGLR2 parser variants. The values indicate the number of bytes allocated in the memory per character in the input, averaged over 100 input files.

Language	Standard	Elkhound	Incremental
Average	0	0	270
Java	0	0	224
WebDSL	0	0	251
SDF3	0	0	336

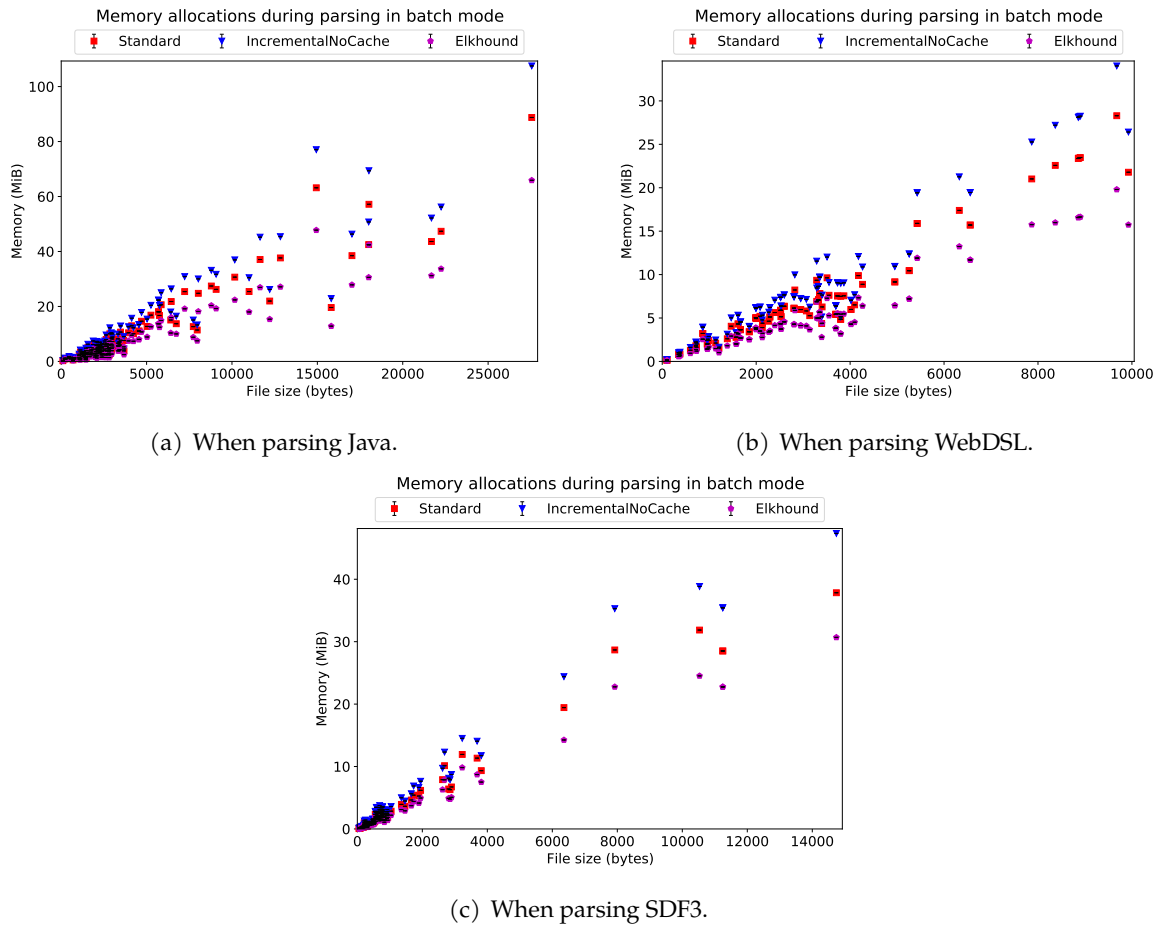


Figure 5.7: Size of memory allocations during parsing for three variants of the JSGLR2 parser: the Standard parser (red squares), the Elkhound parser variant (purple pentagons), and the ISGLR parser in batch mode (blue triangles). Memory usage is calculated for 100 files of different sizes.

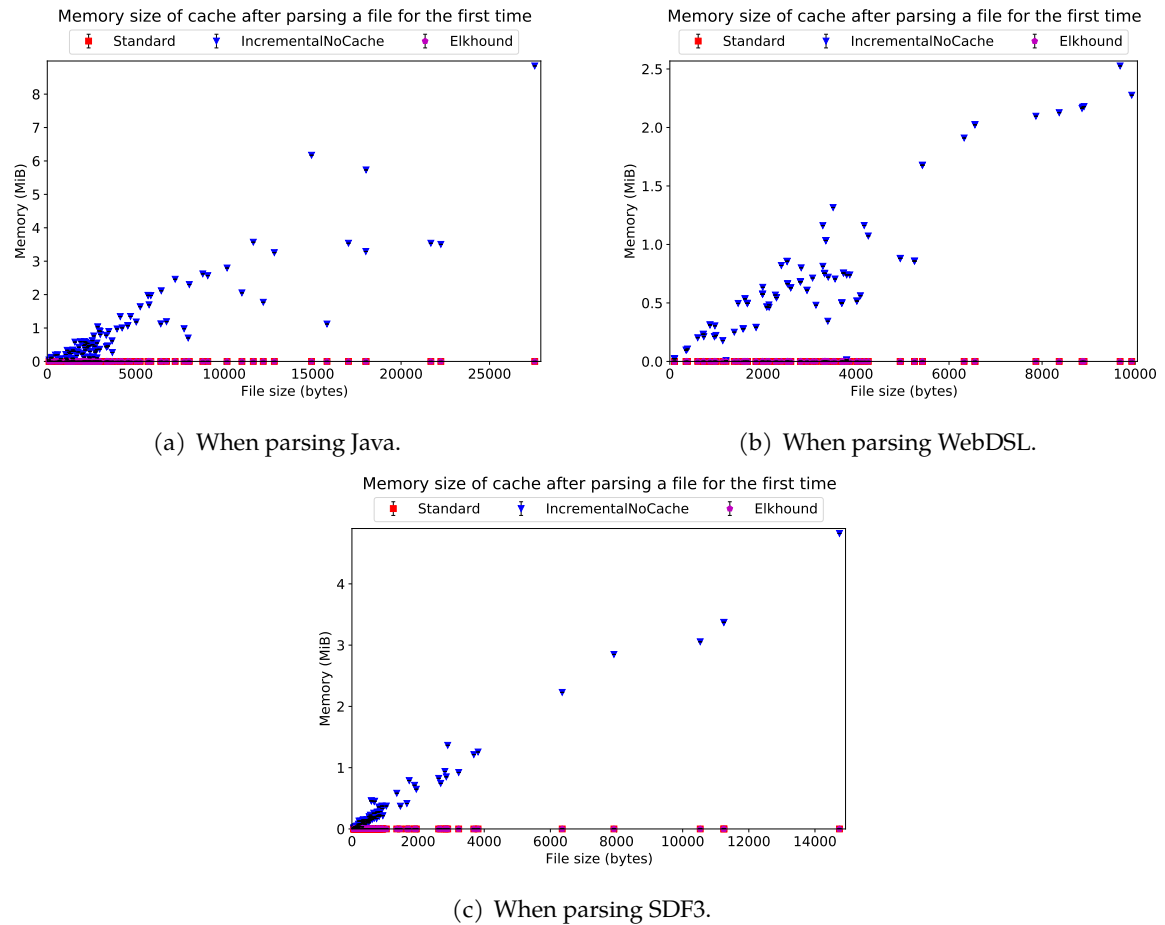


Figure 5.9: Size of the cache after parsing a file for three variants of the JSGLR2 parser: the Standard parser (red squares), the Elkhound parser variant (purple pentagons), and the ISGLR parser in batch mode (blue triangles). Memory usage is calculated for 100 files of different sizes.

5.5 Threats to Validity

The chosen method of performance evaluation has several concerns that might affect its validity. In this section, we discuss these threats to validity and to what extent we countered them.

Sampling Bias The selection of the evaluation corpus described in Section 5.1.1 raises the concern of sampling bias. It is not viable to evaluate the ISGLR parser with all possible languages and all available sources in that language, so selecting a subset of these as corpus is necessary, although it might not be representative. For the evaluation corpus, we selected languages that have different features and use cases to cover a range of languages that is as wide as possible.

Similarly, we need to select a part of the history of a project to use as input for the benchmark, as the full history of projects on GitHub consists of thousands of commits. Some commits may be smaller or larger than others, so by only taking the last 16 versions, we might introduce a bias towards a particular commit size. However, these commits typically still have larger change sizes than the size of a change would be when typing in the editor. The average change size of the sources in the chosen corpus is just below 1% of the input size, spread over multiple files, while typing in an editor leads to change sizes of only a couple of characters in a single file. Since we show that smaller change sizes typically take less time to parse incrementally, typing in the editor should not lead to performance issues. Conversely, a developer might switch branches in Git during development, which leads to larger change sizes than a single commit, since another branch may be multiple commits ahead or behind the current branch. However, we argue that the developer does not need an instant editor update in this scenario, implying that a slightly slower incremental update is acceptable in that case.

Finally, the parse time of the ISGLR parser does not only depend on the size of the change but also on what exactly changed. For example, changes to identifier names have less impact than changes that cause structural rearrangements in the AST. We have not attempted to quantify the “impactfulness” of a change, but we do suspect to have a sampling bias for this metric as well, besides the size of a change.

Correctness We do not prove the correctness of the ISGLR parser, so one might argue that the parser could return invalid results, possibly influencing the run time of the parser by taking certain shortcuts. Providing a full correctness proof of the ISGLR parser is outside the scope of this thesis, but we provide arguments for the necessity of certain parts of the ISGLR parsing algorithm in Chapter 3. However, we do have a suite of integration tests that is run automatically after pushing code to the main repository, ensuring that the basic functionality of the incremental parser is correct. In addition, the preprocessing step of the evaluation suite compares the resulting AST of an incremental parse with the AST of a batch parse for all files in the evaluation corpus. If the ASTs are not exactly equal, the entire benchmark is aborted. Of course, the concern of sampling bias also applies here: creating a test suite that covers all possible inputs is infeasible, and it remains possible that inputs outside the selected evaluation corpus might cause the ISGLR parser to return an incorrect result.

Chapter 6

Related Work

Many other incremental parsing techniques have been published in the past years. In this chapter, we describe these approaches and compare them to our Incremental Scannerless Generalized LR (ISGLR) parsing algorithm. We make a distinction between LR parsers (Section 6.1), recursive descent parsers (Section 6.2), and parser combinators (Section 6.3).

6.1 Incremental LR Parsers

The incremental parsers described in this section are based on LR parsing, with a lexer that produces tokens, and a token-level grammar that the parser uses to generate a parse tree. Section 2.1 describes LR parsers in detail.

Merlin Bour, Refis, and Scherer (2018) present Merlin, a language server for OCaml. As a language server, it can answer queries from the user, like requesting the type of an identifier and navigating to the definition of an identifier.

To gain incrementality, Merlin stores checkpoints during parsing. These checkpoints are references to immutable parse stack objects, which only results in a small memory overhead since there is a lot of sharing within parse stacks. When the input to the parser is edited, Merlin restarts the parser from the last checkpoint belonging to the last unaffected token before the edit region. This approach has the disadvantage that an edit near the start of the input file would require almost a full reparse.

Harmonia Wagner and Graham (1998) present an Incremental LR (ILR) parsing algorithm as an improvement over several earlier approaches to incremental LR parsing. For a description of these earlier approaches and how their shortcomings are resolved, we refer to Wagner and Graham (1998, §2). Later,¹ Wagner and Graham (1997b) extended this to allow non-deterministic parsing with the Incremental Generalized LR (IGLR) parsing algorithm, and Boshernitsan (2001) implemented an IGLR parser in the interactive setting of an IDE called Harmonia. Section 2.3 describes IGLR parsing in detail, as the ISGLR parsing algorithm presented in this work bases its incrementality on the IGLR parsing algorithm.

The ILR parser bases its incrementality on sentential-form parsing. It makes use of the assumption that the parse table reflects the grammar that it is generated from. The parser can reuse a parse node if the symbol that it represents is allowed to follow the parse state on top of the parse stack. The parse table generator provides information on which symbols are allowed to follow each parse state. This technique does not require state matching.

¹These two papers appear to have been published in reverse order, but the paper of Wagner and Graham (1997b) on IGLR parsing refers back to the paper of Wagner and Graham (1998) on ILR parsing as “Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing, 1996. *Submitted to ACM Trans. Program. Lang. Syst.*” In his PhD thesis, Wagner (1998, Chapters 6 & 7) did present ILR parsing *before* IGLR parsing.

However, this assumption of sentential-form parsing does not hold for production rules that have been disambiguated during parse table generation, e.g., using precedence and associativity rules. Wagner and Graham (1998) solve this issue by letting the ILR parser mark parse nodes as *fragile* when they represent such a production rule. If the parser encounters a fragile node during an incremental parse, it will always break it down. The IGLR parser does not need to track fragility, because it does use state matching, which (together with the lookahead test described in Section 2.3) also solves this issue.

6.1.1 Parsers Based on Wagner’s Approach

The three incremental parsing approaches described in this section are all based on the ILR parsing algorithm by Wagner and Graham (1998). All of these approaches allow language composition, but in a less transparent way than scannerless parsing does. The first two approaches do not allow non-deterministic (IGLR) parsing, which is considered an advantage by their authors since it forces language designers to create non-ambiguous grammars, which prevents accidental ambiguities to cause errors in the editor. In contrast, scannerless parsing heavily relies on non-deterministic parsing, as described in Section 3.2.

Eco Diekmann and Tratt (2014) present Eco, a language composition editor. It acts as a normal text editor, but internally it uses structural (also known as syntax-directed) editing to make sure that the user always operates on a valid syntax tree. Adding custom grammars to Eco is possible, but this is “mainly used for testing”² and not documented.

The Eco editor has a fixed list of combinations of languages that it allows. A user can edit this list in configuration files.³ In the editor, a user can create fragments in a different language by creating language boxes, which are zero-width boundaries around text fragments that indicate which language a fragment is written in. Language boxes can be nested arbitrarily deep.

Tree-sitter Brunfeldt (2018) presents Tree-sitter, a parser generator that generates incremental parsers. It is integrated into widely-adopted editors like Atom and Neovim and supports many General-Purpose Languages (GPLs). Language designers can generate Tree-sitter parsers by writing a grammar in JavaScript and running the generator with that grammar as input.⁴

Language composition is possible in Tree-sitter by manually instructing the parser which ranges of the input should be parsed in which language. It is possible to automate this by adding explicit fences to the host language (e.g., “<?= . . . ?>” for the PHP language), then detecting parse nodes corresponding to those constructs, and use the range of such a parse node to call the parser for the embedded language.⁵

Lezer Haverbeke (2019) presents Lezer, a parser generator inspired by Tree-sitter that is built into CodeMirror,⁶ a code editor that can be run inside a web browser. Lezer supports GLR parsing in an opt-in fashion by allowing the grammar designer to annotate which production rules may allow the parser to fork. Lezer’s lexer is integrated into the parser, allowing it to be contextual: token definitions are allowed to overlap, “as long as such tokens can’t occur in the same place anywhere in the grammar”.⁷ This also allows the layout definitions to depend on the context.

²<https://github.com/softdevteam/eco/pull/150>

³<https://github.com/softdevteam/eco/pull/238>

⁴<https://tree-sitter.github.io/tree-sitter/creating-parsers>

⁵<https://tree-sitter.github.io/tree-sitter/using-parsers#multi-language-documents>

⁶<https://codemirror.net/>

⁷<https://lezer.codemirror.net/docs/guide/#contextual-tokenization>

Lezer allows language composition directly in the grammar specification, as long as “the nested syntax has a clearly identifiable end token”.⁸ However, unlike Tree-sitter, Lezer grammars allow directly importing a nested grammar and declaratively specifying the boundary tokens. This comes very close to the seamless language composition that scannerless parsing allows, and is sufficient for the grammars of GPLs that are currently published.

6.2 Incremental Recursive Descent Parsers

This section describes incremental parsing approaches based on recursive descent (or top-down) parsing. Unlike LR(k) parsing, which builds a parse tree from the bottom up, recursive descent parsers start parsing from the top down. Starting with the start symbol, they try to recognize the input by enumerating all production rules for that symbol and recursively do this search for all symbols in a production.

Incremental LL(1) Yang (1993) and Li (1995), among others, have introduced algorithms for incremental LL(1) parsing. Both incremental LL(1) parsing algorithms “cut” the previous parse tree at the location of an edit: say that the input xyz is edited to $x\bar{y}z$, parse tree X is equal to the previous parse tree with holes at the locations where the parse nodes for substrings y and z would be, and Z is a list of subtrees that represent parts of z . Then, the parser initializes the LL(1) parsing algorithm as if it had already parsed X , with the sequences \bar{y} and Z as lookahead. To some extent, this is comparable with how ISGLR breaks down changed parse nodes and stores potentially reusable parse nodes on the input stack.

The algorithm of Yang (1993) decides whether it can reuse a subtree in Z by using its first terminal node to index the parse table, whereas Li (1995) improved this by allowing the parse table to contain non-terminals so that the parser can use the symbol in the root of a subtree to determine the next parse action. The latter approach is equal to how the ILR parser by Wagner and Graham (1997b) tests if it can reuse a parse node.

Incremental Packrat Parsing Packrat parsing (Ford 2002) is an alternative approach to recursive descent parsing that memoizes intermediate results. The so-called *memo table* is a table that saves for every input position whether parsing a production rule at that position succeeds and how much of the input it spans. The memo table is sparse because it only stores the entries that the parser needs during parsing.

