A Modular SGLR Parsing Architecture for Systematic Performance Optimization

by

Jasper Denkers

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday January 24, 2018 at 11:00 AM.

Faculty of Electrical Engineering, Mathematics & Computer Science

Department of Software Technology

Programming Languages Group

Student number:	4212584	
Project duration:	February 1, 2017 – January 24, 2018	
Thesis committee:	Prof. dr. E. Visser,	TU Delft, supervisor
	Prof. dr. J. J. Vinju,	CWI / TU Eindhoven
	Dr. A. E. Zaidman,	TU Delft
	L. E. S. Amorim MSc,	TU Delft, daily supervisor
	Dr. M. J. Steindorfer,	TU Delft, supervisor

Abstract

SGLR parsing is an approach that enables parsing of context-free languages by means of declarative, concise and maintainable syntax definition. Existing implementations suffer from performance issues and their architectures are often highly coupled without clear separation between their components. This work introduces a modular SGLR architecture with several variants implemented for its components to systematically benchmark and improve performance. This work evaluates these variants both independently and combined using artificial and real world programming languages grammars. The architecture is implemented in Java as JSGLR2, the successor of the original parser in Spoofax, interpreting parse tables generated by SDF3. The improvements combined result into a parsing and imploding time speedup from $3x$ on Java to $10x$ on Green-Marl with respect to the previous JSGLR implementation.

Acknowledgements

This thesis is the result of approximately 10 months of work, completing the last 5,5 years as Computer Science student at the Delft University of Technology. I want to thank all members of the Programming Languages Research Group at TU Delft for their support during the course of this project. In particular my supervisors, from whom I have learned a lot and with whom I enjoyed working together: Eduardo Souza, for lots of feedback and the weekly discussions that kept me on track. Michael Steindorfer, for pointing me in the right direction for several performance improvements. Eelco Visser, for inventing the SGLR parsing algorithm and for suggesting doing this thesis on performance optimization on it. With great pleasure I look back on several meetings the four of us had, that often took much more time than planned, but that always led to interesting new insights and that gave me more motivation to proceed. In addition to my supervisors, I would like to thank Jurgen Vinju and Andy Zaidman for serving on my thesis committee. Finally, I would like to thank the two most important people in my life, my parents, whose unconditional support was essential in getting where I am today.

Part of this work was presented at the Parsing@SLE workshop[1] as part of the SPLASH 2017 conference.

Jasper Denkers
Leiden, December 2017

[1] https://2017.splashcon.org/track/parsing-2017

Contents

1 Introduction 1

2 Scannerless Generalized LR Parsing 3

2.1 Context Free Grammars . . . 3
2.2 LR Parser Generation and Parsing . . . 4
2.3 GLR Parsing . . . 9
2.4 Scannerless Parsing . . . 17
2.5 SGLR . . . 18

3 Syntax and Parsing in Spoofax 23

3.1 The Spoofax Language Workbench . . . 23
3.2 Abstract Syntax with Annotated Terms . . . 23
3.3 Syntax Definition with SDF3 . . . 25
3.4 Disambiguation Filters . . . 27
3.5 Parsing with JSGLR . . . 29
3.6 Representation of Characters . . . 30
3.7 Imploding . . . 30
3.8 Error Recovery . . . 30

4 A Modular SGLR Architecture 33

4.1 Motivation . . . 33
4.2 Decomposing SGLR . . . 33
4.3 Components . . . 34
4.4 Implementation . . . 37
4.5 Testing . . . 38
4.6 Observing . . . 38
4.7 Visualization . . . 39

5 Performance Optimizations 41

5.1 Parse Table Data Structures . . . 41
5.2 Elkhound: Hybrid LR/GLR Reducing . . . 41
5.3 Parse Forest Reduction . . . 44
5.4 Stack Collections . . . 44
5.5 Hybrid Stack and Parse Forest Data Structures . . . 45

6 Results 47

6.1 Setup . . . 47
6.2 Parse Table Representation . . . 48
6.3 Elkhound . . . 48
6.4 Parse Forest Reduction . . . 50
6.5 Stack Collections . . . 51
6.6 Hybrid Data Structures . . . 51
6.7 Best Variants . . . 53
6.8 Comparing Languages . . . 55
6.9 Imploding . . . 56
6.10 Threats to Validity . . . 56
6.11 Modularity of Architecture . . . 56

7 Conclusions 59

7.1 Discussion . . . 59

8 Future Work **61**

Bibliography **63**

Appendices **67**

A Parse Table Measurements **69**

B Parsing Measurements **71**

1 Introduction

Effective language designing requires declarative, concise and maintainable syntax definition by means of context-free grammars. This enables more flexibility in the field of language prototyping and embedded language design which requires compilers and interpreters depending on parsers. While implementing such parsers, other concerns are relevant like the parser's complexity, architecture and performance. Those goals in syntax definition and parser implementation conflict and thus an advanced parsing algorithm is required such as SGLR [42], extending the traditional LR parsing algorithm [19] with the concepts of *generalized* and *scannerless* parsing.

Modular and reusable grammars require composable syntax definition. Only the full class of context-free grammars is closed under composition, thus LR parsing and its variants do not suffice because they only support a subset of the class of context-free grammars. Since LR parsing is deterministic it also can not handle grammars that are ambiguous or grammars that require unbounded lookahead. Generalized LR parsing (GLR) [29, 35] does support the full class of context-free grammars.

Conflicts in the interface between lexical and context-free syntax need to be avoided to support language composition. Scannerless parsing [30] solves this issue. Traditional parsing techniques work with a separate lexical and context-free analysis phase, where in the lexical analysis phase a scanner strips layout and transforms characters into tokens which are then processed by a context-free parser. Scannerless parsing uses syntax definition in which the lexical and context-free part of the grammar are integrated, increasing declarativeness and grammar maintainability [17]. Also, lexical disambiguation can then use information from the context, and the scanner is discarded leading to a simpler parser architecture. SGLR combines GLR with scannerless parsing and enables declarative syntax definition.

SGLR is an advanced approach to parsing, but its existing implementations are much slower than e.g. ANTLR [28]. While SGLR has additional features, like the support for ambiguous parses, performance closer to that of ANTLR would increase practical application possibilities. Multiple performance improvements are found for GLR parsing which can be applied to SGLR, too. To systematically evaluate such improvements, a modular architecture is required in which its components can be replaced by improved variants without influencing the rest of the parser. Those variants can include algorithmic, data structure and general software engineering improvements.

This work presents a modular architecture for SGLR parsing, an implementation of this architecture with several improvements for its components and a systematic benchmarking approach to evaluate them. This parsing architecture, including imploding and tokenization, is implemented in Java in Spoofax [15, 39] as JSGLR2, the successor of the original parser implementation JSGLR. It uses parse tables generated by the syntax definition formalism SDF3 [40, 43].

The architecture enables the systematic benchmarking of alternative implementations of its components. Variants are evaluated by replacing only the components they impact, leaving the others intact. Improvements for five variation points are implemented to achieve better performance as compared to the naive implementation of the original SGLR algorithm which serves as the baseline. Benchmarks are performed by parsing inputs for multiple real world programming languages and artificial grammars with specific properties.

First, improved data structures are implemented to retrieve applicable actions from parse tables. Second, reducing is improved by implementing a hybrid LR/GLR approach as presented in Elkhound [21]. It detects which parts of the input are deterministic and switches to regular LR parsing where possible, discarding the overhead involved with GLR. The new architecture enables this variant by replacing the stack representation and by using a specialized reducing component, independently of the parse forest representation. Third, parse forest construction is optimized by skipping the construction of lexical, layout and rejected subtrees. Such parts of the input only need to be recognized and construction of the top, context-free, part of the parse forest suffices. Fourth, alternative implementations for the collections of stacks the parser uses are considered. Last, improved versions are implemented for the two main data structures that are maintained during parsing, the stack and parse forest, in which list instantiations are prevented when nodes only have one link to other nodes.

The rest of this report is structured as follows. First, SGLR parsing is described in detail and the differences with respect to LR and GLR parsing are discussed in Chapter 2. Chapter 3 covers the context in which this work is executed: the Spoofax language workbench with syntax definition and parser generation using SDF3. The main contribution of this work, a modular SGLR architecture and its implementation, JSGLR2, is described in Chapter 4. Several improvements to the components are implemented and described in Chapter 5. Chapter 6 describes the results of benchmarking the variants of the parser that can be composed with the improved components. Lastly, conclusions are given in Chapter 7 and Chapter 8 suggests future work.

Algorithms from different authors are adopted in this report. While their definitions in some cases are equivalent to their originals, styling is adjusted to have all algorithms presented in an uniform manner. Changes in algorithm style definition include notation, variable declarations and names, pseudo code operator names and refactorings.

2

Scannerless Generalized LR Parsing

This chapter describes the fundamentals of SGLR parsing. It describes how context-free grammars can be used to define languages. SGLR parsing is an extension of regular GLR parsing, which itself is an extension of LR parsing and depends on LR parser generation. First, the chapter covers LR and GLR. Next, it describes the concept of scannerless parsing and its influence on parser architecture. Finally, it presents the combination of these concepts, namely SGLR. Pseudocode describes the GLR algorithm, which later indicates the differences between GLR and SGLR.

2.1. Context Free Grammars

The syntax of programming languages can be formally defined by means of a context-free grammar (CFG).

Definition 1. *A context-free grammar G is a tuple* (Σ, N, P, S)*, with* Σ *a set of terminal symbols, N a set of non-terminal symbols, P a set of productions of the form* $A \rightarrow \alpha$ *with* $A \in N$ *and* $\alpha \in (\Sigma \cup N)^*$ *and start symbol* $S \in N$*.*

A further description of the sets and the concepts they represent:

Terminals (Σ**)** The actual lexical building blocks for the strings derived by the language defined by the CFG. Also called tokens.

Non-terminals (N**)** Do not appear in strings derived by a language. They are used to indicate patterns of symbols that the language contains.

Symbols The union of terminals and non-terminals.

Production rules (P**)** Depicted with $A \rightarrow \alpha$ where A is a single non-terminal on the left hand side and α is a sequence of symbols on the right hand side. Additionally, productions can have constructors C using the form $A.C \rightarrow \alpha$. Constructors specify names of abstract syntax nodes when imploding parse trees.

Start symbol (S**)** A non-terminal that indicates how the derivation of strings in a language starts.

The set of production rules derive all strings as defined by the grammar's language. A *derivation* is found by a recursive process of expanding non-terminals by means of production rules. Expanding means replacing a non-terminal that is on the left hand side of a production rule by the symbols on the right hand side of the rule. This process is repeated until all non-terminals are expanded and the result is a sequence of terminal symbols forming a string. Deriving a string starts with expanding on the start symbol.

Listing 2.1: Grammar G_1; expressions of x's with additions and multiplications.

```
Exp.Mul = Exp * Exp
Exp.Add = Exp + Exp
Exp.Var = x
```

An example can be found in Listing 2.1. This grammar describes a language of expressions with multiplications and additions with only the variable x using operators $*$ and $+$. Following Definition 1 the grammar is described by the tuple $G_1 = (\Sigma = \{*, +, x\}, N = \{Exp\}, P = \{Exp.Mul = Exp * Exp, Exp.Add =$

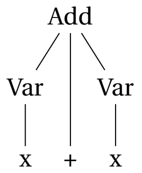
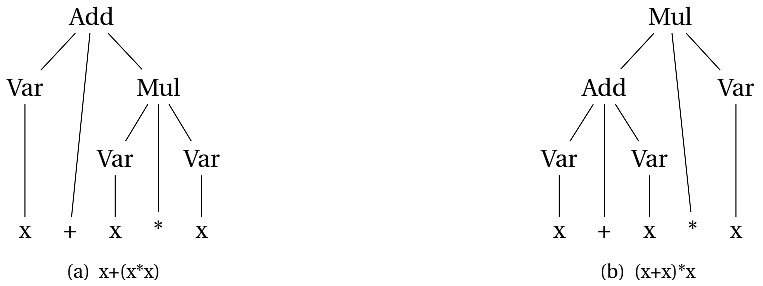


Figure 2.1: Parse tree for G_1 and string $x + x$.



(a) x+(x*x)

(b) (x+x)*x

Figure 2.2: Two parse trees for G_1 and string $x + x * x$.

$Exp + Exp, Exp.Var = x\}, S = Exp)$. For example, the string $x + x$ is in this language, since it can be derived by expanding non-terminals using the grammar's productions, starting from the start symbol *Exp*. First, *Exp* is expanded using production $Exp.Add = Exp + Exp$ resulting into $Exp + Exp$. Then, both *Exp*'s are expanded using production $Exp.Var = x$. The expansion steps can be described as follows (the constructors above arrows indicate which production is used on that step): $Exp \xRightarrow{Exp.Add} Exp + Exp \xRightarrow{Exp.Var} Exp + x \xRightarrow{Exp.Var} x + x$. When applying production $Exp.Var = x$ for the first time we can choose to expand on both the left as right *Exp*. We call a derivation *leftmost* if the expansion always happens on the leftmost non-terminal. Naturally, a *rightmost* derivation always expands on the rightmost non-terminal. The above derivation of $x + x$ is a rightmost derivation. Figure 2.1 visualizes this derivation with a parse tree, using constructors for nodes.

Independent of whether a leftmost or rightmost derivation is chosen, deriving $x + x$ always results into a single parse tree. This means the parse is non-ambiguous. There are also strings in the language that can be ambiguous. $x + x * x$ is such string as we can see in Figure 2.2, since the sentence can be derived by multiple parse trees. Consider the language defined by Grammar 1. A mathematical interpretation would prefer the derivation $x + (x * x)$ over $(x + x) * x$ due to the higher precedence of the $*$ operator over $+$. However, G_1 does not encode this precedence.

In the rest of this work, parse trees for productions $A.C \rightarrow \alpha_1 ... \alpha_n$ are textually represented in the format `[A.C =` t_1 `...` t_n`]`, in which subtrees t_i represent parse trees for their corresponding symbol α_i. For example, the textual representations for the previously discussed examples are:

Figure 2.1 `[Exp.Add = [Exp.Var = x] + [Exp.Var = x]]`

Figure 2.2(a) `[Exp.Add = [Exp.Var = x] + [Exp.Mul = [Exp.Var = x] * [Exp.Var = x]]]`

Figure 2.2(b) `[Exp.Mul = [Exp.Add = [Exp.Var = x] + [Exp.Var = x]] * [Exp.Var = x]]`

2.2. LR Parser Generation and Parsing

Knuth introduced the LR(k) parsing algorithm in 1965 [19] that can be used to parse deterministic languages in linear time. With LR parsing, a grammar is first transformed into a parse table representing a finite automaton in a process called parser generation. The states of the automaton represent sets of items that are productions with a dot somewhere on the right hand side representing the current parser position. The parse table can be constructed according to the sets of items and transitions in the automaton. Such elements specify the actions the parser should take considering the input string. The parser successively performs these types of actions while persisting state transitions on a linear stack:

Shift(s) Read a token from the input, push state s to the stack and transition to state s.

Reduce(p) Reduce by production p. Based on the length of the production's right hand side, a number of states are popped from the stack.

Goto(s) After performing a reduction, an old stack node is now again the topmost stack. Its state together with the left hand side non-terminal symbol of the reduced production derives a goto action from the parse table. This instructs the parser to push state s to the stack and transition to it.

Accept Conclude a successful parse.

Links between stack nodes contain parse tree nodes, which can be either tokens in the case of shifts, or derivations of a production in case of reduces. The process of consuming the input, transitioning between states and mutating the stack is repeated until the parser reaches an accepting state to conclude a successful parse.

For grammars with start symbol S, an extra production $S' \to S\$$ is added with S' becoming the new start symbol and $ representing the end of the input. The parser appends $ to the input before starting a parse. This helps LR parsing detect when it is finished. Algorithm 1 formally describes the algorithm in pseudo code. It successively consumes tokens from the input and performs the actions prescribed by the parse table for its current configuration. It would fail if for a given configuration no or multiple actions are applicable.

Algorithm 1 The basic LR parsing algorithm

```
PARSE(parsetable, input)
stack = [0]
for all tokens $t \in$ concat(input, $) do
    actions = lookupInParseTable(stack.peek, t)
    switch actions do
        case {Accept} return success
        case {Shift(s)}
            input.pop
            stack = stack.push(s)
        case {Reduce(p)}
            rhsLength = length(p.rhs)
            stack.pop(rhsLength)
            gotoAction = lookupInParseTable(stack.peek, p.lhs)
            stack.push(gotoAction.gotoState)
        case otherwise return error
end for
```

The sequence of reduce actions that are performed represent the application of grammar rules for a reversed rightmost derivation of the input program. The intermediate parser configurations are characterised by the input, stack and current state. Table 2.1 is an example of a parse table for G_1, corresponding to the automaton in Figure 2.3. In the automaton we see states representing item sets. Dashed arrows from state s_i to s_j labelled by token t indicate that when the parser is in state s_i and the next input token is t, it should shift to s_j. Similarly, solid arrows indicate goto actions.

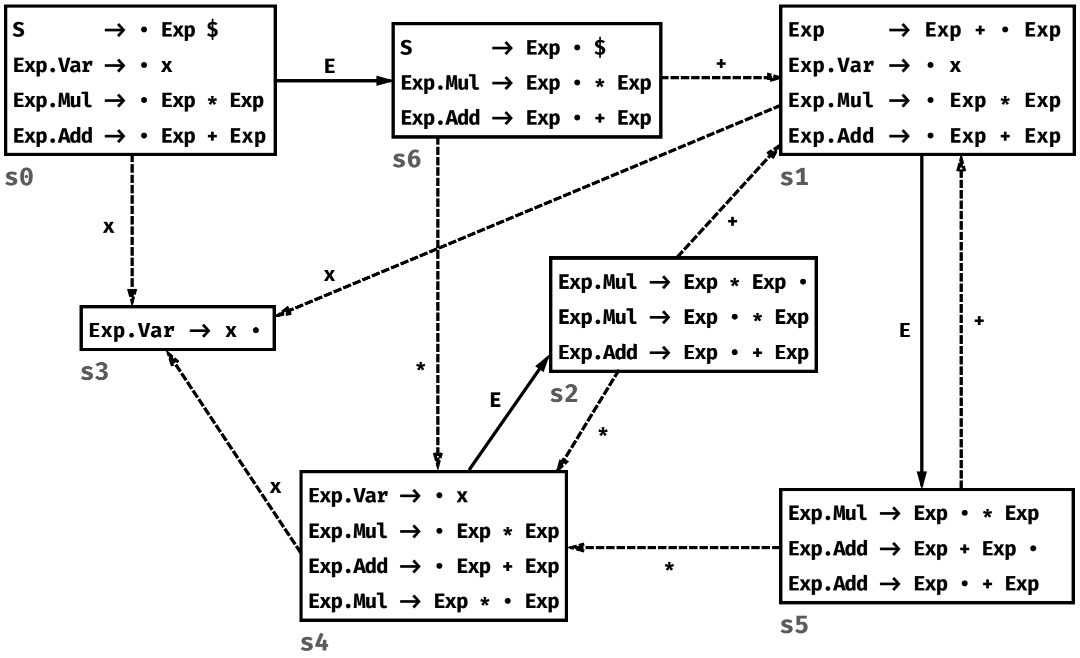
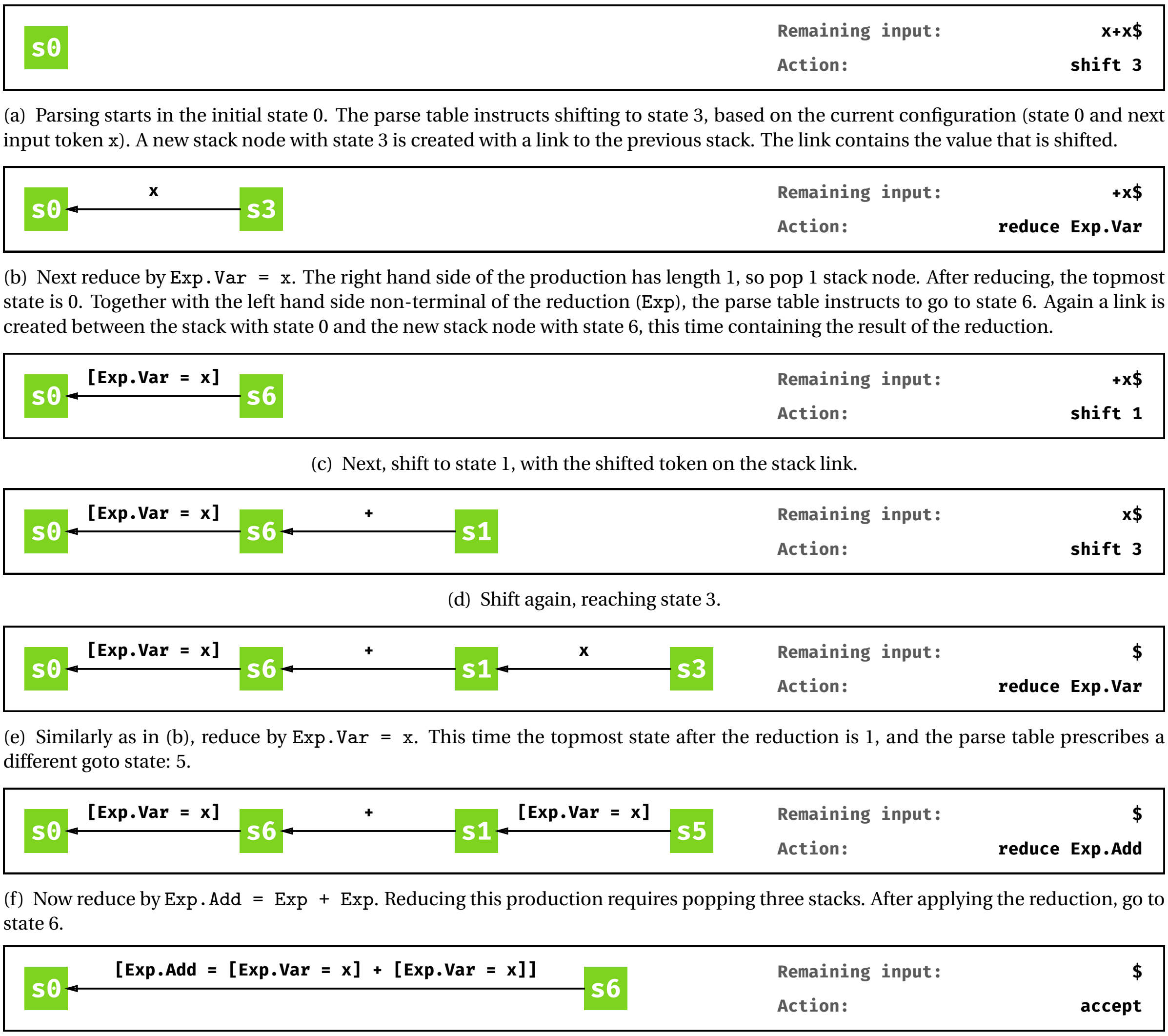