Dubroy and Warth (2017) add incrementality to packrat parsing by taking the memo table from a previous parse, removing entries from it that can conflict with the edit, and starting the parser with this updated memo table. They have implemented this parsing algorithm in Ohm, a parser generator toolkit based on Parsing Expression Grammars (Warth, Dubroy, and Garnock-Jones 2016). The source code for this parser and some visualizations can be found online.⁹

The algorithm that updates the memo table removes all entries that overlap with the positions touched by the edit, including entries that required to look forward to these positions to make a proper decision. For example, the parser can only recognize a number consisting of multiple digits if the position *after* the number is not a digit. Because recursive descent parsers use backtracking, the lookahead is theoretically unlimited, so in the worst case, the update algorithm needs to test for all entries of the memo table whether it should remove them. However, the performance of the incremental packrat parser in practice suggests that this is often not the case.

⁸<https://lezer.codemirror.net/docs/guide/#grammar-nesting>

⁹<https://ohmlang.github.io/sle17/>

6.3 Incremental Parser Combinators

Bernardy and Claessen (2015) present an incremental parsing algorithm that makes use of a divide-and-conquer algorithm using parser combinators. The algorithm is implemented in the Yi editor¹⁰ (Bernardy 2008). They also convert a grammar from Backus–Naur Form (BNF) to a variant of Chomsky Normal Form (CNF) where every production rule is transformed into a binary tree of productions to reduce the height of the parse tree.

The incremental parser of Bernardy and Claessen (2015) assumes lazy access to the parse tree. For example, when a user has an editor of a file open, they only view a small part of this file, and therefore only this part of the file needs to be parsed to perform syntax highlighting. When the user scrolls down in the file, the parser can resume from where it left off.

The parser of Yi is incremental because it caches intermediate parse results. Bernardy describes this in a blog post about Yi: “For given positions in the input (say every half-page), we will store a partially evaluated state of the parsing automaton. Whenever the input is modified, the new parsing result will be computed by using the most relevant cached state, and applying the new input to it. The cached states that became invalidated will also be recomputed on the basis of the most relevant state.”¹¹

¹⁰<https://github.com/yi-editor/yi/>

¹¹<https://yi-editor.github.io/posts/2014-09-04-incremental-parsing/>

Chapter 7

Conclusion

This thesis presents the Incremental Scannerless Generalized LR (ISGLR) parsing algorithm and answers the following research question:

What are the Effects of Combining Scannerless and Incremental GLR Parsing?

To answer this research question, we combined SGLR parsing and IGLR parsing into the ISGLR parsing algorithm and implemented it in the context of Spoofax. The differences of these two parsing algorithms with respect to GLR parsing are orthogonal to each other and are combined without difficulties. While the algorithmic differences are orthogonal, we show that there exist non-trivial interactions between these two techniques.

The fact that scannerless parsing (with character-level grammars) relies on non-deterministic parsing for disambiguation has a negative impact on incrementality. A parse node is irreusable when the parser creates it while exploring multiple possibilities to achieve unbounded lookahead, which is the case for 27% of all parse nodes on average in the evaluated corpus. The ISGLR parser creates many of these irreusable parse nodes when parsing the layout between symbols or when parsing literal keywords that overlap with identifiers. If a production rule ends with a fencing character (like curly brackets (`{}`) or semicolons (`;`)), the parse node that they terminate *can* be reused in a subsequent incremental parse. In general, an incremental parser can reuse more of a previous parse tree if the grammar allows it to parse the input more deterministically.

Nonetheless, we show that the ISGLR parsing algorithm performs better than the batch SGLR parsing algorithm in typical scenarios. The majority of the irreusable parse nodes does not get exposed on top of the input stack during a subsequent incremental parse, so on average, the ISGLR parser can reuse 99% of a previous parse tree. When parsing from scratch, the ISGLR parser has a 24% run time overhead compared to the SGLR parser, but when parsing incrementally for changes that are smaller than 1% of the input size on average, it has a 9× speedup.

7.1 Future Work

Balanced Lists Wagner and Graham (1998) represent list production rules as balanced binary trees in the parse tree, while SDF3 normalizes list productions as left-recursive constructs. Using balanced trees reduces the height of the parse tree to $\mathcal{O}(\log n)$ (where n is the number of characters) instead of $\mathcal{O}(n)$ in the worst case. Reducing the height of the parse tree reduces the length of the spine that needs to be broken down during an incremental parse.

Index Parse Table With Productions Wagner and Graham (1998) adapt the parse table generator so that it allows their Incremental LR (ILR) parser to retrieve the applicable actions

from the action table using a state–non-terminal pair. This information is already available during parse table generation, but SDF3 does not store this in the parse table. The ISGLR parsing algorithm currently calculates the applicable actions for a non-terminal based on the first character in its parse node. Only when a parse node can be reused, it accesses the goto table using the production rule of the parse node. For this thesis, it was out of scope to improve the parse table generator, but it would be interesting to see if this improves the performance of the ISGLR parser.

Layout as Skip Productions The lexer of Lezer (Haverbeke 2019) is integrated into its parser, and this allows layout definitions to depend on the context, among other things. Their parser essentially “skips” any layout that matches these definitions. Reusing some ideas from Lezer might reduce non-determinism during parsing that is related to layout.

Incremental Error Recovery De Jonge et al. (2012) implement error recovery for SGLR parsing, and Wagner (1998, Chapter 8) and Brunsfeld (2018) implement error recovery for IGLR parsing and ILR parsing, respectively. Error recovery is currently a work in progress for JSGLR2 (Denkers 2018), and it would be interesting to combine this implementation with the incremental parser.

Integrate Into Spoofox While Spoofox already supports incremental type checking (Aerts 2019) and incremental compilation (Smits, Konat, and Visser 2020), these implementations are coarse-grained: a changed file will need to be fully reprocessed to some degree. They would benefit from detecting which elements in the AST have changed in a file based on the result from the incremental parser.

Incremental Editor Update Some tasks in the editor plugin that is generated by Spoofox would also benefit from incremental updates from the parser. The editor plugin performs syntax highlighting based on the tokens produced by the tokenization post-processing task. In addition, it generates a program outline based on the AST. Both of these tasks can be made incremental using the result from the incremental parser. Conversely, an editor might already have a built-in way of detecting textual updates, which can be used to improve the diff part of the ISGLR parsing algorithm.

Bibliography

- Aerts, Taico (2019). ‘Incrementalizing Statix: A Modular and Incremental Approach for Type Checking and Name Binding using Scope Graphs’. Master’s thesis. Delft University of Technology. URL: <https://resolver.tudelft.nl/uuid:3e0ea516-3058-4b8c-bfb6-5e846c4bd982>.
- van Antwerpen, Hendrik, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser (2018). ‘Scopes as types’. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA. DOI: 10.1145/3276484.
- Bernardy, Jean-Philippe (2008). ‘Yi: an editor in haskell for haskell’. In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Edited by Andy Gill. ACM, pages 61–62. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411294.
- Bernardy, Jean-Philippe and Koen Claessen (2015). ‘Efficient parallel and incremental parsing of practical context-free languages’. In: *Journal of Functional Programming* 25. DOI: 10.1017/S0956796815000131.
- Boshernitsan, Marat (2001). *HARMONIA: A Flexible Framework for Constructing Interactive Language-Based Programming Tools*. Technical report. URL: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2001/CSD-01-1149.pdf>.
- Bour, Frédéric, Thomas Refis, and Gabriel Scherer (2018). ‘Merlin: a language server for OCaml (experience report)’. In: *Proceedings of the ACM on Programming Languages* 2.ICFP. DOI: 10.1145/3236798.
- Bravenboer, Martin, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser (2008). ‘Stratego/XT 0.17. A language and toolset for program transformation’. In: *Science of Computer Programming* 72.1-2, pages 52–70. DOI: 10.1016/j.scico.2007.11.003.
- Brunsfeld, Max (GitHub) (2018-03-04). *Tree-sitter. A New Parsing System for Programming Tools*. Talk at: *Free and Open source Software Developers’ European Meeting, FOSDEM 2018, Brussels, Belgium, February 3-4, 2018*. URL: https://archive.fosdem.org/2018/schedule/event/code_tree_sitter/.
- Denkers, Jasper (2018). ‘A Modular SGLR Parsing Architecture for Systematic Performance Optimization’. Master’s thesis. Delft University of Technology. URL: <https://resolver.tudelft.nl/uuid:7d9f9bcc-117c-4617-860a-4e3e0bbc8988>.
- Diekmann, Lukas and Laurence Tratt (2014). ‘Eco: A Language Composition Editor’. In: *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*. Edited by Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju. Volume 8706. Lecture Notes in Computer Science. Springer, pages 82–101. ISBN: 978-3-319-11244-2. DOI: 10.1007/978-3-319-11245-9_5.
- Dubroy, Patrick and Alessandro Warth (2017). ‘Incremental packrat parsing’. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*. Edited by Benoît Combemale, Marjan

- Mernik, and Bernhard Rumpe. ACM, pages 14–25. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136022.
- Ford, Bryan (2002). ‘Packrat parsing: simple, powerful, lazy, linear time, functional pearl’. In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional Programming (ICFP 2002)*, pages 36–47. DOI: 10.1145/581478.581483.
- Fourment, Mathieu and Michael R. Gillings (2008). ‘A comparison of common programming languages used in bioinformatics’. In: *BMC Bioinformatics* 9. DOI: 10.1186/1471-2105-9-82.
- Haverbeke, Marijn (2019-09-03). *Lezer*. URL: <https://marijnhaverbeke.nl/blog/lezer.html> (visited on 2021-07-07).
- Jalili, Fahimeh and Jean H. Gallier (1982). ‘Building Friendly Parsers’. In: *POPL*, pages 196–206.
- Johnson, S. C. (1975). *YACC: Yet Another Compiler-Compiler*. Technical report CS-32. Murray Hill, NJ, USA: AT&T Bell Laboratories.
- Johnson, Walter L., James H. Porter, Stephanie I. Ackley, and Douglas T. Ross (1968). ‘Automatic generation of efficient lexical processors using finite state techniques’. In: *Communications of the ACM* 11.12, pages 805–813. DOI: 10.1145/364175.364185.
- de Jonge, Maartje, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg (2012). ‘Natural and Flexible Error Recovery for Generated Modular Language Environments’. In: *ACM Transactions on Programming Languages and Systems* 34.4, page 15. DOI: 10.1145/2400676.2400678.
- Kats, Lennart C. L. and Eelco Visser (2010). ‘The Spoofox language workbench: rules for declarative specification of languages and IDEs’. In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Edited by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, pages 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497.
- Knuth, Donald E. (1965). ‘On the translation of languages from left to right’. In: *Information and control* 8.6.
- Lesk, M. E. and E. Schmidt (1975). *Lex - A Lexical Analyzer Generator*. Technical report CS-39. Murray Hill, NJ, USA: AT&T Bell Laboratories.
- Li, Warren X. (1995). ‘A Simple and Efficient Incremental LL(1) parsing’. In: *SOFSEM 95, 22nd Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 23 - December 1, 1995, Proceedings*. Edited by Miroslav Bartosek, Jan Staudek, and Jiri Wiedermann. Volume 1012. Lecture Notes in Computer Science. Springer, pages 399–404. ISBN: 3-540-60609-2.
- McPeak, Scott and George C. Necula (2004). ‘Elkhound: A Fast, Practical GLR Parser Generator’. In: *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Volume 2985. Lecture Notes in Computer Science. Springer, pages 73–88. ISBN: 3-540-21297-3. DOI: 10.1007/978-3-540-24723-4_6.
- MetaBorg (2016). *The Spoofox Language Workbench*. URL: <https://www.metaborg.org/en/latest/> (visited on 2021-07-07).
- Mickunas, M. Dennis, Ronald L. Lancaster, and Victor B. Schneider (1976). ‘Transforming LR(k) Grammars to LR(1), SLR(1), and (1, 1) Bounded Right-Context Grammars’. In: *Journal of the ACM* 23.3, pages 511–533. DOI: 10.1145/321958.321972.
- Rekers, Jan (1992-01). ‘Parser Generation for Interactive Environments’. PhD thesis. Amsterdam, The Netherlands: University of Amsterdam. URL: <https://homepages.cwi.nl/~paulk/dissertations/Rekers.pdf>.
- Salomon, D. J. and G. V. Cormack (1989). ‘Scannerless NSLR(1) parsing of programming languages’. In: *SIGPLAN Not.* 24.7. DOI: 10.1145/74818.74833.
- Sipser, Michael (2012). *Introduction to the Theory of Computation*. 3rd. Cengage Learning. ISBN: 978-1-133-18779-0.

- Smits, Jeff, Gabriël Konat, and Eelco Visser (2020). 'Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System'. In: *Programming Journal* 4.3, page 16. DOI: 10.22152/programming-journal.org/2020/4/16.
- de Souza Amorim, Luis Eduardo, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser (2018). 'Declarative specification of indentation rules: a tooling perspective on parsing and pretty-printing layout-sensitive languages'. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. Edited by David Pearce 0005, Tanja Mayerhofer, and Friedrich Steimann. ACM, pages 3–15. ISBN: 978-1-4503-6029-6. DOI: 10.1145/3276604.3276607.
- de Souza Amorim, Luis Eduardo, Michael J. Steindorfer, and Eelco Visser (2018). 'Towards Zero-Overhead Disambiguation of Deep Priority Conflicts'. In: *Programming Journal* 2.3, page 13. DOI: 10.22152/programming-journal.org/2018/2/13.
- de Souza Amorim, Luis Eduardo and Eelco Visser (2020). 'Multi-purpose Syntax Definition with SDF3'. In: *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*. Edited by Frank S. de Boer and Antonio Cerone. Volume 12310. Lecture Notes in Computer Science. Springer, pages 1–23. ISBN: 978-3-030-58768-0. DOI: 10.1007/978-3-030-58768-0_1.
- Tomita, Masaru (1985). 'An Efficient Context-Free Parsing Algorithm for Natural Languages'. In: *IJCAI*, pages 756–764.
- Visser, Eelco (1997-07). *Scannerless Generalized-LR Parsing*. Technical report P9707. Programming Research Group, University of Amsterdam.
- Vollebregt, Tobi, Lennart C. L. Kats, and Eelco Visser (2012). 'Declarative specification of template-based textual editors'. In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. Edited by Anthony Sloane and Suzana Andova. ACM, pages 1–7. ISBN: 978-1-4503-1536-4. DOI: 10.1145/2427048.2427056.
- Wagner, Tim A. (1998-03). 'Practical Algorithms for Incremental Software Development Environments'. PhD thesis. EECS Department, University of California, Berkeley. URL: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/5885.html>.
- Wagner, Tim A. and Susan L. Graham (1997a). 'General Incremental Lexical Analysis'.
- Wagner, Tim A. and Susan L. Graham (1997b). 'Incremental Analysis of real Programming Languages'. In: *PLDI*, pages 31–43.
- Wagner, Tim A. and Susan L. Graham (1998). 'Efficient and Flexible Incremental Parsing'. In: *ACM Transactions on Programming Languages and Systems* 20.5, pages 980–1013. DOI: 10.1145/293677.293678.
- Warth, Alessandro, Patrick Dubroy, and Tony Garnock-Jones (2016). 'Modular semantic actions'. In: *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*. Edited by Roberto Ierusalimsky. ACM, pages 108–119. ISBN: 978-1-4503-4445-6. DOI: 10.1145/2989225.2989231.
- Willink, Edward D. (2001). 'Meta-compilation for C++'. British Library, EThOS. PhD thesis. University of Surrey, Guildford, UK. URL: <http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.370061>.
- Yang, Wu (1993). 'An Incremental LL(1) Parsing Algorithm'. In: *Inf. Process. Lett.* 48.2, pages 67–72.

This page is intentionally left blank.

Acronyms

AST	Abstract Syntax Tree
BNF	Backus–Naur Form
CNF	Chomsky Normal Form
CST	Concrete Syntax Tree
CSV	Comma-Separated Values
DSL	Domain-Specific Language
EOF	End-of-File
GLR	Generalized LR
GPL	General-Purpose Language
GSS	Graph-Structured Stack
IDE	Integrated Development Environment
IGLR	Incremental Generalized LR
ILR	Incremental LR
ISGLR	Incremental Scannerless Generalized LR
JMH	Java Microbenchmark Harness
JVM	Java Virtual Machine
LR	Left-to-right Rightmost-derivation
NSLR	Noncanonical Simple LR
SDF3	Syntax Definition Formalism 3 (Vollebregt, Kats, and Visser 2012; de Souza Amorim and Visser 2020)
SGLR	Scannerless Generalized LR
SLR	Simple LR

This page is intentionally left blank.

Glossary

- action** 4–7, 9, 14, 24, 26, 28–31, 33, 34, 37, 53, 65, 66
The three different types of actions listed in the *action table* that decide what the parser should do: *Shift*, *Reduce*, or *Accept*. Section 2.1.3 shows a full description of these actions.
- action table** 4, 30, 31, 34, 66
The part of the *parse table* that maps *state–terminal* pairs to *actions*.
- ambiguity** 6, 12, 40, 62, 77
A part of the *input stream* that can be parsed in multiple ways, represented as a *parse node* with multiple *derivations*.
- ambiguous grammar** 6, 62
A *context-free grammar* for which there exists some string that can be parsed in multiple ways.
- batch parse** 8, 24, 26, 29, 31, 49, 50, 60, *see* parse
A single invocation of a parser without reusing previous results, i.e., not being incremental.
- character class** 11, 45, 113
A compact notation to describe a set of characters. For example, `[A-Za-z]` denotes the set of all letters in the English alphabet.
- character node** 10, 13, 22, 26, 31–34, 36, 50, 53, 77
A leaf node of a *parse tree* for a scannerless parser, corresponding to a single character of the input string.
- character-level grammar** 10, 12, 24, 65
A *context-free grammar* that is described using characters as *terminal symbols*.
- conflict** 4, 6, 29–31
An entry in the *action table* that has more than one *action*.
- context-free grammar** 3–6, 10–12, 17–20, 24–31, 39, 43, 50, 51, 53, 55, 61–65, 115
A description of a language, as defined in Section 2.1.
- derivation** 6, 7, 13, 14, 28, *see* ambiguity
A possible way of parsing the input represented by the *parse node* that contains this derivation.
- diff** 17, 21, 22, 31, 33, 53, 55, 66
A list of changes between two strings of characters.