Figure 2.3: SLR parser automaton for G_1 corresponding the parse table in Table 2.1. Dashed arrows indicate shift actions, solid arrows indicate goto actions.

	+	*	x	$	E
0			**Shift**(3)		**Goto**(6)
1			**Shift**(3)		**Goto**(5)
2	**Shift**(1), **Reduce**(`Exp.Mul`)	**Shift**(4), **Reduce**(`Exp.Mul`)		**Reduce**(`Exp.Mul`)	
3	**Reduce**(`Exp.Var`)	**Reduce**(`Exp.Var`)		**Reduce**(`Exp.Var`)	
4			**Shift**(3)		**Goto**(2)
5	**Shift**(1), **Reduce**(`Exp.Add`)	**Shift**(4), **Reduce**(`Exp.Add`)		**Reduce**(`Exp.Add`)	
6	**Shift**(1)	**Shift**(4)		**Accept**	

Table 2.1: SLR parse table for G_1.



(a) Parsing starts in the initial state 0. The parse table instructs shifting to state 3, based on the current configuration (state 0 and next input token `x`). A new stack node with state 3 is created with a link to the previous stack. The link contains the value that is shifted.

(b) Next reduce by `Exp.Var = x`. The right hand side of the production has length 1, so pop 1 stack node. After reducing, the topmost state is 0. Together with the left hand side non-terminal of the reduction (`Exp`), the parse table instructs to go to state 6. Again a link is created between the stack with state 0 and the new stack node with state 6, this time containing the result of the reduction.

(c) Next, shift to state 1, with the shifted token on the stack link.

(d) Shift again, reaching state 3.

(e) Similarly as in (b), reduce by `Exp.Var = x`. This time the topmost state after the reduction is 1, and the parse table prescribes a different goto state: 5.

(f) Now reduce by `Exp.Add = Exp + Exp`. Reducing this production requires popping three stacks. After applying the reduction, go to state 6.

(g) An accepting configuration is reached and thus parsing succeeded. The result of the parse is the parse tree on the link from the accepting stack node to the initial stack node, in this case `[Exp.Add = [Exp.Var = x] + [Exp.Var = x]]`.

Figure 2.4: LR parsing `x+x` using the parse table from Figure 2.1.

Figure 2.4 describes the steps and configurations the parser goes through while parsing $x+x$. It applies the following sequence of reductions: `Exp.Var = x`, `Exp.Var = x` and `Exp.Add = Exp + Exp`. When reversing this sequence and then expanding on the rightmost symbols this results in the same parse tree as in Figure 2.1. This is characteristic for LR: the reductions it applies represent a reversed rightmost derivation of the input.

If the parser reaches a configuration in which no actions are prescribed by the parse table, parsing fails. Sometimes multiple actions are found. Those situations are called conflicts and then the parsing automaton is not deterministic anymore. Some of such conflicts can be resolved during parser generation. DeRemer suggested two improvements on Knuth's algorithm in 1969 [7]. First, SLR parser generation improves on LR by reducing the amount of *Reduce*-actions that are put in the parse table, thus reducing the amount of conflicts. SLR is the technique used for generating the parse table in Table 2.1. While the number of reductions is lower than that in the parse table generated using regular LR parser generation, there might still be conflicts in SLR parse tables, i.e. cells containing multiple actions. Table 2.1 contains 4 shift/reduce conflicts. When the parser reaches such a configuration, it does not know whether to shift a new token or reduce. reduce/reduce conflicts may also occur in which the parser can not tell by which production to reduce.

Parsing $x + x * x$ with G_1 is an example, as shown by the configuration in Figure 2.5 which the parser reaches after 6 steps. The first steps are the same as for parsing $x + x$. However, when the next input token is then $*$ instead of \$, the parse table indicates both a shift and reduce action and thus a conflicting state is reached. Since the parser now can take more than one action, parsing is non-deterministic. The regular

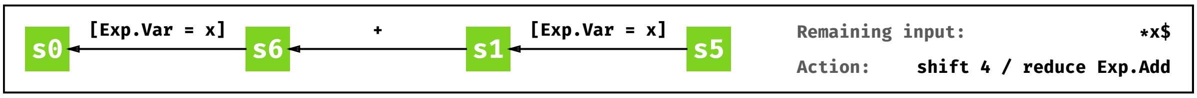
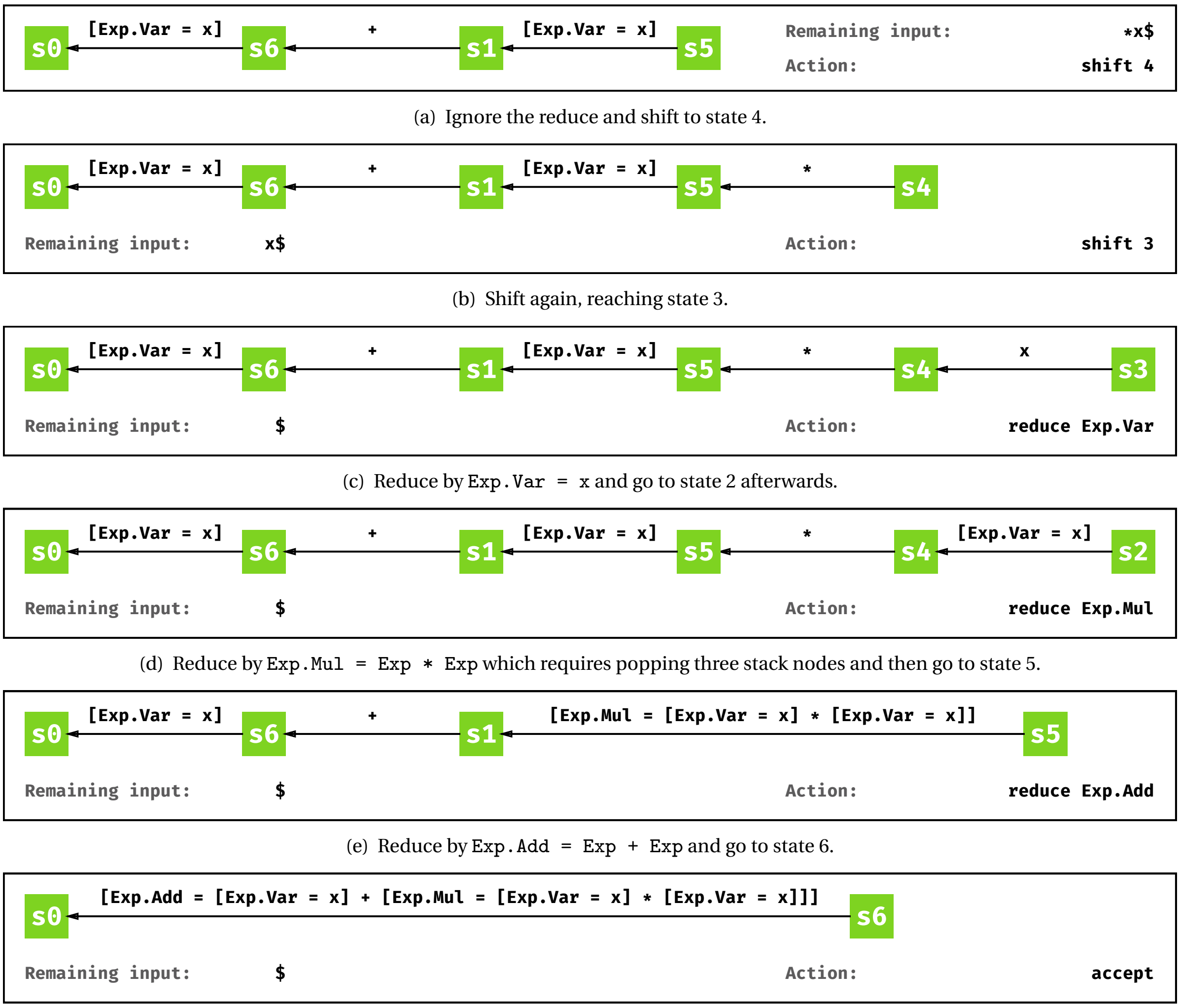


Figure 2.5: LR parsing `x+x*x` using the parse table from Figure 2.1, reaching a conflicting state.

LR parsing algorithm does not support this and fails on non-deterministic input. However, we can show the result of each possibility in Figure 2.6 (choose shift) and Figure 2.7 (choose reduce), corresponding to the parse trees in Figure 2.2(a) and Figure 2.2(b), respectively.



(a) Ignore the reduce and shift to state 4.

(b) Shift again, reaching state 3.

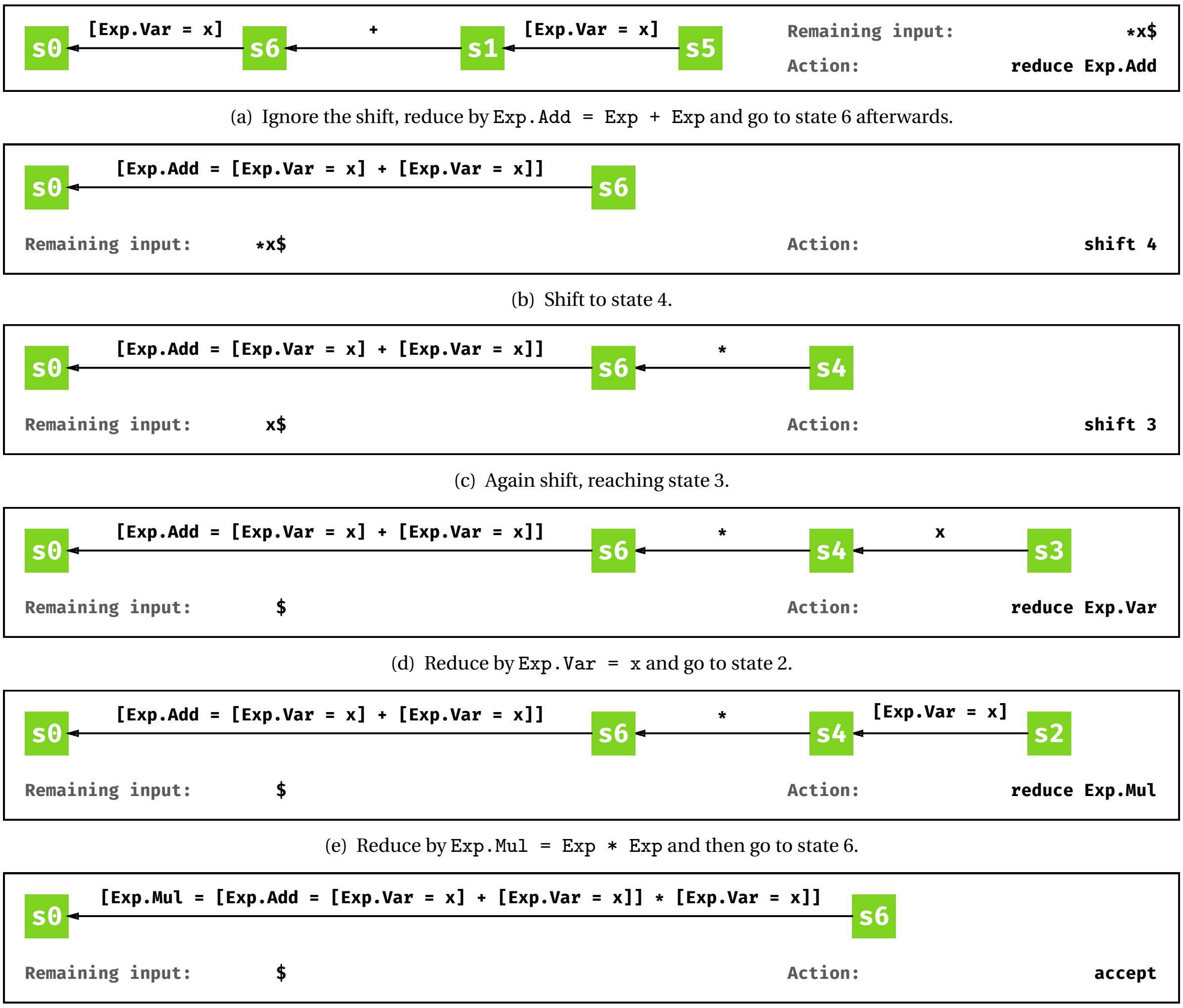
(c) Reduce by `Exp.Var = x` and go to state 2 afterwards.

(d) Reduce by `Exp.Mul = Exp * Exp` which requires popping three stack nodes and then go to state 5.

(e) Reduce by `Exp.Add = Exp + Exp` and go to state 6.

(f) Conclude a successful parse.

Figure 2.6: LR parsing `x+x*x`, continuing after the conflict by choosing shift over reduce.



(a) Ignore the shift, reduce by `Exp.Add = Exp + Exp` and go to state 6 afterwards.

(b) Shift to state 4.

(c) Again shift, reaching state 3.

(d) Reduce by `Exp.Var = x` and go to state 2.

(e) Reduce by `Exp.Mul = Exp * Exp` and then go to state 6.

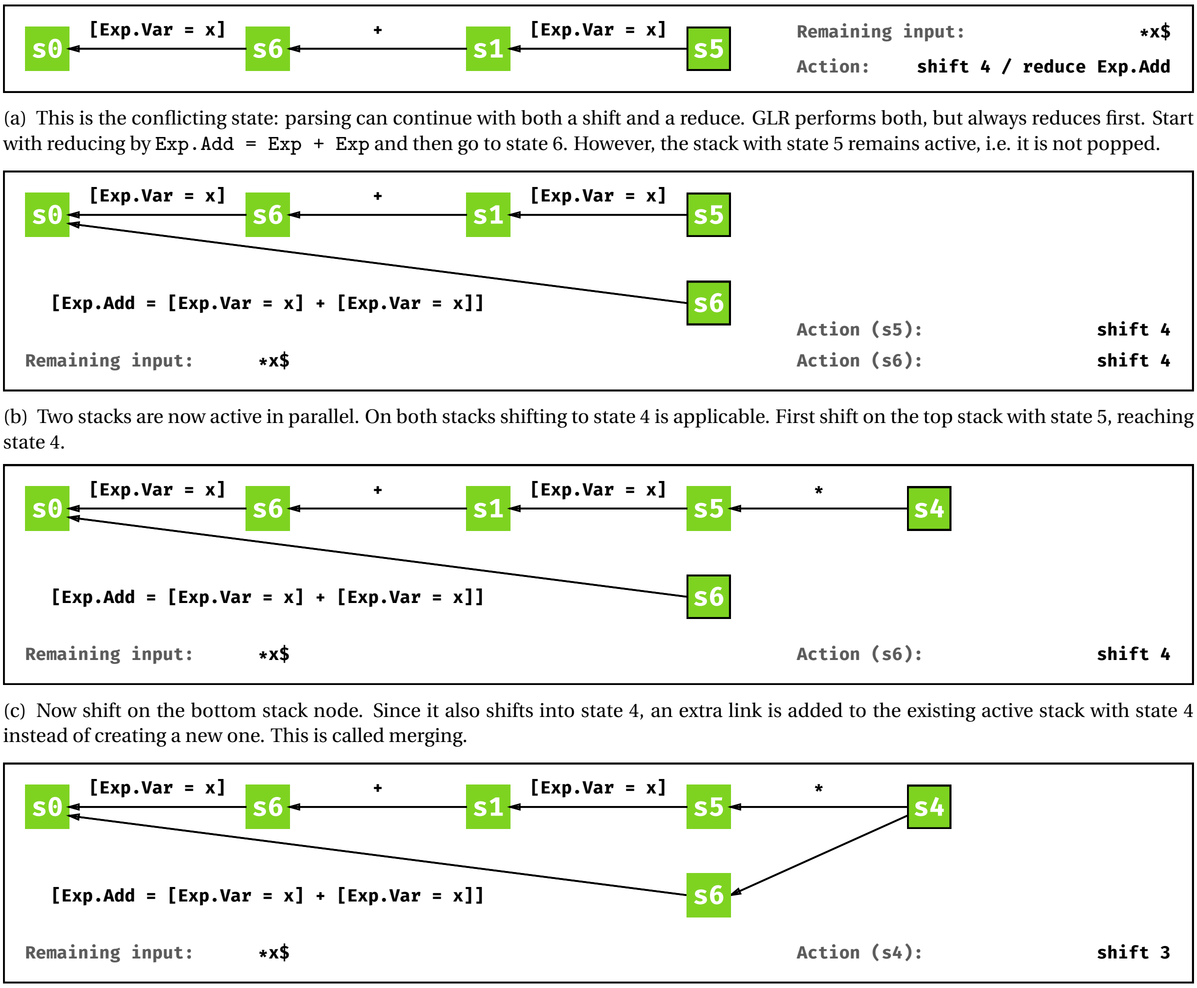
(f) Conclude a successful parse.

Figure 2.7: LR parsing `x+x*x`, continuing after the conflict by choosing reduce over shift.

Incorporating lookahead can make the LR(k) algorithm even more powerful. In that case, the derivations of actions from the parse table are based on the parser configuration and lookahead of a sequence of k symbols. In practice, LR(k) parser generation for $k > 0$ results into big parse tables with many states. By merging states that are similar but with different lookahead, the parse table sizes can be reduced. This might go at the cost of the introduction of reduce-reduce conflicts. That is the idea behind the second proposal of DeRemer: LALR parser generation. Often the simplified versions of DeRemer were found more practical than the original LR(1) parsing algorithm, since parse table size were drastically reduced. However, in 1977 Pager [27] found a way of creating LR(1) parse tables with sizes close to LALR(1) parse tables sizes, while maintaining the applicability to the full class of LR(1) grammars. The improvements on both parser generation and the actual parser can increase the set of unambiguous non-deterministic grammars that can be parsed, but conflicts may still occur due to ambiguous grammars.

2.3. GLR Parsing

There is no version of LR(k) parsing that can handle all context-free grammars. For example, grammars that require unbounded lookahead or that are ambiguous need a more powerful parsing algorithm since they are non-deterministic. This means that in some parser configurations multiple actions are applicable, as we have seen in the example in the previous section. LR parsing simply fails in such cases by reporting a shift/reduce or reduce/reduce conflict. While LR parsing is deterministic and handles a single stack, GLR extends LR by essentially maintaining multiple stacks in parallel. This enables the parser to cope with nondeterminism. GLR is based on the theoretical foundations of Lang from 1974 [20] and was implemented by Tomita in



(a) This is the conflicting state: parsing can continue with both a shift and a reduce. GLR performs both, but always reduces first. Start with reducing by `Exp.Add = Exp + Exp` and then go to state 6. However, the stack with state 5 remains active, i.e. it is not popped.