- goto table** 4, 5, 8, 22, 30, 34, 66
The part of the *parse table* that maps *state–non-terminal* pairs to other parse states.
- grammar normalization** 10–12, 17, 19, 28–30, 65
The process of transforming a high-level SDF3 *grammar* specification to a specification that only uses core constructs of SDF3.
- imploding** 14, 37, 39–42, 44, 47, 54–57, 92
The process of reducing a Concrete Syntax Tree (CST) to an Abstract Syntax Tree (AST).
- incremental parse** 8, 9, 17, 22–24, 26, 29–31, 36, 49, 50, 52, 53, 60, 62, 65, *see* parse
An invocation of the parser on an input that received a change with respect to a previous version. During an incremental parse, the parser tries to reuse subresults of a previous parse.
- injection** 10, 12, 41
A *production rule* in a *grammar* that has only one *symbol* on its right-hand side.
- input stack** 21, 22, 24, 26, 29–34, 36, 37, 49, 53, 63, 65
Used as input by the ISGLR parser. Contains both *character nodes* and *parse nodes*. If the parse node on top of the input stack may not be reused, it is broken down and its children are pushed back to the stack such that the first child ends up on top.
- input stream** 3, 4, 6–8, 13, 14, 21, 36, 37
The input to a parser, usually a stream of characters. In non-scannerless parsers, a *lexer* transforms this *input stream* into a stream of *tokens* before parsing.
- irreusable parse node** 9, 17, 24, 26, 29, 31, 34, 36, 37, 49, 50, 52, 53, 65, 77
A *parse node* that can never be reused during a subsequent *incremental parse* if it ends up at the top of the *input stack*, because it was created during *non-deterministic parsing*.
- kernel syntax** 10, 11, 19, 28–30
A low-level representation of *production rules* in SDF3, in which the *sorts* symbols (representing the *non-terminals*) are annotated with **-CF** or **-LEX** to remember whether it originates from **context-free syntax** or **lexical syntax**, respectively.
- layout** 3, 11, 12, 15, 24, 26, 29, 30, 39, 40, 45, 62, 65, 66
Parts of source code that do not contribute to the meaning of it, but only contribute to readability. Examples are comments (usually denoted with “//” or “/* */”) and characters representing whitespace, such as space (' ', U+0020), tab ('\t', U+0009), and newline ('\n', U+000A).
- lexer** 3, 8, 10, 12, 39, 43, 61, 62, 66
Used for parsers that recognize *token-level grammars*. Transforms an *input stream* of text characters into a stream of *tokens* that can be used as *terminals*.
- lexical element** 10–12, 39
Elements in the *grammar* of a language that would correspond to a single *token* in *token-level grammars*. Examples include identifier names, numbers, strings, and *literals*.
- literal** 11, 39, 40, 65
A non-varying *lexical element* in a *context-free grammar*. Examples include keywords and operators.
- metalinguage** 10, 51, 114
A language specifically used to describe another language, like SDF3.

- non-deterministic parsing** 6, 9, 17, 24, 29, 32, 52, 53, 61, 62, 65
When a parser needs to fork the *parse stack* during parsing, exploring multiple *parse states* at once to achieve *unbounded lookahead*. This does not imply that the final result contains *ambiguities*.
- non-terminal** 3, 4, 8, 10, 12, 63, 66
Shorthand for *non-terminal symbol*. Non-terminal symbols can be used on both sides of *production rules*.
- parse** 8, 13, 23, 57, 63, 77, *see* batch parse & incremental parse
(*noun*) An invocation of the parser.
- parse forest** 6, 13, 14, 36, 39, 49–53, 77
A *parse tree* that contains (or may contain) *ambiguities*. Multiple non-ambiguous parse trees could be constructed from a parse forest.
- parse node** 4, 6–9, 13, 14, 17, 21–37, 39–44, 49, 50, 52, 53, 61–63, 65, 66, 77
A non-leaf node in a *parse tree*, corresponding to a *production rule*.
- parse stack** 4–9, 13, 14, 22, 24–36, 61
Stores references to *parse states*, with the start state at the bottom and the current state at the top of the stack. Each link between two stack nodes stores a *parse node* or *terminal node*.
- parse state** 4–9, 13, 22, 29, 30, 32–37, 53, 61, 66
A state of an LR-based parser automaton, used in the *parse table* and the *parse stack*.
- parse table** 4, 6, 9, 10, 24, 29–31, 33, 34, 37, 53, 61, 66, 115
A two-dimensional table that represents the possible *state* transitions of an LR-based parser automaton. Consists of an *action table* and a *goto table*.
- parse table generator** 4, 6, 10, 14, 24, 30, 31, 61, 62, 65, 66, 115
An algorithm that generates a *parse table* from the *production rules* of a *context-free grammar*.
- parse tree** 3, 4, 6, 8–10, 17, 20–24, 26, 29, 31, 32, 39, 40, 43, 44, 49, 61, 63, 65
A tree-shaped representation of an input string, parsed according to a *context-free grammar*, containing *parse nodes* and *terminal nodes*.
- production rule** 3, 4, 8, 10–14, 26, 28–31, 34, 35, 39–41, 44, 45, 52, 53, 62–66
A rule in a *context-free grammar* of the form $A \rightarrow \alpha$ which maps *non-terminal* A to a string of *symbols* α .
- sort** 10–12, 22, 28–30, 53
(*noun*) A *non-terminal symbol* in SDF3.
- spine** 21, 32, 45, 53, 65
The path between a given node in a tree and the root of that tree.
- start symbol** 3, 12, 30, 63
The root *symbol* of a *context-free grammar*.
- state matching** 9, 17, 26, 29, 34, 53, 61, 62
A test that determines whether unchanged *parse nodes* can be reused, which is the case when the *parse state* reference stored in the parse node is the same as the current parse state.
- symbol** 3, 10–12, 14, 22, 29, 30, 33, 61, 63, 65
A symbol in a *context-free grammar*. Can be either a *non-terminal symbol* or a *terminal symbol*.

- terminal** 3–7, 10, 11, 33, 63
Shorthand for *terminal symbol*. Terminal symbols can only be used on the right-hand side of *production rules*.
- token** 3, 4, 8–10, 12, 14, 21, 24, 39, 43–47, 51, 61–63, 66, 114
A substring of the input to the parser, used as *terminal symbol* when parsing *token-level grammars* (Section 2.1) or generated after parsing with *character-level grammars* (Section 4.2).
- token-level grammar** 10, 12, 29, 43, 61
A *context-free grammar* that is described using *tokens* as *terminal symbols*.
- tokenization** 14, 37, 39, 43–47, 54–57, 66, 92
The process of generating a list of *tokens* from a *parse tree* and attaching these tokens to the corresponding Abstract Syntax Tree (AST).
- unbounded lookahead** 6, 9, 65
A property of a parser that allows it to look ahead an arbitrary number of *terminal symbols* during parsing. In Generalized LR (GLR) parsing, this is accomplished by *non-deterministically parsing* the input in pseudo-parallel, using multiple active *parse stacks* at once.

Appendix A

Full Evaluation Results

This appendix contains the full evaluation results for the Incremental Scannerless Generalized LR (ISGLR) parser. These results are also available online at the Spoofox evaluation website for JSGLR2: https://www.spoofox.dev/jsglr2evaluation-site/2021-06-15_18:13/.

A.1 Measurements

The tables on the left are counts of parse nodes and character nodes after a parse for one version of the input has finished. For parse nodes, it also shows the number of ambiguous parse nodes and the number of irreusable parse nodes (percentages are relative to the “Parse Nodes Count” column).

The tables on the right are counts of certain events happening during parsing. It shows how many parse nodes were created during parsing and how many parse nodes were reused or rebuilt (percentages are relative to the “Parse Nodes Count” of the row at the same height in the left part of the table, i.e., the previous version of the parse forest). The “Shift” column shows how many parse nodes and character nodes the parser has shifted during the parse. The “Breakdown Count” column shows how many parse nodes were broken down during parsing (percentages are relative to the “Parse Nodes Count” column of the row at the same height in the left part of the table). The other “Breakdown” columns give a breakdown (pun intended) of the different types of breakdowns (percentages are relative to the “Breakdown Count” column), for which the full explanation can be found in Section 5.2.2.

A copy of Table 5.2 in Section 5.2.

Language	Parse nodes (% of total nodes)				Breakdowns (% of total breakdowns)			
	Irre-usable	Reused	Broken down	Rebuilt	Contains Change	Irre-usable	No actions	Wrong state
Average	27.25%	99.12%	0.67%	0.48%	40.68%	56.23%	0.00%	3.08%
Java	19.08%	99.60%	0.17%	0.08%	51.35%	46.67%	0.00%	1.98%
WebDSL	21.32%	99.55%	0.28%	0.15%	51.35%	43.76%	0.00%	4.89%
SDF3	41.36%	98.21%	1.57%	1.21%	19.35%	78.28%	0.00%	2.38%

Incremental parsing measurements for all languages.

Language	Parse Nodes			Character Nodes Count	Language	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average	587 518	0.34%	27.25%	336 819	Average	7 816	99.12%	0.48%	411	1 072	0.67%	40.68%	56.23%	0.00%	3.08%
Java	1 453 416	0.00%	19.08%	852 068	Java	13 989	99.60%	0.08%	631	1 849	0.17%	51.35%	46.67%	0.00%	1.98%
WebDSL	198 475	0.00%	21.32%	101 295	WebDSL	3 733	99.55%	0.15%	324	589	0.28%	51.35%	43.76%	0.00%	4.89%
SDF3	110 664	1.01%	41.36%	57 095	SDF3	5 725	98.21%	1.21%	279	778	1.57%	19.35%	78.28%	0.00%	2.38%

A.1.1 Java

Incremental parsing measurements for the Java language.

Source	Parse Nodes			Character Nodes Count	Source	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average	1 453 416	0.00%	19.08%	852 068	Average	13 989	99.60%	0.08%	631	1 849	0.17%	51.35%	46.67%	0.00%	1.98%
StringUtils	572 516	0.00%	15.27%	396 264	StringUtils	8 186	99.65%	0.14%	607	826	0.28%	47.76%	51.46%	0.00%	0.78%
gson	2 304 220	0.00%	23.36%	1 264 778	gson	7 883	99.93%	0.02%	534	907	0.05%	53.68%	45.10%	0.00%	1.22%
slf4j	1 483 511	0.00%	18.61%	895 163	slf4j	25 897	99.21%	0.07%	752	3 814	0.19%	52.63%	43.44%	0.00%	3.93%

Incremental parsing measurements for Java source StringUtils.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Count	Breakdown			
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node		Contains Change	Irreusable	No Actions	Wrong State
Average (0..14)	572 516	0 (0.00%)	87 442 (15.27%)	396 264	Average (1..15)	8 186	570 514 (99.65%)	802 (0.14%)	607	826	1 585 (0.28%)	809 (47.76%)	745 (51.46%)	0 (0.00%)	31 (0.78%)
					→ 0	1 058 237			0	395 111					
0	572 123	0 (0.00%)	87 583 (15.31%)	395 110	0 → 1	4 125	571 624 (99.91%)	269 (0.05%)	463	86	498 (0.09%)	225 (45.18%)	273 (54.82%)	0 (0.00%)	0 (0.00%)
1	572 280	0 (0.00%)	87 611 (15.31%)	395 172	1 → 2	3 430	571 811 (99.92%)	294 (0.05%)	307	139	468 (0.08%)	146 (31.20%)	318 (67.95%)	0 (0.00%)	4 (0.85%)
2	572 360	0 (0.00%)	87 639 (15.31%)	395 216	2 → 3	10 379	569 573 (99.51%)	1 187 (0.21%)	864	643	1 975 (0.35%)	815 (41.27%)	1 136 (57.52%)	0 (0.00%)	24 (1.22%)
3	571 559	0 (0.00%)	87 419 (15.29%)	394 856	3 → 4	10 617	571 140 (99.93%)	207 (0.04%)	345	2 829	418 (0.07%)	214 (51.20%)	203 (48.56%)	0 (0.00%)	1 (0.24%)
4	575 679	0 (0.00%)	88 130 (15.31%)	397 653	4 → 5	38 596	560 927 (97.44%)	4 185 (0.73%)	2 107	5 078	11 288 (1.96%)	7 415 (65.69%)	3 609 (31.97%)	0 (0.00%)	264 (2.34%)
5	571 608	0 (0.00%)	87 203 (15.26%)	395 931	5 → 6	52	571 537 (99.99%)	12 (0.00%)	9	2	17 (0.00%)	14 (82.35%)	3 (17.65%)	0 (0.00%)	0 (0.00%)
6	571 549	0 (0.00%)	87 192 (15.26%)	395 896	6 → 7	2 400	570 976 (99.90%)	250 (0.04%)	257	81	464 (0.08%)	245 (52.80%)	218 (46.98%)	0 (0.00%)	1 (0.22%)
7	571 380	0 (0.00%)	87 129 (15.25%)	395 842	7 → 8	3 306	570 856 (99.91%)	323 (0.06%)	358	84	515 (0.09%)	149 (28.93%)	362 (70.29%)	0 (0.00%)	4 (0.78%)
8	571 370	0 (0.00%)	87 120 (15.25%)	395 842	8 → 9	4 113	571 080 (99.95%)	120 (0.02%)	113	1 415	289 (0.05%)	171 (59.17%)	117 (40.48%)	0 (0.00%)	1 (0.35%)
9	573 253	0 (0.00%)	87 429 (15.25%)	397 158	9 → 10	3 055	572 754 (99.91%)	288 (0.05%)	372	81	498 (0.09%)	180 (36.14%)	311 (62.45%)	0 (0.00%)	7 (1.41%)
10	573 271	0 (0.00%)	87 435 (15.25%)	397 176	10 → 11	3 824	572 704 (99.90%)	253 (0.04%)	505	49	566 (0.10%)	317 (56.01%)	245 (43.29%)	0 (0.00%)	4 (0.71%)
11	573 279	0 (0.00%)	87 435 (15.25%)	397 184	11 → 12	6 150	571 851 (99.75%)	943 (0.16%)	681	171	1 110 (0.19%)	393 (35.41%)	714 (64.32%)	0 (0.00%)	3 (0.27%)
12	572 868	0 (0.00%)	87 432 (15.26%)	397 048	12 → 13	1 941	572 391 (99.92%)	257 (0.04%)	238	114	476 (0.08%)	239 (50.21%)	236 (49.58%)	0 (0.00%)	1 (0.21%)
13	572 909	0 (0.00%)	87 432 (15.26%)	397 077	13 → 14	26 307	566 933 (98.96%)	3 134 (0.55%)	2 027	1 457	4 549 (0.79%)	1 280 (28.14%)	3 125 (68.70%)	0 (0.00%)	144 (3.17%)
14	572 251	0 (0.00%)	87 448 (15.28%)	396 792	14 → 15	4 491	571 559 (99.88%)	307 (0.05%)	465	158	643 (0.11%)	339 (52.72%)	304 (47.28%)	0 (0.00%)	0 (0.00%)
15	572 267	0 (0.00%)	87 424 (15.28%)	396 796											

Incremental parsing measurements for Java source gson.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average (0..14)	2 304 220	0 (0.00%)	538 217 (23.36%)	1 264 778	Average (1..15)	7 883	2 302 621 (99.93%)	430 (0.02%)	534	907	1 039 (0.05%)	559 (53.68%)	459 (45.10%)	0 (0.00%)	20 (1.22%)
					→ 0	6 362 523			0	1 263 722					
0	2 302 158	0 (0.00%)	537 864 (23.36%)	1 263 518	0 → 1	7 120	2 299 996 (99.91%)	457 (0.02%)	580	709	1 622 (0.07%)	1 002 (61.78%)	560 (34.53%)	0 (0.00%)	60 (3.70%)
1	2 301 731	0 (0.00%)	537 769 (23.36%)	1 263 366	1 → 2	2 253	2 301 180 (99.98%)	270 (0.01%)	346	182	477 (0.02%)	214 (44.86%)	232 (48.64%)	0 (0.00%)	31 (6.50%)
2	2 301 561	0 (0.00%)	537 742 (23.36%)	1 263 290	2 → 3	3 267	2 300 876 (99.97%)	333 (0.01%)	494	144	701 (0.03%)	287 (40.94%)	410 (58.49%)	0 (0.00%)	4 (0.57%)
3	2 301 512	0 (0.00%)	537 746 (23.36%)	1 263 286	3 → 4	36 200	2 293 631 (99.66%)	2 972 (0.13%)	2 755	1 742	6 093 (0.26%)	2 519 (41.34%)	3 408 (55.93%)	0 (0.00%)	166 (2.72%)
4	2 300 237	0 (0.00%)	537 100 (23.35%)	1 262 874	4 → 5	0			204	0					
5	2 300 237	0 (0.00%)	537 100 (23.35%)	1 262 874	5 → 6	1 504	2 300 069 (99.99%)	74 (0.00%)	309	160	167 (0.01%)	61 (36.53%)	106 (63.47%)	0 (0.00%)	0 (0.00%)
6	2 300 455	0 (0.00%)	537 130 (23.35%)	1 263 000	6 → 7	3 474	2 299 610 (99.96%)	260 (0.01%)	442	161	442 (0.02%)	304 (68.78%)	134 (30.32%)	0 (0.00%)	4 (0.90%)
7	2 300 304	0 (0.00%)	537 170 (23.35%)	1 262 824	7 → 8	514	2 300 290 (100.00%)	7 (0.00%)	213	377	13 (0.00%)	9 (69.23%)	4 (30.77%)	0 (0.00%)	0 (0.00%)
8	2 300 750	0 (0.00%)	537 218 (23.35%)	1 263 198	8 → 9	3 867	2 297 824 (99.87%)	876 (0.04%)	585	1 741	2 943 (0.13%)	2 099 (71.32%)	832 (28.27%)	0 (0.00%)	12 (0.41%)
9	2 300 828	0 (0.00%)	537 218 (23.35%)	1 263 276	9 → 10	31 772	2 300 657 (99.99%)	84 (0.00%)	321	4 648	169 (0.01%)	83 (49.11%)	86 (50.89%)	0 (0.00%)	0 (0.00%)
10	2 310 537	0 (0.00%)	539 822 (23.36%)	1 267 874	10 → 11	950	2 310 474 (100.00%)	30 (0.00%)	268	166	62 (0.00%)	34 (54.84%)	28 (45.16%)	0 (0.00%)	0 (0.00%)
11	2 310 798	0 (0.00%)	539 875 (23.36%)	1 268 038	11 → 12	1 192	2 310 659 (99.99%)	84 (0.00%)	296	102	138 (0.01%)	32 (23.19%)	106 (76.81%)	0 (0.00%)	0 (0.00%)
12	2 310 965	0 (0.00%)	539 913 (23.36%)	1 268 122	12 → 13	7 289	2 307 965 (99.87%)	304 (0.01%)	375	866	805 (0.03%)	500 (62.11%)	299 (37.14%)	0 (0.00%)	6 (0.75%)
13	2 309 965	0 (0.00%)	539 656 (23.36%)	1 267 740	13 → 14	11 558	2 307 585 (99.90%)	397 (0.02%)	441	1 737	1 145 (0.05%)	748 (65.33%)	388 (33.89%)	0 (0.00%)	9 (0.79%)
14	2 311 257	0 (0.00%)	539 929 (23.36%)	1 268 390	14 → 15	7 289	2 308 257 (99.87%)	304 (0.01%)	375	866	805 (0.03%)	500 (62.11%)	299 (37.14%)	0 (0.00%)	6 (0.75%)
15	2 310 257	0 (0.00%)	539 672 (23.36%)	1 268 008											