(b) Two stacks are now active in parallel. On both stacks shifting to state 4 is applicable. First shift on the top stack with state 5, reaching state 4.

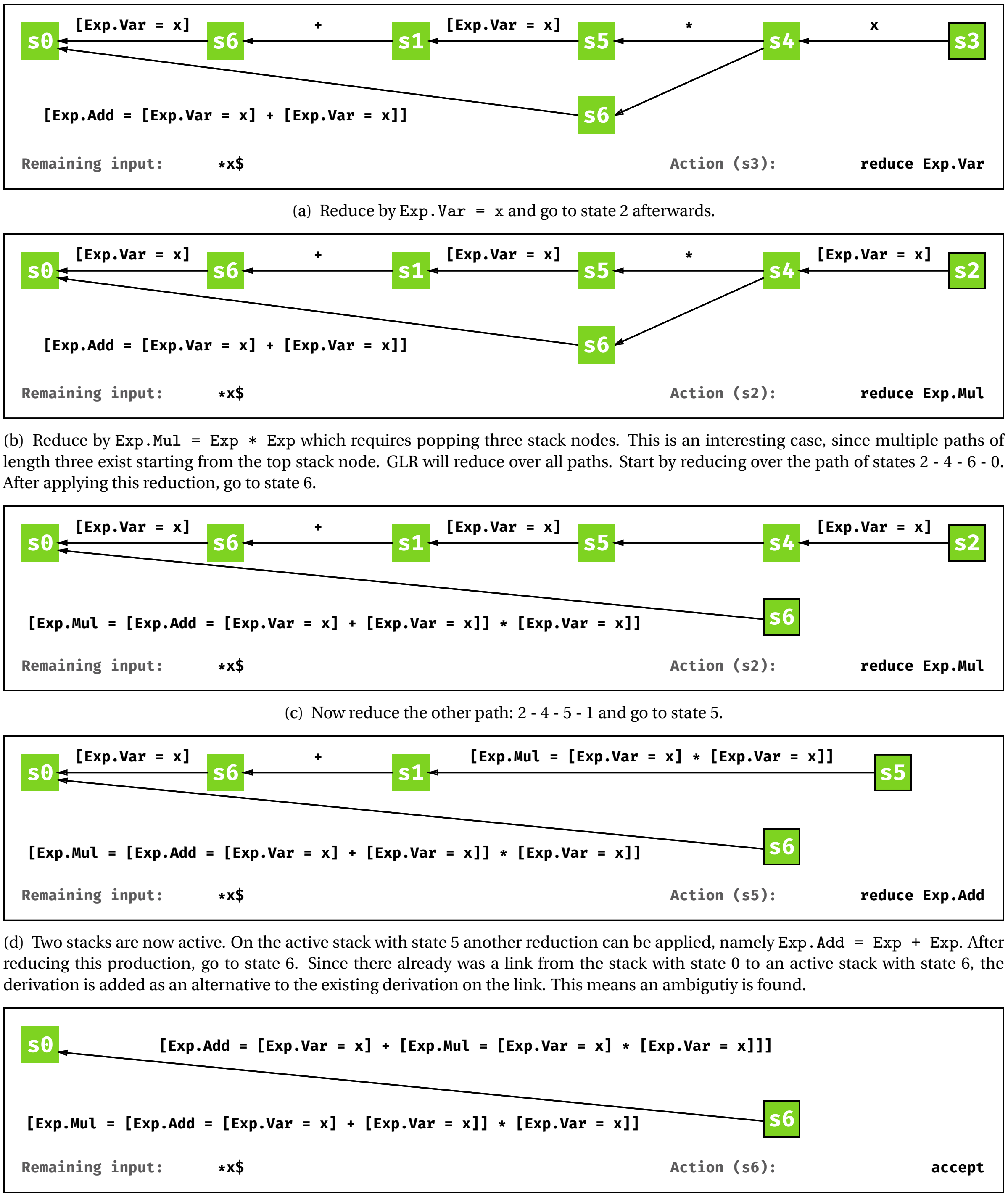
(c) Now shift on the bottom stack node. Since it also shifts into state 4, an extra link is added to the existing active stack with state 4 instead of creating a new one. This is called merging.

(d) Again only a single active stack. Proceed by shifting to state 3 (continuation in Figure 2.9).

Figure 2.8: GLR parsing `x+x*x`, proceeding after reaching the conflicting state as in Figure 2.5. Continues in Figure 2.9.

1987 [34, 35] in the context of natural language processing. Rekers first applied the algorithm to the parsing of programming language and his version with improved sharing from 1992 [29] later formed the basis for SGLR.

When for a certain parser configuration multiple actions are prescribed by the parse table, GLR parsing splits the stack in multiple stacks by performing each action on its own branch of the extended stack. Naively the collections of stacks could be implemented as a set; on every split of the stack it is copied and added to the set. This is basically the approach we took in the example of the previous section. When a conflict occurs, the parser tries both possibilities returning an ambiguous parse containing both the parse trees. This is inefficient however, since the number of stack grows exponentially in the number of ambiguities in the input. An efficient solution to this problem is the modelling of the stack as a graph. The stack starts with a root node and every time an element is added to the stack, it is added as a top node to the graph with a link to a stack node below it. In the case of non-determinism multiple top stack nodes can be added to a single bottom stack node. All paths from top stack nodes to the bottom stack node effectively represent a linear stack like in LR. Multiple branches of the stack can end up in the same parser configuration. In that case, such branches of the stack are merged by having edges to a common top stack node. This data structure that makes GLR parsing feasible in practice is called the graph structured stack (GSS) [36]. Synchronisation between the active top stack nodes takes place by first performing all possible reductions on all stacks, then performing all shift actions and then proceed with the next token and start a new parse round. We see how this works for the `x+x*x` example in Figure 2.8 and 2.9.


s0
[Exp.Var = x]
s6
+
s1
[Exp.Var = x]
s5
*
s4
x
s3
[Exp.Add = [Exp.Var = x] + [Exp.Var = x]]
s6
Remaining input: *x$
Action (s3): reduce Exp.Var
s0
[Exp.Var = x]
s6
+
s1
[Exp.Var = x]
s5
*
s4
[Exp.Var = x]
s2
[Exp.Add = [Exp.Var = x] + [Exp.Var = x]]
s6
Remaining input: *x$
Action (s2): reduce Exp.Mul
s0
[Exp.Var = x]
s6
+
s1
[Exp.Var = x]
s5
s4
[Exp.Var = x]
s2
[Exp.Mul = [Exp.Add = [Exp.Var = x] + [Exp.Var = x]] * [Exp.Var = x]]
s6
Remaining input: *x$
Action (s2): reduce Exp.Mul
s0
[Exp.Var = x]
s6
+
s1
[Exp.Mul = [Exp.Var = x] * [Exp.Var = x]]
s5
[Exp.Mul = [Exp.Add = [Exp.Var = x] + [Exp.Var = x]] * [Exp.Var = x]]
s6
Remaining input: *x$
Action (s5): reduce Exp.Add
s0
[Exp.Add = [Exp.Var = x] + [Exp.Mul = [Exp.Var = x] * [Exp.Var = x]]]
[Exp.Mul = [Exp.Add = [Exp.Var = x] + [Exp.Var = x]] * [Exp.Var = x]]
s6
Remaining input: *x$
Action (s6): accept

(a) Reduce by `Exp.Var = x` and go to state 2 afterwards.

(b) Reduce by `Exp.Mul = Exp * Exp` which requires popping three stack nodes. This is an interesting case, since multiple paths of length three exist starting from the top stack node. GLR will reduce over all paths. Start by reducing over the path of states 2 - 4 - 6 - 0. After applying this reduction, go to state 6.

(c) Now reduce the other path: 2 - 4 - 5 - 1 and go to state 5.

(d) Two stacks are now active. On the active stack with state 5 another reduction can be applied, namely `Exp.Add = Exp + Exp`. After reducing this production, go to state 6. Since there already was a link from the stack with state 0 to an active stack with state 6, the derivation is added as an alternative to the existing derivation on the link. This means an ambigutiy is found.

(e) Conclude a successful parse. Since the link from the accepting stack to the root stack contains multiple derivations, the result of this parse is an ambiguous parse forest with two alternatives, encapsulating both the derivations with the addition and multiplication on top.

Figure 2.9: Continuation of Figure 2.8.

2.3.1. The Algorithm

The GLR algorithm consists of the functions described in Algorithm 2 to Algorithm 12. The algorithm works similarly to the algorithm as described by Rekers, but the notation style and naming is adjusted to match the rest of the algorithms in this work. For example, *parsers* are renamed to *stacks*, *words* to *tokens*, *possibilities* to *derivations* and *tree nodes* to *parse nodes*. *kids* represent child parse nodes that form the right hand side for the production of a reduction. In the algorithms several helper functions are used:

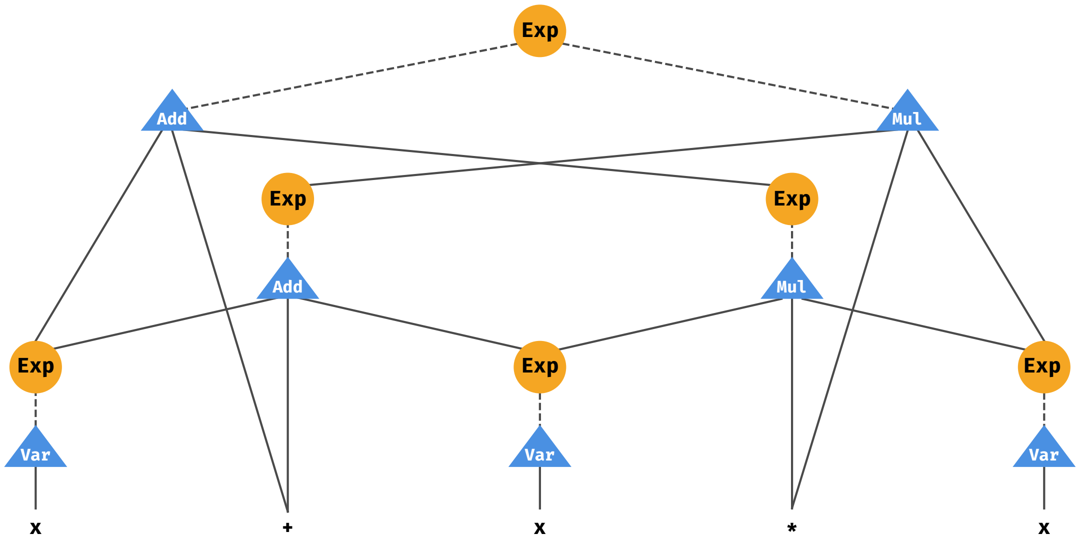


Figure 2.10: Ambiguous parse forest for $x + x * x$ using symbol nodes (circles) and rule nodes (triangles).

start-state Retrieves the start state from the parse table.

actions(s, t) Retrieves the applicable actions for state s and token t from the parse table.

goto(s, n) Retrieves the goto state for state s and non-terminal symbol n from the parse table.

production(r) Returns the production of rule node r.

kids(r) Returns the child parse nodes of rule node r.

cover(p) Returns the start position to end position parse node p covers from the input.

symbol(s) Returns the symbol of symbol node s.

concat($input$, \$) Concatenation of the end of file symbol after the input.

Since a successful GLR parse can result into ambiguous parses, parse forests are returned rather than parse trees. They can be modelled using three types of parse nodes:

Rule Nodes Represent derivations of productions. We depict productions as $A.C \rightarrow \alpha$ where the left hand side A is a non-terminal symbol, C is an optional constructor and α represents the right hand side symbols. A rule node represents a derivation of A, optionally labeled by C and has $|\alpha|$ edges to child parse nodes.

Symbol Nodes Represent non-terminals. An edge from a symbol node to a rule node means the rule node is a derivation for that symbol. The left hand side of the productions of such rule nodes equals the symbol of the symbol node. Symbol nodes can represent ambiguities. This is the case if there are edges from such ambiguity node to multiple rule nodes.

Term Nodes Represent terminal symbols (i.e. tokens) and are leafs in the parse forest.

An example of such a parse forest for the ambiguous $x + x * x$ parse example as seen before can be found in Figure 2.10. Circles represent symbol nodes and triangles represent rule nodes and are labelled by the constructors of their productions. The characters on the leafs are term nodes. Solid lines from a rule node connect the subtrees corresponding to the right hand side of the production for this rule node. Dashed lines from symbol nodes to rule nodes indicate such rule nodes are a derivation for the non terminal in the symbol node. In this example we see the top symbol node is ambiguous since it has multiple links to derivations. The notation style and parse forest representation are similarly applied to the SGLR algorithm as described in section 2.5.

Algorithm 2 The PARSE function of the GLR algorithm

```
PARSE(parsetable, input)
accepting-stack = $\emptyset$
init-stack = new stack with state start-state(parsetable)
active-stacks = {init-stack}
for all tokens $t \in$ concat(input, $) do
    PARSE-TOKEN($t$)
end for
if accepting-stack $\neq \emptyset$ then
    return the parse node on the only link of accepting-stack
else
    return error
end if
```

The main loop of the algorithm occurs in the PARSE function (Algorithm 2) which takes a parse table and an input stream of tokens as its parameters. Usually the input token stream of GLR is the output of a lexical scanner, which transforms a sequence of characters from a file into a token stream based on a lexical grammar. GLR differs from regular LR in the sense that multiple stack are maintained where LR only maintains one. The set of active stacks is initialised like LR with a single initial stack containing the start state of the parse table. The function PARSE-TOKEN (Algorithm 3) is called successively for each token, which handles all applicable actions for each active stack and the current symbol. $ depicts the end of file token which needs to be consumed by the parser to reach the accepting configuration.

Algorithm 3 The PARSE-TOKEN function of the GLR algorithm

```
PARSE-TOKEN($t$)
current-token = $t$
for-actor = active-stacks
for-shifter = $\emptyset$
rulenodes = $\emptyset$
symbolnodes = $\emptyset$
for all stacks $st \in$ for-actor do
    ACTOR(st)
end for
SHIFTER()
```

PARSE-TOKEN resets helper variables first and afterwards processes all the active stacks. Since reduce actions can lead to more stacks that needed to be handled, the active stacks are copied to *for-actor*. *for-actor* is a worklist of remaining stacks that need to be processed in a parse round. When REDUCER (Algorithm 6) creates a new stack, it is added to *for-actor* and processed in the same ACTOR (Algorithm 4) loop, and thus in the same parse round. During the handling of all applicable actions by ACTOR, shift actions are not directly executed but saved in *for-shifter*. After all stacks are processed and all reductions are applied, SHIFTER (Algorithm 8) performs the shift operations, essentially synchronizing the active stacks on shifts.

ACTOR acts on a single stack node and retrieves applicable actions from the parse table for the state of the given stack node and the current input token. Regular LR would fail if no applicable actions are found. However, since other active stacks might still be alive we do not directly report failure here. All applicable actions are handled differently based on their type. An accept action means the current stack is the accepting stack and is thus stored accordingly in *accepting-stack*. Reduce actions are handled immediately by DO-REDUCTIONS (Algorithm 5). As described before, the handling of shift actions is postponed.

When LR parsing performs a reduce for a production with $|\alpha|$ kids, the top $|\alpha|$ items of the linear stack can be directly retrieved and used to derive the production for the given reduce action. For GLR however, the stack is a graph and not linear. Therefore DO-REDUCTIONS finds all paths starting from the stack it is called on in the direction of the initial stack node of length α. For every path that is found, an actual reduction is performed by retrieving the child nodes on the links of the stack path and passing it to REDUCER. The call to REDUCER includes the goto state for the state of the stack at the bottom of the path and the non-terminal of the production so the parser knows to which state to transition after the reduction is applied.

Algorithm 4 The ACTOR function of the GLR algorithm

```
ACTOR($st$)
$s$ = state($st$)
for all actions $a \in$ actions($s$, $current\text{-}token$) do
    switch $a$ do
        case Accept
            $accepting\text{-}stack = st$
        case Shift($s'$)
            $for\text{-}shifter = \{\langle st, s' \rangle\} \cup for\text{-}shifter$
        case Reduce($A \to \alpha$)
            DO-REDUCTIONS($st$, $A \to \alpha$)
end for
```

Algorithm 5 The DO-REDUCTIONS function of the GLR algorithm

```
DO-REDUCTIONS($st, A \to \alpha$)
for all paths from stack $st$ to stack $st'$ of length $|\alpha|$ do
    $kids$ = the parse nodes on the links which form the path from $st$ to $st'$
    REDUCER($st'$, goto(state($st'$), $A$), $A \to \alpha$, $kids$)
end for
```

Algorithm 6 The REDUCER function of the GLR algorithm

```
REDUCER($st, s, A \to \alpha, kids$)
$rulenode$ = GET-RULENODE($A \to \alpha, kids$)
if $\exists st' \in active\text{-}stacks$ : state($st'$) = $s$ then
    if $\exists$ a direct link $l$ from $st'$ to $st$ then
        $symbolnode$ = the parse node on $l$
        ADD-DERIVATION($symbolnode$, $rulenode$)
    else
        $symbolnode$ = GET-SYMBOLNODE($A$, $rulenode$)
        add a link $l$ from $st'$ to $st$ with parse node $symbolnode$
        for all $st'' \in active\text{-}stacks \wedge st'' \notin for\text{-}actor$ do
            for all Reduce($A' \to \alpha'$) $\in$ actions(state($st''$), $current\text{-}token$) do
                DO-LIMITED-REDUCTIONS($st''$, $A' \to \alpha'$, $l$)
            end for
        end for
    end if
else
    $st'$ = new stack with state $s$
    $symbolnode$ = GET-SYMBOLNODE($A$, $rulenode$)
    add a link $l$ from $st'$ to $st$ with parse node $symbolnode$
    $active\text{-}stacks = \{st'\} \cup active\text{-}stacks$
    $for\text{-}actor = \{st'\} \cup for\text{-}actor$
end if
```

When REDUCER performs a reduction, a rule node is created using the production and the child parse nodes found on the links of the path on that has been reduced. This is done using the GET-RULENODE (Algorithm 9) helper function, which checks if a similar rule node was created in this round. In shift-reduce parsing the parser switches to a goto state after a reduction. REDUCER gets the goto state *s* passed via its arguments from DO-REDUCTIONS. Based on whether there is an active stack with the goto state and if so, whether a link between the two stacks exists (the stack REDUCER is called and an active stack with the goto state), different things happen:

Existing active stack with goto state, existing stack link The only thing that should happen is retrieving the symbol node from the existing stack link and add the rule node as a new derivation to it. Since symbol nodes always have at least one derivation, adding a derivation to an existing symbol node here means an ambiguity is found.

Existing active stack with goto state, no existing stack link A new link is created between the two stacks. The symbol node on this link is also newly created, it is for the non-terminal of the production and has as its first derivation the rulenode created at the beginning of REDUCER. When this happens, it could be that the newly created link introduces stack paths that are applicable to previously executed reductions and thus were missed at the time they were executed. Therefore, all active stack that are not still in the *for-actor* queue are processed again for applicable reduce actions using DO-LIMITED-REDUCTIONS (Algorithm 9).

No existing active stack with goto state A new stack with the goto state is created. This new stack gets a link to the stack REDUCER is called on with on it a newly created symbol node with the rule node as its first derivation. The new stack is added to both *for-actor* and *active-stacks*, which means it is handled for possibly new reductions in this parse round and also in the next parse round for all actions on the next token, respectively.

Algorithm 7 The DO-LIMITED-REDUCTIONS function of the GLR algorithm

```
DO-LIMITED-REDUCTIONS($st$, $A \to \alpha$, $l$)
for all paths from stack $st$ to stack $st'$ of length $|\alpha|$ going through link $l$ do
    $kids$ = the parse nodes on the links that form the path form $st$ to $st'$
    REDUCER($st'$, goto(state($st'$), $A$), $A \to \alpha$, $kids$)
end for
```

As described above, DO-LIMITED-REDUCTIONS is called to make sure no reductions are missed when a stack link is created between two existing stack nodes. It is similar to DO-REDUCTIONS except for the filter on paths that they go through the stack link that is created. This filter is required, because otherwise the paths that are found without going through this link are reduced multiple times, since they were already reduced before.