Incremental parsing measurements for Java source slf4j.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average (0..14)	1483511	0 (0.00%)	276101 (18.61%)	895163	Average (1..15)	25897	1471637 (99.21%)	1053 (0.07%)	752	3814	2918 (0.19%)	1936 (52.63%)	773 (43.44%)	0 (0.00%)	209 (3.93%)
					→ 0	3584556			0	913020					
0	1519876	0 (0.00%)	282515 (18.59%)	912784	0 → 1	13481	1517062 (99.81%)	2548 (0.17%)	1486	1074	2813 (0.19%)	9 (0.32%)	2384 (84.75%)	0 (0.00%)	420 (14.93%)
1	1519878	0 (0.00%)	282515 (18.59%)	912785	1 → 2	320	1519682 (99.99%)	74 (0.00%)	274	175	203 (0.01%)	181 (89.16%)	21 (10.34%)	0 (0.00%)	1 (0.49%)
2	1519912	0 (0.00%)	282519 (18.59%)	912813	2 → 3	1626	1519648 (99.98%)	160 (0.01%)	451	40	263 (0.02%)	81 (30.80%)	178 (67.68%)	0 (0.00%)	4 (1.52%)
3	1519914	0 (0.00%)	282518 (18.59%)	912814	3 → 4	62643	1499575 (98.66%)	604 (0.04%)	639	10153	1719 (0.11%)	1134 (65.97%)	533 (31.01%)	0 (0.00%)	52 (3.03%)
4	1520978	0 (0.00%)	282566 (18.58%)	913852	4 → 5	121190	1466834 (96.44%)	5295 (0.35%)	2698	12787	16988 (1.12%)	12645 (74.43%)	3903 (22.98%)	0 (0.00%)	440 (2.59%)
5	1496179	0 (0.00%)	278176 (18.59%)	902914	5 → 6	1012	1495997 (99.99%)	80 (0.01%)	339	42	187 (0.01%)	86 (45.99%)	99 (52.94%)	0 (0.00%)	2 (1.07%)
6	1496177	0 (0.00%)	278176 (18.59%)	902912	6 → 7	125435	1422510 (95.08%)	2978 (0.20%)	1278	21699	10631 (0.71%)	7981 (75.07%)	1693 (15.93%)	0 (0.00%)	957 (9.00%)
7	1464481	0 (0.00%)	272295 (18.59%)	885244	7 → 8	46590	1440557 (98.37%)	3243 (0.22%)	1579	7967	9335 (0.64%)	6035 (64.65%)	2065 (22.12%)	0 (0.00%)	1235 (13.23%)
8	1455828	0 (0.00%)	271349 (18.64%)	881191	8 → 9	740	1455669 (99.99%)	107 (0.01%)	334	33	144 (0.01%)	47 (32.64%)	94 (65.28%)	0 (0.00%)	3 (2.08%)
9	1455801	0 (0.00%)	271330 (18.64%)	881184	9 → 10	431	1455639 (99.99%)	70 (0.00%)	303	11	84 (0.01%)	47 (55.95%)	35 (41.67%)	0 (0.00%)	2 (2.38%)
10	1455715	0 (0.00%)	271317 (18.64%)	881132	10 → 11	977	1455512 (99.99%)	103 (0.01%)	406	17	197 (0.01%)	98 (49.75%)	99 (50.25%)	0 (0.00%)	0 (0.00%)
11	1455707	0 (0.00%)	271322 (18.64%)	881134	11 → 12	3895	1455364 (99.98%)	179 (0.01%)	381	494	350 (0.02%)	155 (44.29%)	188 (53.71%)	0 (0.00%)	7 (2.00%)
12	1456442	0 (0.00%)	271509 (18.64%)	881518	12 → 13	281	1456396 (100.00%)	30 (0.00%)	265	112	43 (0.00%)	20 (46.51%)	21 (48.84%)	0 (0.00%)	2 (4.65%)
13	1456563	0 (0.00%)	271522 (18.64%)	881618	13 → 14	7226	1455937 (99.96%)	137 (0.01%)	404	2192	247 (0.02%)	97 (39.27%)	148 (59.92%)	0 (0.00%)	2 (0.81%)
14	1459217	0 (0.00%)	271885 (18.63%)	883547	14 → 15	2602	1458180 (99.93%)	190 (0.01%)	445	407	567 (0.04%)	423 (74.60%)	137 (24.16%)	0 (0.00%)	7 (1.23%)
15	1458960	0 (0.00%)	271819 (18.63%)	883490											

A.1.2 WebDSL

Incremental parsing measurements for the WebDSL language.

Source	Parse Nodes			Character Nodes Count	Source	Parse Nodes			Shift		Breakdown				
	Count	Ambi- guous	Irre- usable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irre- usable	No Actions	Wrong State
Average	198 475	0.00%	21.32%	101 295	Average	3 733	99.55%	0.15%	324	589	0.28%	51.35%	43.76%	0.00%	4.89%
builtin.app	182 107	0.00%	23.09%	98 145	builtin.app	2 693	99.28%	0.15%	457	343	0.27%	56.16%	39.28%	0.00%	4.56%
YellowGrass	332 492	0.00%	19.68%	166 333	YellowGrass	6 632	99.79%	0.09%	334	1 220	0.17%	49.13%	46.03%	0.00%	4.84%
elib-utils	80 827	0.00%	21.20%	39 406	elib-utils	1 874	99.57%	0.21%	181	204	0.40%	48.75%	45.97%	0.00%	5.28%

Incremental parsing measurements for WebDSL source builtin.app.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average (1..9)	182 107	0 (0.00%)	42 046 (23.09%)	98 145	Average (2..10)	2 693	180 774 (99.28%)	277 (0.15%)	457	343	497 (0.27%)	222 (56.16%)	247 (39.28%)	0 (0.00%)	28 (4.56%)
					→ 1	479 900			0	100 883					
1	186 944	0 (0.00%)	43 193 (23.10%)	100 882	1 → 2	249	180 143 (96.36%)	78 (0.04%)	121	11	120 (0.06%)	95 (79.17%)	18 (15.00%)	0 (0.00%)	7 (5.83%)
2	180 226	0 (0.00%)	41 560 (23.06%)	97 219	2 → 3	5 078	180 026 (99.89%)	59 (0.03%)	303	883	199 (0.11%)	143 (71.86%)	49 (24.62%)	0 (0.00%)	7 (3.52%)
3	181 858	0 (0.00%)	42 160 (23.18%)	98 047	3 → 4	3 060	181 400 (99.75%)	208 (0.11%)	291	308	447 (0.25%)	338 (75.62%)	88 (19.69%)	0 (0.00%)	21 (4.70%)
4	182 288	0 (0.00%)	42 160 (23.13%)	98 231	4 → 5	6 365	179 810 (98.64%)	1 174 (0.64%)	943	963	1 653 (0.91%)	516 (31.22%)	988 (59.77%)	0 (0.00%)	149 (9.01%)
5	181 241	0 (0.00%)	41 808 (23.07%)	97 627	5 → 6	605	181 126 (99.94%)	24 (0.01%)	198	66	114 (0.06%)	89 (78.07%)	18 (15.79%)	0 (0.00%)	7 (6.14%)
6	181 366	0 (0.00%)	41 853 (23.08%)	97 683	6 → 7	1 855	180 898 (99.74%)	212 (0.12%)	538	124	467 (0.26%)	203 (43.47%)	255 (54.60%)	0 (0.00%)	9 (1.93%)
7	181 437	0 (0.00%)	41 867 (23.08%)	97 721	7 → 8	2 865	180 846 (99.67%)	320 (0.18%)	724	308	590 (0.33%)	230 (38.98%)	323 (54.75%)	0 (0.00%)	37 (6.27%)
8	181 783	0 (0.00%)	41 902 (23.05%)	97 935	8 → 9	1 773	181 355 (99.76%)	181 (0.10%)	519	103	427 (0.23%)	204 (47.78%)	214 (50.12%)	0 (0.00%)	9 (2.11%)
9	181 824	0 (0.00%)	41 909 (23.05%)	97 956	9 → 10	2 387	181 361 (99.75%)	236 (0.13%)	474	318	456 (0.25%)	179 (39.25%)	270 (59.21%)	0 (0.00%)	7 (1.54%)
10	182 132	0 (0.00%)	41 944 (23.03%)	98 172											

Incremental parsing measurements for WebDSL source YellowGrass.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average (0..14)	332 492	0 (0.00%)	65 441 (19.68%)	166 333	Average (1..15)	6 632	331 816 (99.79%)	301 (0.09%)	334	1 220	549 (0.17%)	237 (49.13%)	286 (46.03%)	0 (0.00%)	26 (4.84%)
0	318 753	0 (0.00%)	63 030 (19.77%)	158 975	→ 0	703 238	317 608 (99.64%)	563 (0.18%)	0	159 026	1 045 (0.33%)	486 (46.51%)	529 (50.62%)	0 (0.00%)	30 (2.87%)
1	321 421	0 (0.00%)	63 523 (19.76%)	160 568	0 → 1	11 089	317 565 (98.80%)	1 344 (0.42%)	653	2 080	2 193 (0.68%)	790 (36.02%)	1 245 (56.77%)	0 (0.00%)	158 (7.20%)
2	320 082	0 (0.00%)	63 043 (19.70%)	159 970	1 → 2	9 470	319 478 (99.81%)	306 (0.10%)	970	1 014	576 (0.18%)	246 (42.71%)	290 (50.35%)	0 (0.00%)	40 (6.94%)
3	321 187	0 (0.00%)	63 228 (19.69%)	160 445	2 → 3	4 400	319 903 (99.60%)	791 (0.25%)	388	670	1 297 (0.40%)	411 (31.69%)	853 (65.77%)	0 (0.00%)	33 (2.54%)
4	323 904	0 (0.00%)	64 046 (19.77%)	161 720	3 → 4	13 573	323 678 (99.93%)	189 (0.06%)	818	1 714	229 (0.07%)	65 (28.38%)	126 (55.02%)	0 (0.00%)	38 (16.59%)
5	323 900	0 (0.00%)	64 047 (19.77%)	161 716	4 → 5	936	323 741 (99.95%)	81 (0.03%)	176	99	157 (0.05%)	42 (26.75%)	96 (61.15%)	0 (0.00%)	19 (12.10%)
6	326 852	0 (0.00%)	64 504 (19.73%)	163 359	5 → 6	6 626	326 756 (99.97%)	26 (0.01%)	157	1 679	96 (0.03%)	63 (65.63%)	33 (34.38%)	0 (0.00%)	0 (0.00%)
7	326 874	0 (0.00%)	64 504 (19.73%)	163 370	6 → 7	345	326 775 (99.97%)	72 (0.02%)	105	49	98 (0.03%)	22 (22.45%)	72 (73.47%)	0 (0.00%)	4 (4.08%)
8	335 813	0 (0.00%)	65 837 (19.61%)	168 635	7 → 8	21 228	344 565 (99.94%)	89 (0.03%)	148	5 297	183 (0.05%)	135 (73.77%)	32 (17.49%)	0 (0.00%)	16 (8.74%)
9	344 778	0 (0.00%)	67 624 (19.61%)	172 696	8 → 9	19 230	344 782 (99.96%)	32 (0.01%)	53	4 062	150 (0.04%)	120 (80.00%)	30 (20.00%)	0 (0.00%)	0 (0.00%)
10	344 930	0 (0.00%)	67 645 (19.61%)	172 793	9 → 10	977	344 967 (99.99%)	19 (0.01%)	93	178	45 (0.01%)	21 (46.67%)	24 (53.33%)	0 (0.00%)	0 (0.00%)
11	345 013	0 (0.00%)	67 645 (19.61%)	172 833	10 → 11	481	344 215 (99.77%)	157 (0.05%)	121	102	799 (0.23%)	603 (75.47%)	195 (24.41%)	0 (0.00%)	1 (0.13%)
12	345 015	0 (0.00%)	67 645 (19.61%)	172 834	11 → 12	144	344 235 (99.93%)	41 (0.01%)	170	70	142 (0.04%)	108 (76.06%)	30 (21.13%)	0 (0.00%)	4 (2.82%)
13	344 461	0 (0.00%)	67 645 (19.64%)	172 557	12 → 13	844	343 164 (99.64%)	810 (0.24%)	121	72	1 222 (0.35%)	437 (35.76%)	740 (60.56%)	0 (0.00%)	45 (3.68%)
14	344 393	0 (0.00%)	67 996 (19.65%)	173 274	13 → 14	420			947	1 202					
15	346 034	0 (0.00%)			14 → 15	9 710									

Incremental parsing measurements for WebDSL source elib-utils.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average (0..14)	80 827	0 (0.00%)	17 136 (21.20%)	39 406	Average (1..15)	1 874	80 481 (99.57%)	172 (0.21%)	181	204	326 (0.40%)	148 (48.75%)	160 (45.97%)	0 (0.00%)	18 (5.28%)
					→ 0	187 245			0	38 972					
0	80 035	0 (0.00%)	16 982 (21.22%)	38 955	0 → 1	327	79 962 (99.91%)	65 (0.08%)	73	22	74 (0.09%)	41 (55.41%)	33 (44.59%)	0 (0.00%)	0 (0.00%)
1	80 033	0 (0.00%)	16 982 (21.22%)	38 953	1 → 2	3 529	79 550 (99.40%)	381 (0.48%)	170	375	482 (0.60%)	31 (6.43%)	449 (93.15%)	0 (0.00%)	2 (0.41%)
2	80 255	0 (0.00%)	16 986 (21.17%)	39 118	2 → 3	3 138	79 671 (99.27%)	328 (0.41%)	227	323	523 (0.65%)	165 (31.55%)	297 (56.79%)	0 (0.00%)	61 (11.66%)
3	80 437	0 (0.00%)	17 030 (21.17%)	39 211	3 → 4	1 085	80 424 (99.98%)	7 (0.01%)	31	169	12 (0.01%)	7 (58.33%)	5 (41.67%)	0 (0.00%)	0 (0.00%)
4	80 765	0 (0.00%)	17 147 (21.23%)	39 378	4 → 5	1 125	80 417 (99.57%)	109 (0.13%)	111	104	308 (0.38%)	215 (69.81%)	86 (27.92%)	0 (0.00%)	7 (2.27%)
5	80 669	0 (0.00%)	17 106 (21.21%)	39 317	5 → 6	1 378	80 053 (99.24%)	376 (0.47%)	296	71	661 (0.82%)	512 (77.46%)	101 (15.28%)	0 (0.00%)	48 (7.26%)
6	80 575	0 (0.00%)	17 112 (21.24%)	39 283	6 → 7	628	80 498 (99.90%)	45 (0.06%)	71	57	74 (0.09%)	38 (51.35%)	36 (48.65%)	0 (0.00%)	0 (0.00%)
7	80 645	0 (0.00%)	17 140 (21.25%)	39 320	7 → 8	655	80 368 (99.66%)	99 (0.12%)	133	84	240 (0.30%)	45 (18.75%)	123 (51.25%)	0 (0.00%)	72 (30.00%)
8	80 553	0 (0.00%)	17 093 (21.22%)	39 302	8 → 9	893	80 537 (99.98%)	9 (0.01%)	34	131	15 (0.02%)	7 (46.67%)	8 (53.33%)	0 (0.00%)	0 (0.00%)
9	80 843	0 (0.00%)	17 182 (21.25%)	39 430	9 → 10	1 425	80 517 (99.60%)	150 (0.19%)	193	166	314 (0.39%)	122 (38.85%)	145 (46.18%)	0 (0.00%)	47 (14.97%)
10	80 975	0 (0.00%)	17 182 (21.22%)	39 496	10 → 11	1 260	80 610 (99.55%)	237 (0.29%)	195	82	354 (0.44%)	100 (28.25%)	236 (66.67%)	0 (0.00%)	18 (5.08%)
11	80 921	0 (0.00%)	17 160 (21.21%)	39 480	11 → 12	1 040	80 617 (99.62%)	106 (0.13%)	267	124	303 (0.37%)	183 (60.40%)	100 (33.00%)	0 (0.00%)	20 (6.60%)
12	81 043	0 (0.00%)	17 160 (21.17%)	39 547	12 → 13	1 425	81 008 (99.96%)	9 (0.01%)	60	278	34 (0.04%)	26 (76.47%)	8 (23.53%)	0 (0.00%)	0 (0.00%)
13	81 539	0 (0.00%)	17 274 (21.18%)	39 821	13 → 14	8 686	80 084 (98.22%)	569 (0.70%)	742	900	1 285 (1.58%)	587 (45.68%)	698 (54.32%)	0 (0.00%)	0 (0.00%)
14	83 116	0 (0.00%)	17 504 (21.06%)	40 480	14 → 15	1 518	82 900 (99.74%)	84 (0.10%)	108	168	217 (0.26%)	143 (65.90%)	72 (33.18%)	0 (0.00%)	2 (0.92%)
15	83 373	0 (0.00%)	17 593 (21.10%)	40 589											

A.1.3 SDF3

Incremental parsing measurements for the SDF3 language.