Algorithm 8 The SHIFTER function of the GLR algorithm

```
SHIFTER()
$active\text{-}stacks = \emptyset$
$position$ = position($current\text{-}token$)
$term\text{-}node$ = new term node with token $current\text{-}token$ and cover $\langle position, position\rangle$
for all $\langle s, st\rangle \in for\text{-}shifter$ do
    if $\exists st' \in active\text{-}stacks$: state($st$) = $s$ then
        add a link from $st'$ to $st$ with parse node $term\text{-}node$
    else
        $st'$ = new stack node with state $s$
        add a link from $st'$ to $st$ with parse node $term\text{-}node$
        $active\text{-}stacks = \{st'\} \cup active\text{-}stacks$
    end if
end for
```

After all reductions are applied, the parser can continue by performing shift operations in SHIFTER (Algorithm 8). Since all reductions are performed for this parse round, the list of active stacks is reset. For the current token a new term node is created. For all parse nodes their *cover* (the part of the input that is covered, further described later) is maintained and for a term node the cover has its begin and end position both as the position of the token in the input stream. All items that where added to *for-shifter* are now handled one by one. The items are pairs of states and stacks. For each pair a new stack link is created containing the term node. If there already is a stack node in *active-stacks* for the state, the link is added between the stack node of the pair and the existing stack node containing the state of the pair. If not, a new stack is created and it is added to *active-stacks*. When SHIFTER finishes, all possible actions for this parse round are executed and the parser can proceed to the next token and parse round, with a new set of active stacks.

Algorithm 9 The GET-RULENODE function of the GLR algorithm

```
GET-RULENODE($A \to \alpha$, kids)
if $\exists rulenode \in rulenodes$ with production($rulenode$) = $A \to \alpha$ and kids($rulenode$) = $kids$ then
    return rulenode
else
    create rulenode rulenode with production $A \to \alpha$, kids kids and cover COVER(kids)
    $rulenodes = \{rulenode\} \cup rulenodes$
    return rulenode
end if
```

GET-RULENODE is a helper function for the creation of rule nodes. It makes sure that existing rule nodes are re-used. That is the first mechanism Rekers introduced to improve sharing (*rule node sharing*). The existential checks are performed by maintaining a set of rule nodes per parse round and on the request of a rule node check whether this set does not already contain a rule node for the same production and with the same child parse nodes. If such rule node already exists, it is returned and no new one is created. If it does not exist, a new one is created and its cover is computed based on the cover of its kids using COVER (Algorithm 10).

Algorithm 10 The COVER function of the GLR algorithm

```
COVER(kids)
if $kids = \emptyset$ or $\forall kid \in kids$ : cover($kid$) = empty then
    return empty
else
    begin = the start position of the first kid with a non-empty cover
    end = the end position of the last kid with a non-empty cover
    return $\langle begin, end \rangle$
end if
```

COVER is a helper function for determining the cover (i.e. the subsequence of tokens a parse node covers from the input stream) for a rule node based on its child parse nodes. The result is an empty cover if there are no child parse nodes or if all child parse nodes have an empty cover. Otherwise, the cover spans from the start of the first non-empty cover to the end of the last non-empty cover.

Algorithm 11 The ADD-DERIVATION function of the GLR algorithm

```
ADD-DERIVATION(symbolnode, rulenode)
if $rulenode \notin$ the derivations of symbolnode then
    add rulenode to the derivations of symbolnode
end if
```

ADD-DERIVATION (Algorithm 11) adds a link from a symbol node to a rule node, which represents that the rule node is a derivation for the symbol of the symbol node.

GET-SYMBOLNODE (Algorithm 12) is a helper function for the creation of symbol nodes given a symbol and a derivation. It checks if a symbol node already exists for the same symbol and cover. If this is the case, the given derivation is just added to the existing symbol node using ADD-DERIVATION and thus indicates an ambiguity. Otherwise a new symbol node is created for the given symbol and its cover is set to the cover of the given derivation. Newly created symbol nodes are added to *symbolnodes* so they can be re-used later.

Algorithm 12 The GET-SYMBOLNODE function of the GLR algorithm

```
GET-SYMBOLNODE(symbol, rulenode)
if ∃symbolnode with symbol(symbolnode) = symbol and cover(symbolnode) = cover(rulenode) then
    ADD-DERIVATION(symbolnode, rulenode)
    return symbolnode
else
    create a symbol node symbolnode with symbol symbol, derivations { rulenode } and cover
    cover(rulenode)
    symbolnodes = {symbolnode} ∪ symbolnodes
    return symbolnode
end if
```

This re-using of symbol nodes is the second mechanism Rekers introduced for improved shared (*symbol node sharing*).

The parse loop continues until no active stacks are left or until all input tokens are processed. If at last an accepting stack is found, the parse tree on the link of the accepting stack to the initial stack node is the parse result. If no accepting stack is found, the parser reports failure.

2.3.2. Related Work

From the original introduction of GLR by Tomita several improvements are introduced by others to make GLR support the full class of context-free grammars correctly. The main contribution by Tomita was the efficient handling of stacks using the graph-structured stack. However, it contained a problem for certain grammars with nullable productions (i.e. productions with an empty right-hand side). When multiple stacks are active in parallel, a link introduced by one stack could lead to new reduction possibilities for other active stacks and they thus need to be revisited. In the original algorithm only paths where the first links is new were reconsidered. The solution introduced by Nozohoor-Farshi [26] reconsiders all paths with the new link. Additionally, it detects nullable productions and reuses its derivations, preventing the algorithm from looping on cyclic grammars. Rekers' algorithm as described in the previous section builds on top of this improvement of the original GLR algorithm by improving the sharing of parse nodes, which goes at the cost of searching through parse nodes. Another approach of solving the problems of the original Tomita algorithm is introduced by using RN parse tables, leading to the introduction of RNGLR parsing by Scott and Johnstone [2, 32, 33]. The RNGLR parsing approach is similar to Rekers' GLR version in how it improves sharing of the parse forest, but with a reduced amount of searching through nodes that is required. An overview and comparison of these variants of GLR parsing algorithms can be found in [9, 13].

2.4. Scannerless Parsing

Traditional parsing techniques first use a scanner in the lexical phase to strip layout and transform a stream of characters to a stream of tokens. Afterwards, the context-free analysis phase parses the stream of tokens into parse trees. With scannerless parsing [30, 31] there is no distinction between a lexical and context-free analysis phase. Scannerless parsing does not require a scanner, working directly on characters rather than tokens. It requires syntax definition in grammars to integrate lexical syntax and context-free syntax, leading to several advantages. First, lexical disambiguation can then use context. Furthermore, conflicts in the interface between lexical and context-free syntax when composing or embedding languages are avoided [3]. Declarativeness and maintainability of the syntax definition is also improved since only one grammar is used. However, scannerless parsing also has disadvantages. Lexical disambiguation conflicts are introduced such as longest match and during the preference of literals over identifiers. Additionally parse tables and parse forests become bigger and thus efficiency is impacted negatively.

Several syntax definition constructs are required to maintain expressiveness in grammars for scannerless parsing [3, 42]. Examples include regular expressions for describing lexical syntax, character classes, priorities, lists and modular grammars. Since SGLR parsing works on characters, a process called grammar normalization is required to transform such constructs into a normal form. The desired primitive form is one in which the grammar consists of productions with characters (or character classes) as the terminal symbols.

In addition to context-free syntax, grammars should also define lexical syntax and layout. Layout is usually represented by whitespace: spaces, tabs, newlines, etc. During grammar normalization scannerless parser generation typically injects optional layout between context free symbols. This enables parsing programs with layout between the constructs. The lexical syntax defines the derivation of lexical building blocks from characters. These building blocks include literals, identifiers, operators, etc. The next chapter describes an example of grammar normalization for the syntax definition formalism SDF3.

Usually the result of a successful parse is a parse tree or a parse forest in the case of ambiguities. Such parse trees represent the syntactic structure of the input as we have seen in the description of (G)LR parsing, including non-relevant information like e.g. literals. Often, parsers transform parse trees in abstract syntax trees (ASTs) in a phase called imploding. With scannerless parsing, even more non-relevant information is stored in the parse trees like layout and lexical structure, which influences imploding. During imploding, sub parse trees representing layout are discarded since they are not relevant in later processing by e.g. compilers. Additionally, tokenization, i.e. the process of forming lexical tokens from characters, happens during imploding. In scannerless parsing lexical constructs are actually sub trees with characters on the leaf nodes. Another example is the construct to represent a list of derivations in the grammar. While it is desired to have this represented as an actual list in the AST, the parse tree still represents such parts of the input as a tree. During imploding the parser can see when such a construct is used by looking up the properties of the production on a parse node and in case of a list construct the tree hierarchy is flattened into an actual list in the AST.

2.5. SGLR

SGLR [42] combines GLR with scannerless parsing. This section describes the algorithm in detail and the differences with GLR parsing.

2.5.1. Differences with GLR

The differences of SGLR with respect to Rekers' GLR with improved sharing are the following:

- Instead of working with token streams as input, the input is represented by a stream of characters. Therefore, a parse table that indicates actions for states and characters is required rather than for states and tokens. Furthermore this does not influence the working of the parsing algorithm.
- SGLR has support for reject productions. When a reject production is derived its result is excluded from the resulting parse tree. Reject productions enable solutions for the lexical disambiguation conflicts scannerless parsing introduces.
- It does not exercise the parse node sharing mechanisms *rule node sharing* and *symbol node sharing*.

2.5.2. The Algorithm

The original SGLR algorithm is described in pseudo code in Algorithm 13 to 19. The addition of reject productions changes the algorithm such that extra exceptions are required to be handled during reducing. Therefore, the *for-actor-delayed* variable is introduced. The description below is similar to Visser 1997 [42] besides one fix in the REDUCER function (Algorithm 17) which is described later and the modification to match the style of the other algorithms in this work.

Some helper functions are changed and added:

goto(*s*, *p*) Retrieves the goto state for state *s* and production *p* from the parse table rather than from a state and non-terminal symbol.

pop(*for-actor-delayed*) Pops the stack with the highest priority from *for-actor-delayed*.

The main PARSE loop (Algorithm 13) is similar to that of GLR except that the functions are called on characters rather than tokens.

PARSE-CHARACTER (Algorithm 14) replaces GLR's PARSE-TOKEN. The variable *for-actor-delayed* is added here which is also reset on every parse round. It contains the stacks that are added during a reduction and contain a state that is rejectable. If the states of such stacks are not rejectable, they are added to *for-actor* like in regular GLR. A state is rejectable if it is reachable by a reject production. That is, if the parser could transition into this state by following a goto action to this state after reducing by a reject production. The mechanism that excludes results from the resulting parse forest is present here. Namely, stack that contain links that are all rejected are skipped. The marking of links as rejected happens in REDUCER (Algorithm 17).