Source	Parse Nodes			Character Nodes Count	Source	Parse Nodes			Shift		Breakdown				
	Count	Ambi- guous	Irre- usable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irre- usable	No Actions	Wrong State
Average	110 664	1.01%	41.36%	57 095	Average	5 725	98.21%	1.21%	279	778	1.57%	19.35%	78.28%	0.00%	2.38%
NaBL	194 597	0.88%	39.72%	98 595	NaBL	7 174	99.40%	0.37%	374	912	0.44%	18.74%	79.14%	0.00%	2.12%
DynSem	20 549	1.23%	40.08%	9 680	DynSem	2 200	98.15%	1.25%	132	300	1.72%	20.64%	78.00%	0.00%	1.36%
FlowSpec	24 064	1.07%	48.77%	12 079	FlowSpec	6 187	95.14%	3.04%	254	798	4.05%	21.18%	77.06%	0.00%	1.77%
Stratego	151 528	0.72%	36.82%	79 123	Stratego	6 874	99.25%	0.62%	303	864	0.73%	24.81%	69.59%	0.00%	5.60%
WebDSL	162 581	1.14%	41.39%	85 998	WebDSL	6 191	99.10%	0.79%	334	1 017	0.89%	11.37%	87.58%	0.00%	1.05%

Incremental parsing measurements for SDF3 source NaBL.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Count	Breakdown			
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node		Contains Change	Irreusable	No Actions	Wrong State
Average (0..14)	194 597	1 704 (0.88%)	77 292 (39.72%)	98 595	Average (1..15)	7 174	193 424 (99.40%)	731 (0.37%)	374	912	866 (0.44%)	91 (18.74%)	761 (79.14%)	0 (0.00%)	13 (2.12%)
0	190 950	1 675 (0.88%)	75 884 (39.74%)	96 540	→ 0	512 465			0	96 664					
1	191 566	1 681 (0.88%)	76 083 (39.72%)	96 879	0 → 1	5 005	190 304 (99.66%)	561 (0.29%)	278	763	644 (0.34%)	49 (7.61%)	588 (91.30%)	0 (0.00%)	7 (1.09%)
2	191 211	1 677 (0.88%)	75 900 (39.69%)	96 705	1 → 2	1 335	190 919 (99.66%)	262 (0.14%)	222	125	374 (0.20%)	136 (36.36%)	235 (62.83%)	0 (0.00%)	3 (0.80%)
3	191 212	1 677 (0.88%)	75 901 (39.69%)	96 706	2 → 3	933	191 006 (99.89%)	156 (0.08%)	191	134	207 (0.11%)	38 (18.36%)	169 (81.64%)	0 (0.00%)	0 (0.00%)
4	194 181	1 707 (0.88%)	77 100 (39.71%)	98 475	3 → 4	11 041	190 313 (99.53%)	101 (0.05%)	184	2 217	275 (0.14%)	165 (60.00%)	108 (39.27%)	0 (0.00%)	2 (0.73%)
5	194 177	1 707 (0.88%)	77 096 (39.70%)	98 471	4 → 5	658	194 024 (99.92%)	139 (0.07%)	179	85	160 (0.08%)	37 (23.13%)	123 (76.88%)	0 (0.00%)	0 (0.00%)
6	194 255	1 707 (0.88%)	77 096 (39.69%)	98 540	5 → 6	1 492	193 915 (99.87%)	245 (0.13%)	165	234	261 (0.13%)	10 (3.83%)	251 (96.17%)	0 (0.00%)	0 (0.00%)
7	194 265	1 707 (0.88%)	77 099 (39.69%)	98 556	6 → 7	13 604	192 231 (98.96%)	1 797 (0.93%)	741	1 236	2 023 (1.04%)	37 (1.83%)	1 967 (97.23%)	0 (0.00%)	19 (0.94%)
8	194 617	1 717 (0.88%)	77 231 (39.68%)	98 731	7 → 8	1 832	194 072 (99.90%)	145 (0.07%)	188	281	192 (0.10%)	30 (15.63%)	157 (81.77%)	0 (0.00%)	5 (2.60%)
9	194 634	1 717 (0.88%)	77 231 (39.68%)	98 742	8 → 9	1 149	194 411 (99.89%)	167 (0.09%)	192	120	205 (0.11%)	22 (10.73%)	173 (84.39%)	0 (0.00%)	10 (4.88%)
10	197 569	1 717 (0.87%)	78 553 (39.76%)	100 108	9 → 10	33 184	188 957 (97.08%)	2 682 (1.38%)	1 091	4 545	3 105 (1.60%)	306 (9.86%)	2 760 (88.89%)	0 (0.00%)	39 (1.26%)
11	197 573	1 717 (0.87%)	78 551 (39.76%)	100 110	10 → 11	15 001	195 195 (98.80%)	2 002 (1.01%)	792	1 452	2 238 (1.13%)	71 (3.17%)	2 140 (95.62%)	0 (0.00%)	27 (1.21%)
12	197 557	1 717 (0.87%)	78 553 (39.76%)	100 104	11 → 12	15 405	195 000 (98.70%)	1 964 (0.99%)	789	1 564	2 220 (1.12%)	105 (4.73%)	2 091 (94.19%)	0 (0.00%)	24 (1.08%)
13	197 574	1 717 (0.87%)	78 553 (39.76%)	100 115	12 → 13	1 149	197 351 (99.90%)	167 (0.08%)	192	120	205 (0.10%)	22 (10.73%)	173 (84.39%)	0 (0.00%)	10 (4.88%)
14	197 617	1 717 (0.87%)	78 553 (39.75%)	100 148	13 → 14	2 793	196 875 (99.65%)	242 (0.12%)	202	394	337 (0.17%)	118 (35.01%)	200 (59.35%)	0 (0.00%)	19 (5.64%)
15	197 574	1 717 (0.87%)	78 553 (39.76%)	100 115	14 → 15	3 036	196 783 (99.58%)	340 (0.17%)	207	410	539 (0.27%)	216 (40.07%)	287 (53.25%)	0 (0.00%)	36 (6.68%)

Incremental parsing measurements for SDF3 source DynSem.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average (0..14)	20 549	253 (1.23%)	8 237 (40.08%)	9 680	Average (1..15)	2 200	20 169 (98.15%)	256 (1.25%)	132	300	353 (1.72%)	75 (20.64%)	273 (78.00%)	0 (0.00%)	5 (1.36%)
					→ 0	50 935			0	9 070					
0	19 331	239 (1.24%)	7 763 (40.16%)	9 066	0 → 1	4 425	18 864 (97.58%)	349 (1.81%)	180	655	466 (2.41%)	86 (18.45%)	373 (80.04%)	0 (0.00%)	7 (1.50%)
1	20 227	252 (1.25%)	8 091 (40.00%)	9 540	1 → 2	2 218	19 750 (97.64%)	340 (1.68%)	196	200	476 (2.35%)	86 (18.07%)	387 (81.30%)	0 (0.00%)	3 (0.63%)
2	20 241	252 (1.24%)	8 088 (39.96%)	9 550	2 → 3	1 061	20 056 (99.09%)	155 (0.77%)	85	167	186 (0.92%)	32 (17.20%)	148 (79.57%)	0 (0.00%)	6 (3.23%)
3	20 231	251 (1.24%)	8 083 (39.95%)	9 546	3 → 4	1 875	19 851 (98.12%)	288 (1.42%)	133	165	379 (1.87%)	44 (11.61%)	333 (87.86%)	0 (0.00%)	2 (0.53%)
4	20 233	251 (1.24%)	8 083 (39.95%)	9 548	4 → 5	1 857	19 847 (98.09%)	326 (1.61%)	138	174	387 (1.91%)	42 (10.85%)	331 (85.53%)	0 (0.00%)	14 (3.62%)
5	20 222	250 (1.24%)	8 077 (39.94%)	9 544	5 → 6	1 493	19 835 (98.09%)	217 (1.07%)	124	225	384 (1.90%)	176 (45.83%)	204 (53.13%)	0 (0.00%)	4 (1.04%)
6	20 119	249 (1.24%)	8 008 (39.80%)	9 470	6 → 7	3 458	19 538 (97.11%)	334 (1.66%)	211	540	478 (2.38%)	109 (22.80%)	368 (76.99%)	0 (0.00%)	1 (0.21%)
7	20 438	254 (1.24%)	8 160 (39.93%)	9 652	7 → 8	1 718	20 260 (99.13%)	135 (0.66%)	93	285	177 (0.87%)	21 (11.86%)	152 (85.88%)	0 (0.00%)	4 (2.26%)
8	20 683	255 (1.23%)	8 285 (40.06%)	9 759	8 → 9	903	20 140 (97.37%)	148 (0.72%)	78	173	223 (1.08%)	104 (46.64%)	118 (52.91%)	0 (0.00%)	1 (0.45%)
9	20 304	249 (1.23%)	8 119 (39.99%)	9 550	9 → 10	1 789	20 146 (99.22%)	121 (0.60%)	82	323	157 (0.77%)	20 (12.74%)	136 (86.62%)	0 (0.00%)	1 (0.64%)
10	20 650	251 (1.22%)	8 275 (40.07%)	9 711	10 → 11	4 180	20 061 (97.15%)	398 (1.93%)	159	603	605 (2.93%)	180 (29.75%)	407 (67.27%)	0 (0.00%)	18 (2.98%)
11	21 180	256 (1.21%)	8 540 (40.32%)	9 968	11 → 12	1 736	20 926 (98.80%)	192 (0.91%)	101	263	253 (1.19%)	36 (14.23%)	210 (83.00%)	0 (0.00%)	7 (2.77%)
12	21 312	259 (1.22%)	8 600 (40.35%)	10 027	12 → 13	2 133	20 826 (97.72%)	367 (1.72%)	166	199	487 (2.29%)	80 (16.43%)	406 (83.37%)	0 (0.00%)	1 (0.21%)
13	21 312	259 (1.22%)	8 602 (40.36%)	10 027	13 → 14	2 630	20 990 (98.49%)	234 (1.10%)	148	360	321 (1.51%)	66 (20.56%)	254 (79.13%)	0 (0.00%)	1 (0.31%)
14	21 750	267 (1.23%)	8 785 (40.39%)	10 248	14 → 15	1 525	21 440 (98.57%)	239 (1.10%)	93	166	311 (1.43%)	39 (12.54%)	272 (87.46%)	0 (0.00%)	0 (0.00%)
15	21 750	267 (1.23%)	8 785 (40.39%)	10 248											

Incremental parsing measurements for SDF3 source FlowSpec.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average (0..14)	24 064	259 (1.07%)	11 733 (48.77%)	12 079	Average (1..15)	6 187	22 937 (95.14%)	695 (3.04%)	254	798	925 (4.05%)	211 (21.18%)	695 (77.06%)	0 (0.00%)	19 (1.77%)
					→ 0	59 603			0	10 180					
0	19 941	210 (1.05%)	9 753 (48.91%)	10 166	0 → 1	10 431	17 770 (89.11%)	1 408 (7.06%)	474	1 169	2 072 (10.39%)	533 (25.72%)	1 520 (73.36%)	0 (0.00%)	19 (0.92%)
1	20 465	215 (1.05%)	9 920 (48.47%)	10 398	1 → 2	5 162	19 646 (96.00%)	653 (3.19%)	271	583	816 (3.99%)	103 (12.62%)	691 (84.68%)	0 (0.00%)	22 (2.70%)
2	20 853	219 (1.05%)	10 127 (48.56%)	10 584	2 → 3	7 087	19 710 (94.52%)	820 (3.93%)	254	883	1 150 (5.51%)	246 (21.39%)	889 (77.30%)	0 (0.00%)	15 (1.30%)
3	21 591	230 (1.07%)	10 563 (48.92%)	10 931	3 → 4	14 153	19 357 (89.65%)	1 738 (8.05%)	605	1 656	2 227 (10.31%)	384 (17.24%)	1 787 (80.24%)	0 (0.00%)	56 (2.51%)
4	23 044	248 (1.08%)	11 312 (49.09%)	11 530	4 → 5	7 615	22 006 (95.50%)	852 (3.70%)	355	845	1 036 (4.50%)	111 (10.71%)	891 (86.00%)	0 (0.00%)	34 (3.28%)
5	23 832	263 (1.10%)	11 690 (49.05%)	11 891	5 → 6	8 917	22 702 (95.26%)	797 (3.34%)	408	1 140	1 139 (4.78%)	338 (29.68%)	764 (67.08%)	0 (0.00%)	37 (3.25%)
6	25 088	275 (1.10%)	12 362 (49.27%)	12 522	6 → 7	6 956	23 570 (93.95%)	1 113 (4.44%)	426	785	1 439 (5.74%)	261 (18.14%)	1 120 (77.83%)	0 (0.00%)	58 (4.03%)
7	25 250	275 (1.09%)	12 372 (49.00%)	12 580	7 → 8	920	24 957 (98.84%)	218 (0.86%)	86	116	268 (1.06%)	85 (31.72%)	183 (68.28%)	0 (0.00%)	0 (0.00%)
8	25 181	273 (1.08%)	12 345 (49.03%)	12 551	8 → 9	926	24 994 (99.26%)	133 (0.53%)	62	114	186 (0.74%)	30 (16.13%)	154 (82.80%)	0 (0.00%)	2 (1.08%)
9	25 239	273 (1.08%)	12 369 (49.01%)	12 582	9 → 10	1 948	24 817 (98.33%)	393 (1.56%)	122	282	421 (1.67%)	11 (2.61%)	395 (93.82%)	0 (0.00%)	15 (3.56%)
10	25 304	274 (1.08%)	12 402 (49.01%)	12 620	10 → 11	670	25 216 (99.65%)	56 (0.22%)	63	86	87 (0.34%)	21 (24.14%)	66 (75.86%)	0 (0.00%)	0 (0.00%)
11	25 373	275 (1.08%)	12 435 (49.01%)	12 659	11 → 12	857	25 179 (99.24%)	141 (0.56%)	69	94	201 (0.79%)	52 (25.87%)	147 (73.13%)	0 (0.00%)	2 (1.00%)
12	25 368	275 (1.08%)	12 430 (49.00%)	12 654	12 → 13	20 204	20 515 (80.87%)	1 437 (5.66%)	427	3 248	2 037 (8.03%)	863 (42.37%)	1 156 (56.75%)	0 (0.00%)	18 (0.88%)
13	26 820	283 (1.06%)	12 771 (47.62%)	13 540	13 → 14	5 218	26 220 (97.76%)	530 (1.98%)	98	693	599 (2.23%)	63 (10.52%)	530 (88.48%)	0 (0.00%)	6 (1.00%)
14	27 616	295 (1.07%)	13 137 (47.57%)	13 980	14 → 15	1 741	27 397 (99.21%)	143 (0.52%)	92	280	198 (0.72%)	57 (28.79%)	139 (70.20%)	0 (0.00%)	2 (1.01%)
15	28 023	295 (1.05%)	13 248 (47.28%)	14 154											

Incremental parsing measurements for SDF3 source Stratego.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Breakdown				
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node	Count	Contains Change	Irreusable	No Actions	Wrong State
Average (9..14)	151 528	1 094 (0.72%)	55 800 (36.82%)	79 123	Average (10..15)	6 874	150 400 (99.25%)	933 (0.62%)	303	864	1 108 (0.73%)	171 (24.81%)	879 (69.59%)	0 (0.00%)	58 (5.60%)
9	150 091	1 080 (0.72%)	55 045 (36.67%)	78 291	→ 9	401 368	149 947 (99.90%)	31 (0.02%)	0	78 354	132 (0.09%)	110 (83.33%)	22 (16.67%)	0 (0.00%)	0 (0.00%)
10	150 138	1 080 (0.72%)	55 087 (36.69%)	78 333	9 → 10	402	147 456 (98.21%)	2 250 (1.50%)	74	122	2 696 (1.80%)	323 (11.98%)	2 330 (86.42%)	0 (0.00%)	43 (1.59%)
11	151 550	1 096 (0.72%)	55 773 (36.80%)	79 231	10 → 11	19 726	151 382 (99.89%)	153 (0.10%)	476	2 558	167 (0.11%)	15 (8.98%)	127 (76.05%)	0 (0.00%)	25 (14.97%)
12	151 908	1 097 (0.72%)	55 978 (36.85%)	79 378	11 → 12	2 406	151 137 (99.49%)	572 (0.38%)	109	209	750 (0.49%)	109 (14.53%)	590 (78.67%)	0 (0.00%)	51 (6.80%)
13	152 430	1 100 (0.72%)	56 287 (36.93%)	79 672	12 → 13	4 650	149 645 (98.17%)	2 431 (1.59%)	231	634	2 681 (1.76%)	438 (16.34%)	2 018 (75.27%)	0 (0.00%)	225 (8.39%)
14	153 053	1 108 (0.72%)	56 632 (37.00%)	79 832	13 → 14	13 306	152 835 (99.86%)	163 (0.11%)	804	1 562	219 (0.14%)	30 (13.70%)	185 (84.47%)	0 (0.00%)	4 (1.83%)
15	153 054	1 108 (0.72%)	56 633 (37.00%)	79 833	14 → 15	756			125	96					

Incremental parsing measurements for SDF3 source WebDSL.

Version	Parse Nodes			Character Nodes Count	Version	Parse Nodes			Shift		Count	Breakdown			
	Count	Ambiguous	Irreusable			Created	Reused	Rebuilt	Parse Node	Character Node		Contains Change	Irreusable	No Actions	Wrong State
Average (0..14)	162 581	1 853 (1.14%)	67 285 (41.39%)	85 998	Average (1..15)	6 191	161 117 (99.10%)	1 287 (0.79%)	334	1 017	1 444 (0.89%)	99 (11.37%)	1 331 (87.58%)	0 (0.00%)	14 (1.05%)
					→ 0	422 079			0	85 526					
0	161 682	1 829 (1.13%)	67 013 (41.45%)	85 500	0 → 1	1 074	161 420 (99.84%)	87 (0.05%)	82	169	263 (0.16%)	175 (66.54%)	88 (33.46%)	0 (0.00%)	0 (0.00%)
1	161 735	1 829 (1.13%)	67 020 (41.44%)	85 515	1 → 2	2 491	161 488 (99.85%)	190 (0.12%)	115	426	246 (0.15%)	34 (13.82%)	211 (85.77%)	0 (0.00%)	1 (0.41%)
2	162 255	1 837 (1.13%)	67 200 (41.42%)	85 783	2 → 3	16 085	158 693 (97.80%)	3 212 (1.98%)	967	2 217	3 558 (2.19%)	131 (3.68%)	3 382 (95.05%)	0 (0.00%)	45 (1.26%)
3	162 345	1 838 (1.13%)	67 260 (41.43%)	85 846	3 → 4	7 615	160 514 (98.87%)	1 704 (1.05%)	434	1 254	1 830 (1.13%)	53 (2.90%)	1 771 (96.78%)	0 (0.00%)	6 (0.33%)
4	162 361	1 838 (1.13%)	67 260 (41.43%)	85 856	4 → 5	8 149	160 343 (98.76%)	1 771 (1.09%)	446	1 323	1 982 (1.22%)	153 (7.72%)	1 821 (91.88%)	0 (0.00%)	8 (0.40%)
5	162 369	1 841 (1.13%)	67 220 (41.40%)	85 844	5 → 6	4 912	161 016 (99.17%)	1 196 (0.74%)	277	861	1 277 (0.79%)	73 (5.72%)	1 196 (93.66%)	0 (0.00%)	8 (0.63%)
6	162 273	1 840 (1.13%)	67 184 (41.40%)	85 799	6 → 7	8 917	159 740 (98.44%)	1 845 (1.14%)	479	1 655	2 399 (1.48%)	457 (19.05%)	1 874 (78.12%)	0 (0.00%)	68 (2.83%)
7	162 457	1 856 (1.14%)	67 210 (41.37%)	85 931	7 → 8	16 962	158 781 (97.74%)	3 490 (2.15%)	528	2 532	3 675 (2.26%)	29 (0.79%)	3 642 (99.10%)	0 (0.00%)	4 (0.11%)
8	162 615	1 857 (1.14%)	67 304 (41.39%)	86 050	8 → 9	2 066	162 256 (99.78%)	326 (0.20%)	110	375	358 (0.22%)	23 (6.42%)	334 (93.30%)	0 (0.00%)	1 (0.28%)
9	162 878	1 864 (1.14%)	67 403 (41.38%)	86 167	9 → 10	7 783	160 967 (98.83%)	1 751 (1.08%)	445	1 263	1 897 (1.16%)	67 (3.53%)	1 826 (96.26%)	0 (0.00%)	4 (0.21%)
10	162 841	1 864 (1.14%)	67 376 (41.38%)	86 154	10 → 11	2 079	162 477 (99.78%)	332 (0.20%)	108	398	363 (0.22%)	18 (4.96%)	345 (95.04%)	0 (0.00%)	0 (0.00%)
11	163 063	1 868 (1.15%)	67 436 (41.36%)	86 285	11 → 12	2 814	162 257 (99.51%)	595 (0.36%)	189	500	797 (0.49%)	149 (18.70%)	607 (76.16%)	0 (0.00%)	41 (5.14%)
12	163 158	1 876 (1.15%)	67 445 (41.34%)	86 330	12 → 13	1 632	162 778 (99.77%)	324 (0.20%)	132	281	379 (0.23%)	34 (8.97%)	331 (87.34%)	0 (0.00%)	14 (3.69%)
13	163 210	1 876 (1.15%)	67 445 (41.32%)	86 370	13 → 14	1 606	162 913 (99.82%)	279 (0.17%)	99	412	296 (0.18%)	13 (4.39%)	283 (95.61%)	0 (0.00%)	0 (0.00%)
14	163 466	1 876 (1.15%)	67 495 (41.29%)	86 543	14 → 15	8 683	161 111 (98.56%)	2 208 (1.35%)	605	1 591	2 337 (1.43%)	78 (3.34%)	2 248 (96.19%)	0 (0.00%)	11 (0.47%)
15	163 466	1 876 (1.15%)	67 495 (41.29%)	86 538											

A.2 Time Benchmarks

The tables on the left show parse times for only the parsing phase, while the tables on the right show parse times for the full JSGLR2 parsing pipeline, which includes imploding and tokenization, as shown in the bottom row of Figures 2.10 and 3.12. All times are measured in milliseconds. The tables in the middle show the size of the input and the number of added/removed characters for each version.

Average parse time for all languages, excluding version 0.

Language	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Language	Standard	Elkhound	Incremental (no cache)	Incremental	Tree-sitter (no cache)	Tree-sitter
Average	785.510	597.235	839.299	82.503	336 635	682	816	Average	810.266	630.811	1 007.738	90.302	-	-
Java	1 974.375	1 497.341	2 112.104	203.043	851 080	1 757	1 725	Java	2 033.161	1 579.048	2 518.010	217.591	233.929	19.715
WebDSL	233.029	175.540	244.903	23.681	101 548	205	458	WebDSL	241.125	185.905	303.368	27.597	-	-
SDF3	149.127	118.824	160.888	20.784	57 277	83	265	SDF3	156.514	127.480	201.835	25.719	-	-

A.2.1 Java

Average parse times for the Java language, excluding version 0.

Source	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Source	Standard	Elkhound	Incremental (no cache)	Incremental	Tree-sitter (no cache)	Tree-sitter
Average	1 974.375	1 497.341	2 112.104	203.043	851 080	1 757	1 725	Average	2 033.161	1 579.048	2 518.010	217.591	233.929	19.715
StringUtils	548.470	406.090	576.152	163.664	396 376	369	481	StringUtils	580.587	436.086	692.993	170.913	54.080	21.040
gson	3 530.745	2 748.975	3 775.883	289.502	1 265 099	344	644	gson	3 626.562	2 866.369	4 530.654	301.643	434.953	20.072
slf4j	1 843.908	1 336.959	1 984.278	155.963	891 764	4 557	4 049	slf4j	1 892.333	1 434.687	2 330.383	180.218	212.755	18.034

Parse times for Java source StringUtils.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental	Tree-sitter (no cache)	Tree-sitter
Average	548.470	406.090	576.152	163.664	396 376	369	481	Average	580.587	436.086	692.993	170.913	54.080	21.040
0	510.245	410.015	607.831	567.981	395 110	-	-	0	613.234	447.702	685.152	677.608	53.953	55.989
1	569.867	412.210	560.635	203.766	395 172	0	62	1	589.096	430.918	665.578	204.115	54.375	5.190
2	571.769	406.264	579.527	137.604	395 216	0	44	2	525.333	442.995	670.220	133.418	54.245	14.311
3	512.182	399.330	573.533	242.027	394 856	515	155	3	596.581	444.350	702.911	253.706	55.092	54.925
4	517.057	418.798	591.289	166.918	397 653	0	2 797	4	598.532	442.535	673.724	175.444	54.990	5.662
5	560.044	408.426	565.200	274.735	395 931	3 940	2 218	5	589.638	431.712	749.065	296.903	53.480	77.045
6	565.144	403.616	582.418	43.652	395 896	35	0	6	553.568	435.171	710.847	44.800	53.708	2.056
7	574.693	401.232	559.943	105.395	395 842	64	10	7	580.305	429.797	683.152	109.176	53.764	5.572
8	563.347	410.130	564.563	104.429	395 842	7	7	8	557.878	431.062	684.287	110.730	54.037	5.479
9	556.038	405.553	584.746	63.173	397 158	0	1 316	9	554.663	426.705	678.664	67.274	53.735	5.411
10	558.353	408.050	557.377	142.577	397 176	0	18	10	625.021	437.813	685.739	151.585	54.321	6.441
11	526.810	400.134	636.828	201.492	397 184	0	8	11	587.075	443.293	678.918	220.436	53.922	11.363
12	544.163	398.685	562.959	192.542	397 048	136	0	12	577.435	442.426	733.281	198.861	53.780	23.409
13	535.120	410.863	596.468	99.104	397 077	0	29	13	598.358	440.583	670.686	96.101	53.761	16.372
14	541.432	397.419	553.625	289.401	396 792	754	469	14	620.832	429.453	690.750	309.318	54.291	71.858
15	531.037	410.638	573.166	188.143	396 796	80	84	15	554.494	432.485	717.076	191.827	53.704	10.500

Parse times for Java source gson.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental	Tree-sitter (no cache)	Tree-sitter
Average	3 530.745	2 748.975	3 775.883	289.502	1 265 099	344	644	Average	3 626.562	2 866.369	4 530.654	301.643	434.953	20.072
0	3 114.155	2 763.787	3 540.657	3 473.504	1 263 538	0	1 263 518	0	3 533.697	2 877.887	4 453.189	4 786.196	432.136	443.473
1	3 618.639	2 701.489	3 955.413	287.628	1 263 386	573	421	1	3 278.285	2 699.250	4 267.664	303.627	447.886	19.127
2	3 602.246	2 692.837	3 936.817	285.333	1 263 310	98	22	2	3 822.314	2 858.514	4 484.484	289.680	438.720	16.571
3	3 678.500	2 726.585	3 704.812	286.764	1 263 306	24	20	3	3 386.830	2 883.233	4 454.842	293.020	431.946	19.386
4	3 135.207	2 719.616	3 949.931	304.329	1 262 894	1 083	671	4	3 821.525	2 875.712	4 656.754	353.911	433.808	34.124
5	3 628.444	2 720.835	3 528.902	274.734	1 262 894	0	0	5	3 816.020	2 878.528	4 804.442	294.436	434.877	15.768
6	3 647.294	2 895.733	3 862.009	270.388	1 263 022	0	126	6	3 900.653	2 859.448	4 307.176	288.156	431.076	20.862
7	3 379.192	2 819.992	3 591.335	314.200	1 262 846	302	126	7	3 797.019	2 912.520	4 339.391	307.115	439.001	21.316
8	3 153.701	2 787.409	3 645.922	276.597	1 263 220	0	374	8	3 459.621	3 009.576	4 827.296	285.400	431.967	18.220
9	3 737.047	2 699.772	3 786.667	286.849	1 263 298	25	103	9	3 361.746	2 857.024	4 297.740	299.716	429.407	20.830
10	3 695.981	2 772.365	3 690.220	310.451	1 267 896	0	4 598	10	3 464.597	2 823.515	4 449.837	309.913	433.013	21.895
11	3 280.822	2 805.696	3 634.455	288.995	1 268 060	0	164	11	3 370.940	2 872.003	4 797.996	289.960	436.452	18.685
12	3 712.268	2 772.718	3 755.381	290.828	1 268 144	0	84	12	3 820.651	2 819.557	4 560.324	290.045	434.129	18.649
13	3 395.237	2 652.254	3 952.558	289.544	1 267 762	1 135	753	13	3 864.571	2 873.621	4 452.984	293.021	436.982	18.472
14	3 666.518	2 740.640	3 678.815	296.714	1 268 412	788	1 438	14	3 822.406	2 944.329	4 553.043	312.105	433.934	19.199
15	3 630.081	2 726.681	3 965.007	279.173	1 268 030	1 135	753	15	3 411.260	2 828.712	4 705.833	314.536	431.090	17.972

Parse times for Java source slf4j.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental	Tree-sitter (no cache)	Tree-sitter
Average	1843.908	1336.959	1984.278	155.963	891764	4557	4049	Average	1892.333	1434.687	2330.383	180.218	212.755	18.034
0	1950.869	1396.467	1924.703	1895.252	912784	-	-	0	1962.037	1461.973	2301.925	2429.781	217.761	219.257
1	1914.720	1345.672	2191.663	160.343	912785	0	1	1	1949.528	1453.325	2256.715	172.429	217.246	8.911
2	1910.553	1395.761	1905.263	135.988	912813	8	36	2	1780.389	1411.431	2238.574	143.523	216.544	10.035
3	1688.898	1418.170	1927.477	140.715	912814	0	1	3	2086.934	1446.944	2467.858	151.674	219.741	10.524
4	1952.318	1273.339	2138.546	178.288	913852	8814	9852	4	1937.893	1484.835	2524.441	268.604	218.903	25.982
5	1913.849	1302.889	1963.789	225.093	902914	20902	9964	5	2009.672	1558.417	2273.049	326.079	215.636	67.605
6	1887.770	1391.150	1883.856	135.516	902912	6	4	6	1942.201	1467.338	2490.050	142.852	214.497	10.747
7	1977.712	1361.097	1914.908	226.041	863562	37270	19602	7	1741.697	1503.599	2298.376	290.557	212.140	41.282
8	1625.192	1370.718	2067.581	180.761	881191	671	18300	8	1853.447	1406.083	2182.388	201.929	208.579	20.808
9	1879.117	1250.072	1861.661	132.147	881184	7	0	9	1728.795	1394.955	2292.012	137.419	209.333	10.251
10	1851.934	1343.630	2096.098	135.887	881132	52	0	10	1897.723	1409.310	2211.121	140.060	207.715	10.656
11	1853.402	1242.876	1855.218	145.600	881134	5	7	11	1993.814	1397.318	2475.115	153.019	210.503	8.233
12	1878.930	1329.713	2131.885	134.209	881518	16	400	12	1911.829	1446.683	2280.729	143.075	208.636	11.411
13	1883.879	1331.258	1904.392	135.011	881618	2	102	13	1833.444	1434.646	2300.400	136.817	209.983	7.986
14	1856.527	1327.615	1775.771	138.090	883547	202	2131	14	1701.047	1295.931	2489.464	151.905	210.005	13.943
15	1583.828	1370.429	2146.060	135.751	883490	399	342	15	2016.578	1409.487	2175.459	143.323	211.867	12.134

A.2.2 WebDSL

Average parse times for the WebDSL language, excluding version 0.

Source	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Source	Standard	Elkhound	Incremental (no cache)	Incremental
Average	233.029	175.540	244.903	23.681	101548	205	458	Average	241.125	185.905	303.368	27.597
builtin.app	249.465	185.023	263.670	27.441	97843	475	174	builtin.app	267.130	196.868	334.089	31.030
YellowGrass	360.934	270.288	375.227	34.731	167287	107	1060	YellowGrass	364.214	285.688	458.316	40.419
elib-utils	88.690	71.310	95.813	8.871	39515	32	141	elib-utils	92.029	75.159	117.699	11.342

Parse times for WebDSL source builtin.app.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental
Average	249.465	185.023	263.670	27.441	97 843	475	174	Average	267.130	196.868	334.089	31.030
1	266.187	191.839	278.880	270.626	100 882	-	-	1	279.817	196.008	340.347	342.084
2	262.139	185.533	277.219	25.195	97 219	3 663	0	2	268.534	202.189	324.073	26.479
3	223.751	179.368	262.086	27.702	98 047	0	828	3	279.992	202.742	344.977	38.476
4	248.952	180.288	259.710	26.949	98 231	10	194	4	263.362	187.933	352.829	28.644
5	250.799	192.804	254.144	36.115	97 627	604	0	5	261.398	191.781	332.648	45.938
6	256.824	182.942	259.105	24.456	97 683	0	56	6	265.544	191.989	317.260	25.824
7	244.860	184.720	279.631	26.242	97 721	0	38	7	271.876	201.561	351.438	28.053
8	246.218	187.573	268.891	27.521	97 935	0	214	8	262.419	196.931	326.250	29.291
9	260.944	184.491	252.059	25.631	97 956	0	21	9	264.730	202.284	333.181	27.165
10	250.694	187.492	260.181	27.155	98 172	1	217	10	266.317	194.397	324.146	29.401

Parse times for WebDSL source YellowGrass.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental
Average	360.934	270.288	375.227	34.731	167 287	107	1 060	Average	364.214	285.688	458.316	40.419
0	326.021	258.895	382.747	375.826	158 975	-	-	0	322.583	270.795	429.477	431.878
1	350.203	269.790	381.338	37.656	160 568	121	1 714	1	360.298	290.884	427.663	50.274
2	349.087	268.371	374.891	43.228	159 970	954	356	2	318.511	269.640	434.078	61.583
3	350.056	256.040	348.459	32.232	160 445	67	542	3	362.199	275.319	433.132	44.342
4	349.585	268.767	361.169	40.465	161 720	38	1 313	4	359.778	277.303	421.927	55.434
5	340.806	261.018	383.822	30.672	161 716	4	0	5	368.950	278.087	466.042	31.967
6	358.853	258.436	351.155	32.425	163 359	0	1 643	6	354.989	280.019	437.152	34.685
7	371.666	264.268	347.987	28.763	163 370	12	23	7	372.142	274.603	437.677	30.182
8	371.762	274.609	393.565	40.464	168 635	0	5 265	8	335.676	286.318	469.427	44.186
9	368.866	277.560	352.047	39.598	172 696	0	4 061	9	358.421	297.573	446.813	43.347
10	369.962	280.473	399.384	29.453	172 793	32	129	10	374.248	299.657	494.391	32.145
11	373.611	280.305	376.083	31.326	172 833	32	72	11	390.910	295.862	508.930	31.577
12	373.763	262.900	370.201	31.007	172 834	0	1	12	389.511	289.188	452.468	31.168
13	331.360	275.490	408.462	31.922	172 557	281	4	13	391.335	297.690	494.219	33.675
14	382.947	270.806	409.803	30.669	172 529	67	39	14	340.434	290.551	465.037	32.109
15	371.483	285.486	370.034	41.082	173 274	0	745	15	385.813	282.629	485.790	49.609

Parse times for WebDSL source elib-utils.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental
Average	88.690	71.310	95.813	8.871	39 515	32	141	Average	92.029	75.159	117.699	11.342
0	82.423	70.155	89.681	95.225	38 955	-	-	0	96.551	71.086	110.582	114.429
1	87.218	68.698	98.440	7.603	38 953	2	0	1	92.814	75.213	113.256	8.016
2	82.132	69.742	91.284	9.371	39 118	0	165	2	86.217	72.883	112.333	12.733
3	89.226	69.606	98.881	10.049	39 211	66	159	3	91.201	73.849	122.497	13.913
4	89.641	71.085	87.965	8.119	39 378	0	167	4	93.397	75.025	124.901	10.583
5	90.753	71.517	93.305	8.514	39 317	94	33	5	95.497	73.691	114.172	9.534
6	90.943	72.859	98.665	8.662	39 283	48	14	6	93.696	77.906	119.379	11.018
7	92.183	70.864	95.146	8.058	39 320	1	38	7	94.806	74.913	110.154	8.675
8	82.246	71.391	94.134	8.571	39 302	21	3	8	92.301	73.878	120.606	9.304
9	91.406	70.044	93.005	8.001	39 430	0	128	9	93.427	74.362	117.196	8.808
10	90.550	72.145	94.159	8.697	39 496	29	95	10	86.160	74.077	114.960	13.685
11	86.594	72.144	94.190	9.121	39 480	22	6	11	94.842	77.152	117.428	11.032
12	89.672	71.642	100.388	8.367	39 547	2	69	12	93.339	74.075	114.295	9.201
13	83.455	71.824	93.306	8.454	39 821	0	274	13	95.111	78.341	116.424	10.879
14	91.709	72.245	102.869	12.852	40 480	169	828	14	87.856	75.412	127.476	21.259
15	92.622	73.845	101.460	8.625	40 589	22	131	15	89.771	76.603	120.402	11.486

A.2.3 SDF3

Average parse times for the SDF3 language, excluding version 0.

Source	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Source	Standard	Elkhound	Incremental (no cache)	Incremental
Average	149.127	118.824	160.888	20.784	57 277	83	265	Average	156.514	127.480	201.835	25.719
NaBL	248.609	194.725	268.438	33.179	98 834	184	422	NaBL	254.829	209.901	330.948	38.750
DynSem	26.006	21.133	28.662	5.145	9 759	25	104	DynSem	27.797	22.963	37.408	7.857
FlowSpec	35.278	29.035	39.252	8.577	12 345	125	391	FlowSpec	37.772	31.233	50.636	13.299
Stratego	231.338	183.947	251.471	26.424	79 380	61	318	Stratego	246.166	195.066	308.393	31.794
WebDSL	204.402	165.280	216.618	30.596	86 067	22	92	WebDSL	216.004	178.237	281.788	36.896

Parse times for SDF3 source NaBL.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental
Average	248.609	194.725	268.438	33.179	98 834	184	422	Average	254.829	209.901	330.948	38.750
0	250.544	192.285	271.866	249.973	96 540	-	-	0	260.355	209.091	319.534	341.853
1	251.955	194.495	251.967	30.628	96 879	0	339	1	265.994	214.581	317.569	33.395
2	240.936	193.368	255.114	28.513	96 705	174	0	2	257.144	207.690	337.709	30.296
3	244.266	191.811	253.297	27.862	96 706	3	4	3	229.444	202.478	307.133	29.336
4	244.303	198.509	259.455	33.187	98 475	341	2 110	4	269.891	211.170	341.943	36.723
5	253.614	199.143	280.289	27.593	98 471	4	0	5	263.225	203.466	342.024	28.480
6	254.523	187.106	287.502	28.362	98 540	0	69	6	236.581	212.924	344.119	29.309
7	248.624	193.144	274.419	40.047	98 556	0	16	7	232.504	220.040	351.583	54.574
8	255.298	193.190	247.621	27.635	98 731	0	175	8	237.903	211.939	303.800	29.732
9	242.818	200.243	260.494	28.209	98 742	0	11	9	232.127	209.990	321.503	29.496
10	306.991	195.278	286.163	55.790	100 108	1 465	2 831	10	270.070	209.106	352.497	73.302
11	256.088	190.641	284.046	42.526	100 110	104	106	11	260.686	212.197	339.839	56.342
12	245.111	196.228	283.476	41.294	100 104	248	242	12	270.553	207.202	313.593	57.086
13	230.777	200.295	277.397	27.543	100 115	0	11	13	273.302	205.698	349.267	28.707
14	232.957	192.485	250.312	29.751	100 148	191	224	14	272.789	209.727	325.196	31.764
15	220.869	194.939	275.021	28.744	100 115	224	191	15	250.227	210.304	316.438	32.708

Parse times for SDF3 source DynSem.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental
Average	26.006	21.133	28.662	5.145	9 759	25	104	Average	27.797	22.963	37.408	7.857
0	23.964	19.627	26.813	26.882	9 066	–	–	0	26.048	20.996	35.466	34.575
1	25.837	20.681	28.228	6.442	9 540	0	474	1	26.480	22.778	34.985	11.515
2	26.099	20.172	28.725	5.290	9 550	0	10	2	27.801	22.160	38.180	7.767
3	25.055	20.832	27.904	4.302	9 546	4	0	3	26.280	21.637	35.854	5.443
4	25.606	21.268	27.607	4.886	9 548	0	2	4	26.967	22.217	36.576	6.528
5	25.833	21.062	27.942	4.962	9 544	4	0	5	27.043	22.704	36.470	7.190
6	25.701	20.908	28.053	4.555	9 470	78	4	6	26.993	22.661	37.548	6.234
7	24.829	20.894	28.100	6.013	9 652	56	238	7	27.228	22.747	35.500	9.673
8	25.709	20.468	28.012	4.584	9 759	0	107	8	27.195	23.737	37.294	7.186
9	24.915	21.236	27.862	4.118	9 550	209	0	9	27.946	21.747	37.103	5.161
10	25.617	20.891	28.331	4.752	9 711	0	161	10	27.948	22.732	37.217	7.150
11	26.959	21.202	28.452	6.502	9 968	27	284	11	28.708	23.794	37.248	11.343
12	26.000	21.504	29.461	4.745	10 027	0	59	12	28.030	23.470	38.551	7.021
13	27.729	21.020	30.324	5.536	10 027	2	2	13	28.903	23.261	38.458	8.613
14	26.929	22.194	30.200	5.621	10 248	0	221	14	29.427	24.079	40.860	9.644
15	27.276	22.670	30.726	4.874	10 248	2	2	15	30.001	24.714	39.279	7.395

Parse times for SDF3 source FlowSpec.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental
Average	35.278	29.035	39.252	8.577	12 345	125	391	Average	37.772	31.233	50.636	13.299
0	29.827	24.026	33.622	33.251	10 166	-	-	0	31.252	25.072	41.035	41.284
1	29.923	24.145	32.734	10.640	10 398	76	308	1	31.045	25.964	40.726	16.745
2	30.332	24.278	33.491	7.185	10 584	0	186	2	31.887	26.411	40.593	10.672
3	31.568	26.343	33.947	8.389	10 931	31	378	3	33.944	28.244	43.793	14.631
4	33.100	26.475	36.851	13.340	11 530	46	645	4	35.116	30.201	49.271	27.023
5	34.484	28.446	38.581	9.606	11 891	0	361	5	36.672	30.289	49.327	15.180
6	35.335	29.165	40.731	10.228	12 522	17	648	6	38.860	31.028	53.332	16.547
7	35.644	30.645	41.263	9.998	12 580	72	130	7	38.130	31.224	51.229	14.669
8	35.244	30.601	40.648	5.494	12 551	29	0	8	38.145	31.944	54.883	6.419
9	37.907	29.822	40.323	5.311	12 582	0	31	9	39.160	32.925	49.874	7.507
10	35.821	29.734	40.161	6.497	12 620	0	38	10	39.768	32.681	54.161	9.018
11	36.921	30.450	40.689	5.377	12 659	0	39	11	38.310	31.982	51.647	6.029
12	36.599	29.350	40.453	5.273	12 654	8	3	12	39.467	33.015	51.115	7.454
13	37.272	31.347	41.948	16.568	13 540	1 581	2 467	13	41.886	33.651	53.122	23.345
14	37.864	31.579	42.971	8.415	13 980	0	440	14	41.807	34.052	57.361	15.078
15	41.159	33.146	43.984	6.336	14 154	14	188	15	42.385	34.883	59.102	9.165

Parse times for SDF3 source Stratego.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental
Average	231.338	183.947	251.471	26.424	79 380	61	318	Average	246.166	195.066	308.393	31.794
9	226.619	180.479	248.538	248.358	78 291	-	-	9	232.024	186.610	294.465	292.806
10	235.082	179.032	246.124	20.951	78 333	54	96	10	254.045	190.244	312.593	21.307
11	228.249	180.503	242.531	35.282	79 231	162	1 060	11	242.430	201.798	307.034	47.601
12	233.428	190.932	267.120	23.903	79 378	0	147	12	245.755	191.530	300.525	25.607
13	213.648	183.172	251.578	23.754	79 672	18	312	13	245.527	196.365	297.620	31.827
14	234.556	186.140	252.648	32.954	79 832	129	289	14	242.960	194.322	311.118	42.320
15	243.065	183.902	248.824	21.699	79 833	2	3	15	246.280	196.138	321.467	22.102

Parse times for SDF3 source WebDSL.

Version	Standard	Elkhound	Incremental (no cache)	Incremental	Size (B)	Removed (B)	Added (B)	Version	Standard	Elkhound	Incremental (no cache)	Incremental
Average	204.402	165.280	216.618	30.596	86 067	22	92	Average	216.004	178.237	281.788	36.896
0	200.194	165.581	215.281	227.837	85 500	-	-	0	217.891	171.354	281.202	267.256
1	204.197	166.431	207.700	26.615	85 515	52	67	1	200.142	175.458	295.214	27.340
2	189.313	164.352	220.025	27.370	85 783	0	268	2	217.900	176.373	275.369	29.527
3	197.230	164.831	207.987	39.591	85 846	0	63	3	218.666	184.393	274.712	49.673
4	200.683	163.307	215.856	31.261	85 856	0	10	4	220.651	180.516	295.892	51.704
5	199.334	163.215	230.702	31.604	85 844	50	38	5	217.303	178.121	295.963	41.833
6	207.198	166.165	218.716	29.377	85 799	45	0	6	210.368	179.741	280.381	32.282
7	208.579	166.837	211.050	33.062	85 931	114	246	7	209.642	178.047	275.611	44.323
8	208.245	164.092	207.910	36.819	86 050	0	119	8	220.227	177.271	279.323	49.866
9	206.942	164.201	209.997	26.767	86 167	0	117	9	219.935	178.604	277.716	29.116
10	206.028	166.243	213.583	31.967	86 154	17	4	10	222.944	172.873	270.160	40.494
11	206.340	167.478	234.758	30.051	86 285	0	131	11	216.337	179.912	269.427	29.112
12	215.316	163.458	209.008	27.314	86 330	24	69	12	217.212	176.450	293.319	31.444
13	206.024	165.222	234.109	26.943	86 370	0	40	13	222.033	177.199	270.401	27.974
14	199.102	169.170	213.681	26.868	86 543	0	173	14	212.739	177.661	280.448	28.142
15	211.497	164.197	214.195	33.323	86 538	35	30	15	213.956	180.937	292.892	40.610

Appendix B

SPLASH Conference 2019 – ACM Student Research Competition

In the week of 20–25 October 2019, I had the opportunity to present my work at the ACM Student Research Competition (SRC), during the SPLASH conference in Athens, Greece.¹ In the SRC held at SPLASH, I achieved first place in the category of graduate (Master and PhD) students. The competition consisted of several rounds, with several deliverables, which have been listed below.

Call for Submissions: Extended Abstract To participate in the SRC, participants had to submit an extended abstract of no more than 800 words and no more than 2 pages (excluding references). Appendix B.1 contains my submission, which is also published in the Proceedings Companion of SPLASH 2019.² After review, eight participants were invited to present their work at the conference, of which four undergraduate and four graduate students.

First Round: Poster On Wednesday the 23rd, the eight participants presented a poster showing their work to the jury. The poster that I presented is included in Appendix B.2 and available online on the website of the Programming Languages research group.³ The jury selected six participants (three in each category) to advance to the next round.

Second Round: Presentation On Thursday the 24th, the six selected participants gave a presentation of ten minutes followed by a question session of five minutes. In each category, the winning participant advanced to the Grand Finals.

Grand Finals: Short Paper The winners of all SRCs throughout the year competed in the Grand Finals. A different panel of judges evaluated these winners against each other via the web. Three undergraduates and three graduates were chosen as the SRC Grand Finals winners. Finalists had to submit a short paper of no more than 4000 words and no more than 5 pages (excluding references). Appendix B.3 contains my submission, which is also available on the website of the ACM SRC.⁴ It was not selected as one of the winners but did receive one “weak accept” and two “accept” judgements.

¹<https://2019.splashcon.org/track/splash-2019-SRC>

²<https://dl.acm.org/doi/10.1145/3359061.3361085>

³<https://pl.ewi.tudelft.nl/posters/2019/10/22/incremental-scannerless-generalized-lr-parsing/>

⁴<https://src.acm.org/binaries/content/assets/src/2020/maarten-p.-sijm.pdf>

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, Spoofox

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 lexing (or scanning) phase, supports modelling the entire language syntax in one single grammar, and composing grammars for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. *SPLASH Companion '19, October 20–25, 2019, Athens, Greece*
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6992-3/19/10...\$15.00
<https://doi.org/10.1145/3359061.3361085>

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.

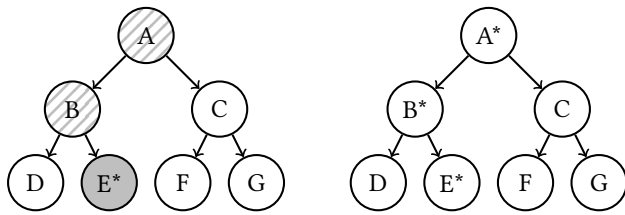
2 Incremental Scannerless GLR Parsing

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 Spoofox 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.

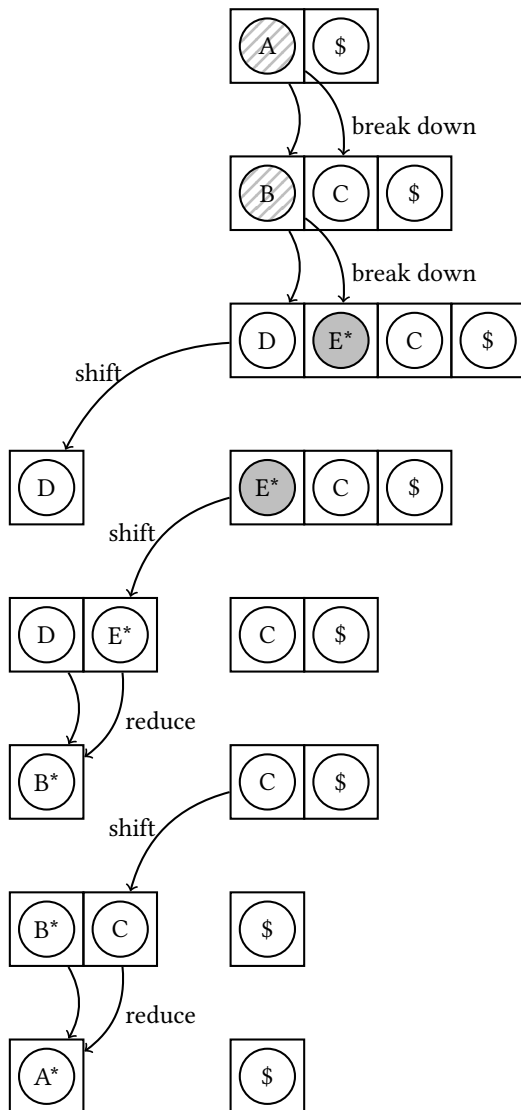
Input Preprocessing After successfully parsing an input file, the parser saves both the input string and the resulting parse forest. When reparsing the same file, it calculates the difference between the previous and the new input strings. The changes can be deletions or insertions, or both at the same 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](#).



(a) Left: a preprocessed parse tree, where node E has been changed. Because of this, its ancestors A and B become invalid. 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.

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 JSGLR2 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:7d9f9bcc-117c-4617-860a-4e3e0bbc8988>

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>

- [2] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofox 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.

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 Spoofox 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.

Processing Changes (Diff)

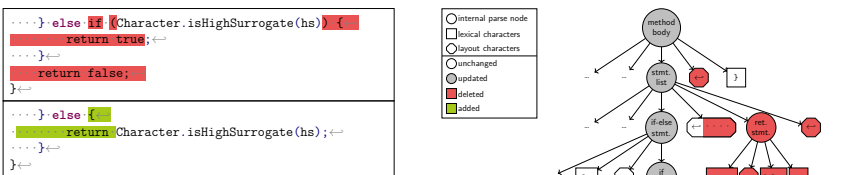
```

AnyKeyboardViewBase.java
final char hs = label.charAt(0);
if (0xd800 <= hs && hs <= 0xdbff) {
    return true;
} else if (Character.isHighSurrogate(hs)) {
    return true;
}
return false;
    
```

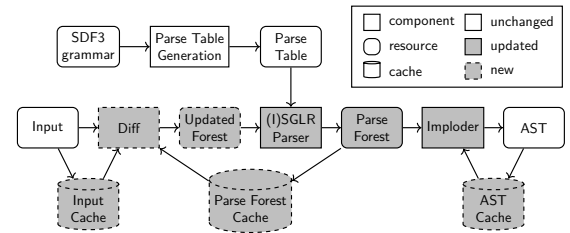
change

```

AnyKeyboardViewBase.java
final char hs = label.charAt(0);
if (0xd800 <= hs && hs <= 0xdbff) {
    return true;
} else {
    return Character.isHighSurrogate(hs);
}
    
```



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").

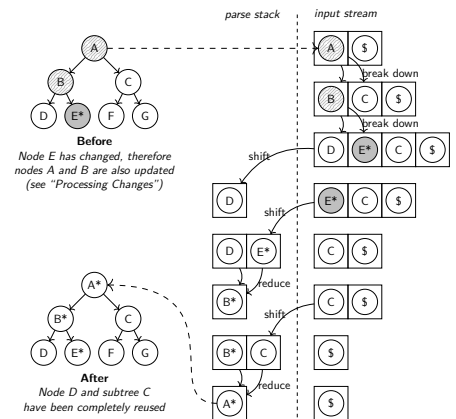


The incremental parsing pipeline architecture. **Top row:** executed during language development. **Middle row:** executed every time that a file is parsed. **Bottom row:** caches that are maintained between parses.

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.

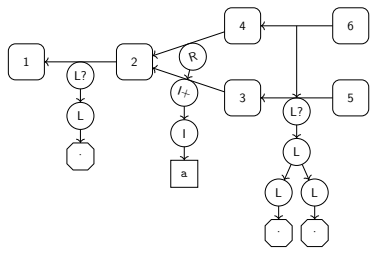


Non-determinism

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]
    
```

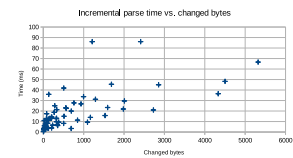
LAYOUT? =

```

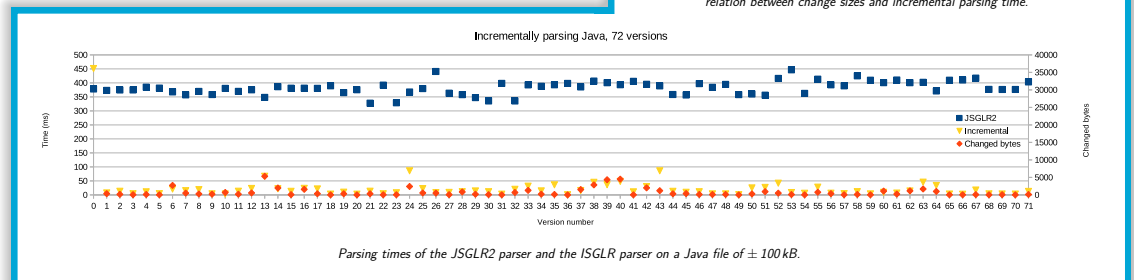
LAYOUT? = LAYOUT
LAYOUT = LAYOUT LAYOUT
LAYOUT = [ \ ]
    
```

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 25x 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:749f9bcc-117c-4617-860a-4e3e0bbcb988>
- [2] Lennart CL Kats and Eelco Visser. 2010. The Spoofox 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/10.1145/1860459.1860497>
- [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.

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, Spofax

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACM Student Research Competition, Grand Finals, 2020
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

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 Spofax 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.

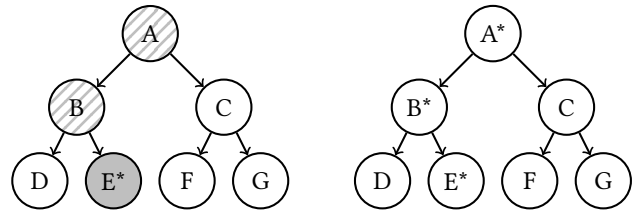
Any new parse nodes created in this process are marked as *irreusable*, signalling to the parser that they are not valid for reuse. This includes the parse nodes created because of insertions and replacements. In addition, all ancestors of any changed nodes are marked irreusable, simply because they 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).

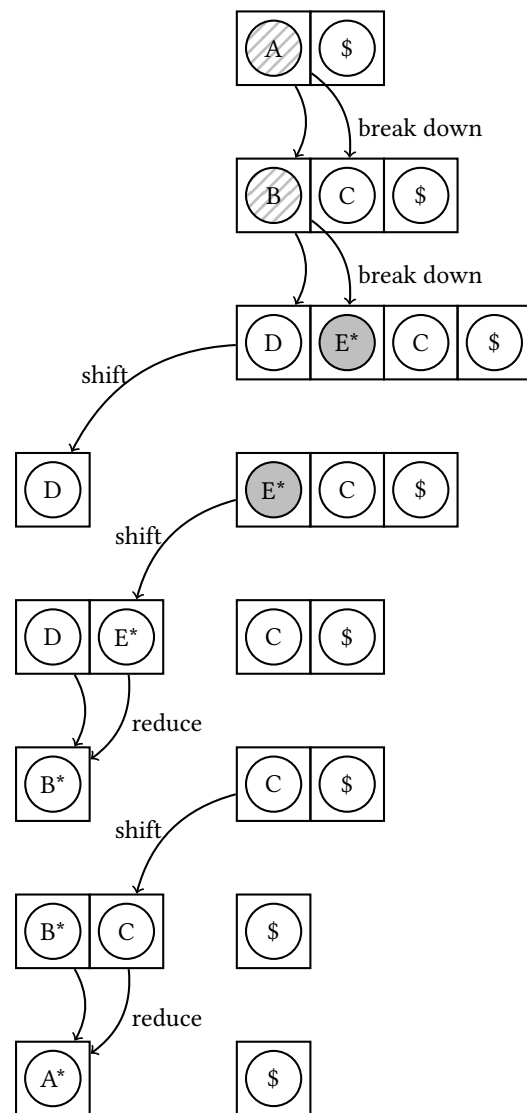
When parsing starts, the input stream consists only of the preprocessed parse tree and the end-of-file marker. When the parser encounters an irreusable parse node in the input stream, it is broken down, meaning that its child nodes will become part of the input stream instead. Ultimately, the parser 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.

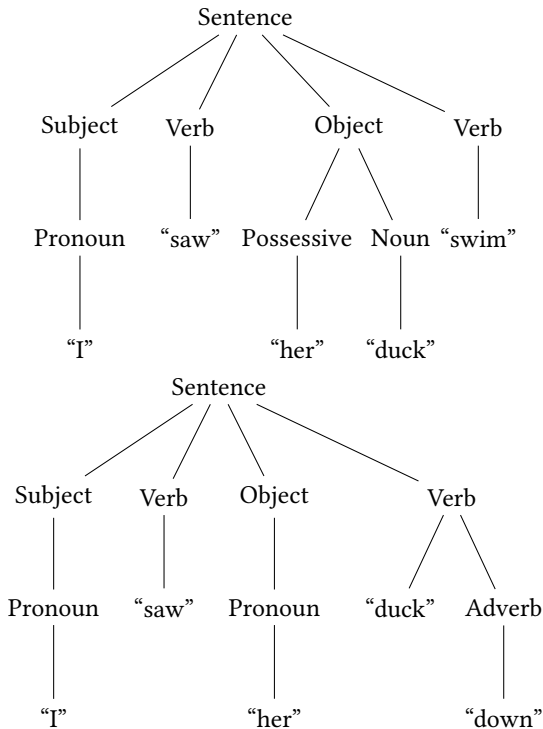


Figure 2. An example where non-determinism is used during parsing. In both sentences “I saw her duck swim” and “I saw her duck down”, there is no ambiguity in the final result, but the word “her” has a different interpretation depending on the final word of the sentence. Therefore, the parser must explore 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 Spoofox language workbench [3]. In Spoofox, language designers can specify programming languages declaratively by describing their syntax, static and dynamic semantics, and transformations. Spoofox 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 Spoofox pipeline, such as an incremental type checker [1], an incremental build system [4], and incremental compilation [7].

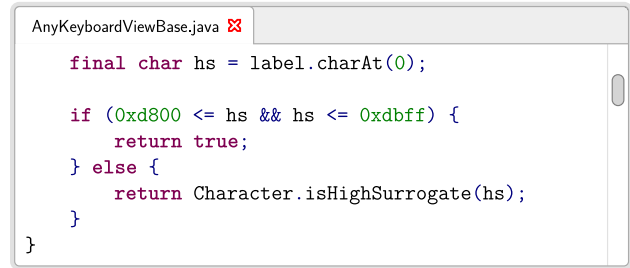
We implemented the ISGLR parsing algorithm as an extension to the JSGLR2 parsing algorithm [2]. The modular nature of JSGLR2 helped to only extend those parts of the parser that required changes to allow incremental parsing, resulting in about 1000 added lines of Java code.

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 Spoofox, *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.



```
AnyKeyboardViewBase.java
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 Spoofox. Language designers can indicate which program elements get which colour and Spoofox will show the right colours in the editor.

Spoofox 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.

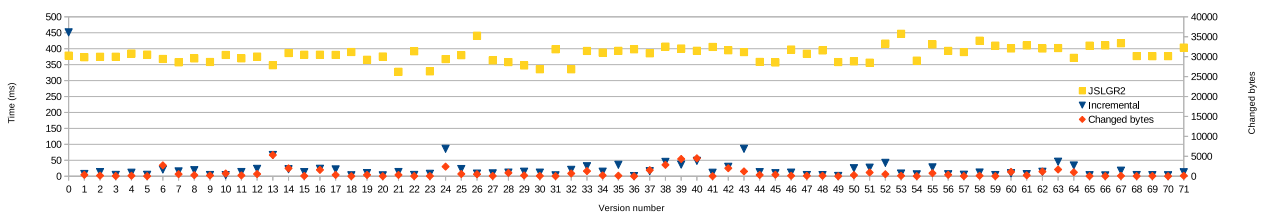


Figure 4. Parsing times of the JSGLR2 parser and the ISGLR parser on a Java file of almost 100 kB. The yellow squares and blue triangles indicate the parsing times (left y-axis) for these two parsers, respectively. The red diamonds indicate the number of changed bytes between each version (right y-axis).

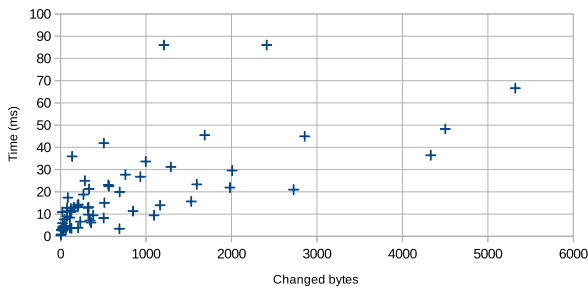


Figure 5. The same incremental parsing times as in Figure 4, showing the relation between change sizes and incremental parsing time.

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 repository¹) to 25 times faster (for a single file of 90 kilobytes that has changes averaging 700 bytes, with a standard deviation of 1100 bytes²). 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 Spoofox 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, NJ.
- [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).

Appendix C

Spoofox Enhancements

The Spoofox language workbench simplifies the development of programming languages by generating many of the common features for a language based on declarative specifications. During my thesis, I have worked on several enhancements of Spoofox that were not directly related to my research on incremental parsing, but rather served as “side quests”. Some of these enhancements were significant enough that I did not want to leave them unmentioned.

Unicode Support Before this enhancement, languages created in Spoofox were constrained to use characters in the ASCII range (`[\0-\255]`). The EOF value was hardcoded to have value `\256`, which hindered the full adoption of Unicode in JSGLR and SDF. In a series of pull requests,¹ I refactored the EOF value to be `-1` instead and adapted the parser (generator) code to work with characters wider than 16 bits. The parse table generator now explicitly handles the EOF as a separate entity, while before, it was just appended to the end of the character class ranges. In addition, SDF3 now supports binary (e.g., `\0b101010`), octal (e.g., `\052`), and hexadecimal (e.g., `\0x2a`) escape codes in character classes, to aid in specifying Unicode characters with high code point values. An example of a Spoofox language that uses Unicode characters is shown in Figure C.1.

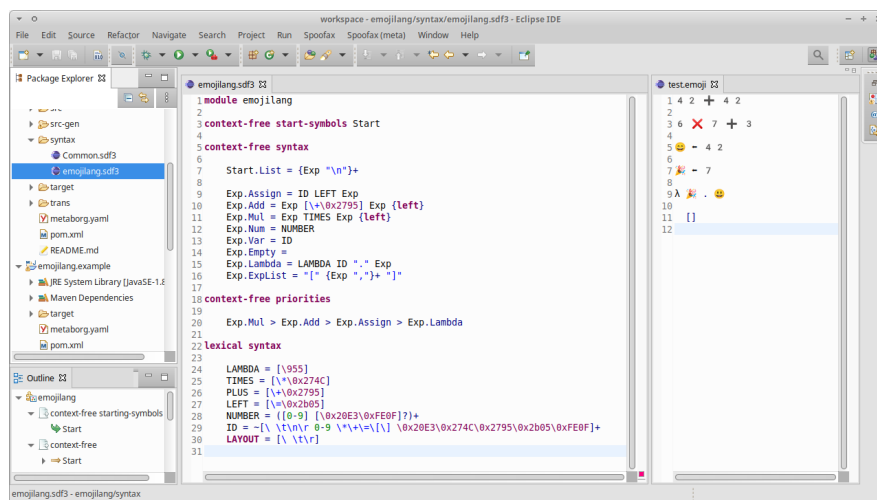


Figure C.1: A screenshot of Spoofox in Eclipse, showing a language that uses Unicode characters. The language supports the lambda symbol, emoji for addition and multiplication symbols and the left-pointing arrow, and numbers with combining code points that turn them into emoji. In this language, identifiers can consist of any characters that are not already used in the language.

¹<https://github.com/metaborg/jsclr/pull/72>, <https://github.com/metaborg/sdf/pull/38>, and more

Spoofax Syntax Highlighting in LaTeX Typesetting larger snippets of code in LaTeX with syntax highlighting is usually done with the `listings` or `minted` package. However, these packages typically only support the most-used General-Purpose Languages (GPLs). Using them to typeset a Domain-Specific Language (DSL) generally involves a bunch of workarounds, including manually listing keywords or manually overriding colours for certain parts of the displayed code. Therefore, Chiel Bruin and I have automated the typesetting of languages that are built with Spoofax.² The implementation is a plugin for the Python package `pygments`, which is used by the `minted` package to generate syntax highlighting in LaTeX. Our plugin wraps the JSGLR2 command-line JAR³ and converts the resulting tokens and associated colours to a format that `pygments` understands. This thesis also makes use of this tool to typeset code snippets of SDF3, a DSL that is also defined in Spoofax, for example in Figures 2.7 and 3.1.

Spoofax Dark Theme Many text editors for programming languages (and many other applications as well) include a dark theme, to reduce eye strain from looking at a bright screen for a long period. The Spoofax language workbench is implemented as a plugin in Eclipse and IntelliJ, two major IDEs that have both light and dark themes. However, the languages generated by Spoofax are fixed to a single colouring scheme, defined in the ESV metalangage.⁴ This colouring scheme is usually defined to have dark letters on a light background, so setting Eclipse to use a dark theme makes Spoofax languages unreadable. To resolve this, I devised a workaround (also known as “hack”) that inverts the lightness of the colours defined in ESV when Eclipse is using a dark theme,⁵ which can be used until ESV changes the way it uses colour schemes. I preferred this over simply inverting the colours since that results in very bright text colours for most Spoofax languages. An example of a Spoofax instance in Eclipse with a dark theme is shown in Figure C.2.

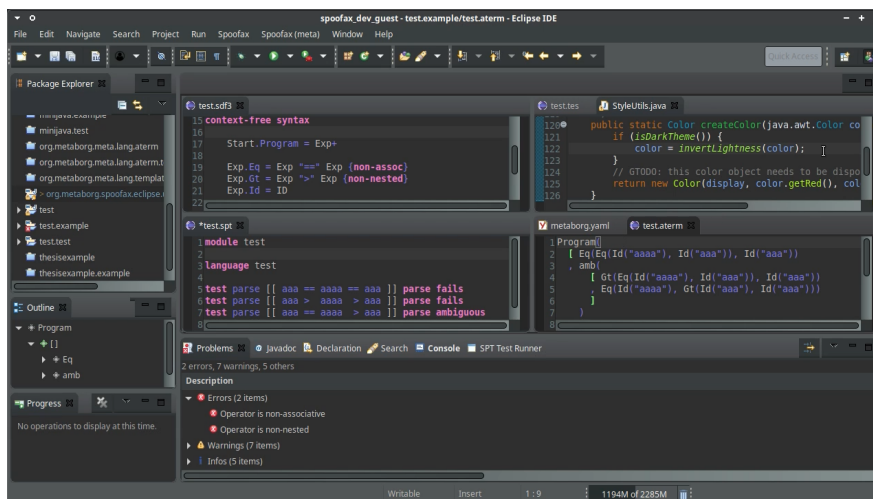


Figure C.2: A screenshot of Spoofax in Eclipse, using a dark theme. Top-left: SDF3. Top-right: Java (default Eclipse colours, not the Spoofax colours). Bottom-left: the Spoofax testing language (SPT). Bottom-right: the ATerm language, which describes ASTs.

²<https://github.com/ChielBruin/spoofax-latex-tools/#spoofax-pygments>

³<https://github.com/metaborg/jsjglr/tree/master/org.spoofax.jsjglr2.cli>

⁴<https://www.metaborg.org/en/latest/source/langdev/meta/lang/esv.html#syntax-highlighting>

⁵<https://github.com/metaborg/spoofax-eclipse/pull/19>

SLR(1) Parse Table Generation As an unfinished experiment, I have implemented an SLR(1) parse table generator in SDF3.⁶ While SLR(1) parse table generation works for most grammars, it currently does not yet work for layout-sensitive grammars. In addition, it still needs a configuration option in the properties file of Spoofox languages (`metaborg.yaml`). I have used this experimental branch to test if it improves the performance of the ISGLR parser, but its impact did not look promising enough to finish the implementation. However, it did help to provide me insight on the different parsing behaviour between LR(0) and (S)LR(1) parse tables, which I refer to in Sections 3.2 and 3.3.

⁶<https://github.com/metaborg/sdf/pull/27>