In the function REDUCER (Algorithm 17) many differences with GLR are present due to the handling of reject productions. Again, we identify three different scenarios for the workings of REDUCER:

Algorithm 13 The PARSE function of the SGLR algorithm

```
PARSE(*parsetable*, *input*)
*accepting-stack* = $\emptyset$
*init-stack* = new stack with state start-state(*parsetable*)
*active-stacks* = { *init-stack* }
**for all** characters $c \in$ concat(*input*, \$) **do**
    PARSE-CHARACTER($c$)
**end for**
**if** *accepting-stack* $\neq \emptyset$ **then**
    **return** the parse node on the only link of *accepting-stack*
**else**
    **return** error
**end if**
```

Algorithm 14 The PARSE-CHARACTER function of the SGLR algorithm

```
PARSE-CHARACTER($c$)
*current-character* = $c$
*for-actor* = *active-stacks*
*for-actor-delayed* = $\emptyset$
*for-shifter* = $\emptyset$
**while** *for-actor* $\neq \emptyset \wedge$ *for-actor-delayed* $\neq \emptyset$ **do**
    **if** *for-actor* = $\emptyset$ **then**
        *for-actor* = {pop(*for-actor-delayed*)}
    **end if**
    **for all** stacks *st* $\in$ *for-actor* **do**
        **if** $\neg$ all links of stack *st* rejected **then**
            ACTOR(*st*)
        **end if**
    **end for**
**end while**
SHIFTER()
```

Algorithm 15 The ACTOR function of the SGLR algorithm

```
ACTOR(*st*)
$s$ = state(*st*)
**for all** actions $a \in$ actions($s$, *current-character*) **do**
    **switch** $a$ **do**
        **case** Accept
            *accepting-stack* = *st*
        **case** Shift($s'$)
            *for-shifter* = $\{\langle st, s' \rangle\} \cup$ *for-shifter*
        **case** Reduce($A \rightarrow \alpha$)
            DO-REDUCTIONS(*st*, $A \rightarrow \alpha$)
**end for**
```

Algorithm 16 The DO-REDUCTIONS function of the SGLR algorithm

```
DO-REDUCTIONS(*st*, $A \rightarrow \alpha$)
**for all** paths from stack *st* to stack $st'$ of length $|\alpha|$ **do**
    *kids* = the parse nodes on the links which form the path from *st* to $st'$
    REDUCER($st'$, goto(state($st'$), $A \rightarrow \alpha$), $A \rightarrow \alpha$, *kids*)
**end for**
```

Algorithm 17 The REDUCER function of the SGLR algorithm

```
REDUCER($st$, $s$, $A \to \alpha$, $kids$)
$rulenode$ = new rule node with production $A \to \alpha$ and child parse nodes $kids$
if $\exists st' \in active\text{-}stacks$: state($st'$) = $s$ then
    if $\exists$ a direct link $l$ from $st'$ to $st$ then
        $symbolnode$ = the parse node at $l$
        add $rulenode$ to the derivations of $symbolnode$
        if $A \to \alpha$ is a reject production then
            mark link $l$ as rejected
        end if
    else
        $symbolnode$ = new symbol node for symbol $A$ with $rulenode$ as first derivation
        add a link $l$ from $st'$ to $st$ with parse node $symbolnode$
        if $A \to \alpha$ is a reject production then
            mark link $l$ as rejected
        end if
        for all $st'' \in active\text{-}stacks \wedge \neg$ all links of $st''$ rejected $\wedge st'' \notin for\text{-}actor \wedge st'' \notin for\text{-}actor\text{-}delayed$ do
            for all Reduce($A \to \alpha$) $\in$ actions(state($st''$), $current\text{-}token$) do
                DO-LIMITED-REDUCTIONS($st''$, $A \to \alpha$, $nl$)
            end for
        end for
    end if
else
    $st'$ = new stack with state $s$
    $symbolnode$ = new symbol node for symbol $A$ with $rulenode$ as first derivation
    add a link $l$ from $st'$ to $st$ with parse node $symbolnode$
    $active\text{-}stacks = \{st'\} \cup active\text{-}stacks$
    if rejectable(state($st'$)) then
        $for\text{-}actor\text{-}delayed$ = push($st'$, $for\text{-}actor\text{-}delayed$)
    else
        $for\text{-}actor = \{st'\} \cup for\text{-}actor$
    end if
    if $A \to \alpha$ is a reject production then
        mark link $l$ as rejected
    end if
end if
```

Existing active stack with goto state, existing stack link The only difference with GLR is that if the production is a reject production, the existing stack link that is found is marked as rejected.

Existing active stack with goto state, no existing stack link Similarly to GLR a stack link and symbol node with a single derivation are created. The new link is marked as rejected if the production is a reject production. Since the addition of this link can introduce new (possibly previously missed) paths, DO-LIMITED-REDUCTIONS is called on some stacks. The filter on active stacks is a bit more strict than with GLR. The extra criteria added are that only stacks that do not have only rejected links to it and stacks not in *for-actor-delayed* apply and should be processed by DO-LIMITED-REDUCTIONS.

No existing active stack with goto state Similarly to GLR a new stack node, stack link and symbol node with a single derivation are created and the new stack is added to *active-stacks*. However, depending on whether the newly create stack has a rejectable state, it is added to *for-actor-delayed* or *for-actor*. Note the difference between add operators to *for-actor-delayed* and *for-actor*. Since *for-actor-delayed* is a priority queue adding a stack is depicted with *push*. Since there is no priority for stacks on *for-actor* it is basically a set and adding stacks is thus depicted with ∪.

Thus, *for-actor-delayed* contains the newly created stacks that have a rejectable state. They are handled in PARSE-CHARACTER after the stacks in *for-actor*. This makes sure that no stacks are missed due to them being rejected by other reductions. If all rejectable states are processed after the other states, this problem is prevented. Since *for-actor-delayed* can contain multiple stacks, one of them can lead to reductions incorrectly rejecting one of the other stacks. Therefore, the *pop* operator in PARSE-CHARACTER should take a priority into account. It is not clear how this priority should be implemented. Here we see the SGLR algorithm does not explicitly re-uses rule nodes and symbol nodes but just creates them on demand. Furthermore this function contains a small fix on line 30 with respect to the original algorithm which incorrectly stated $for\text{-}actor = \{st'\} \cup for\text{-}actor\text{-}delayed$.

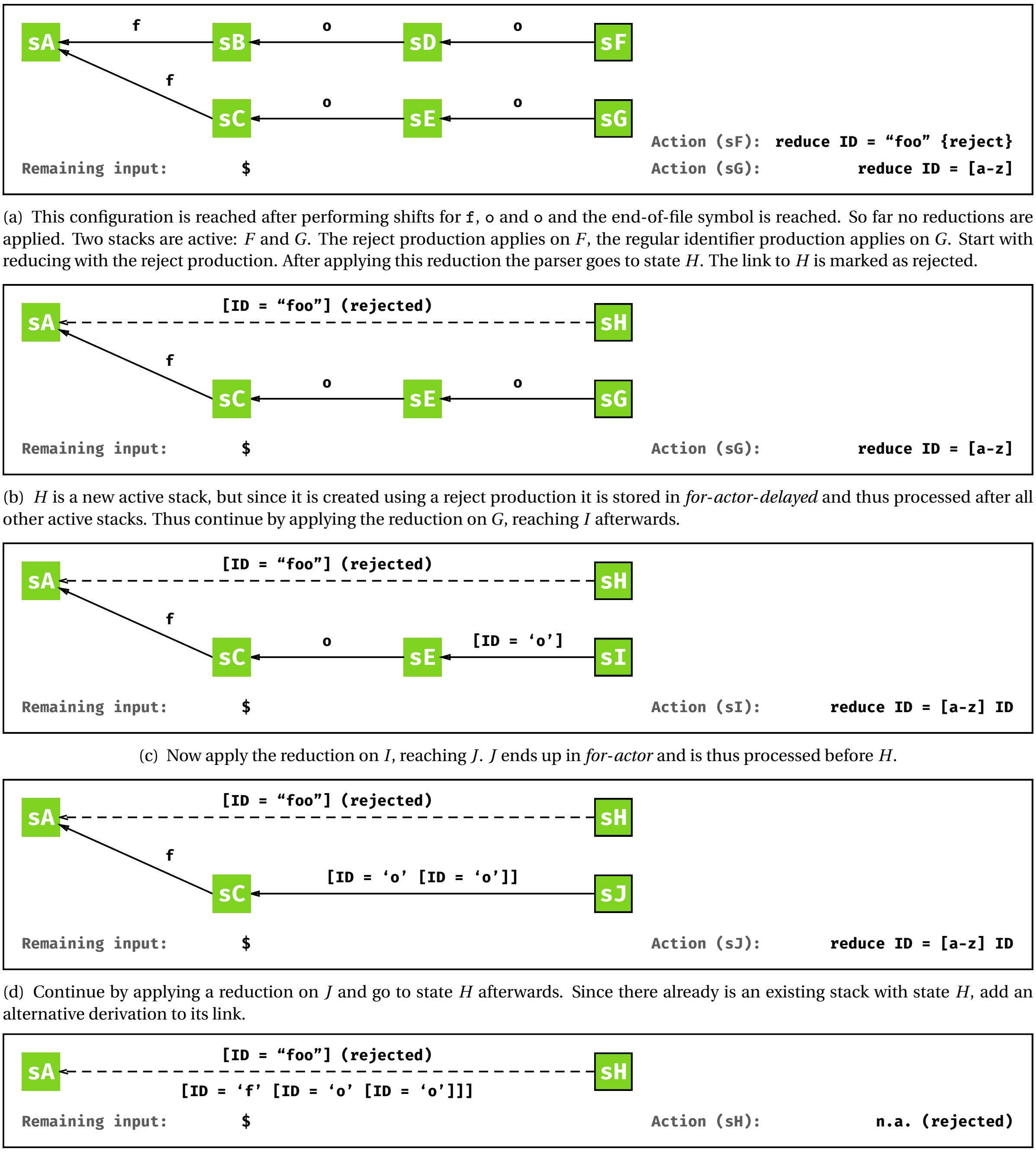
Algorithm 18 The DO-LIMITED-REDUCTIONS function of the SGLR algorithm

```
DO-LIMITED-REDUCTIONS($st$, $A \rightarrow \alpha$, $l$)
for all paths from stack $st$ to stack $st'$ of length $|\alpha|$ going through link $l$ do
    $kids$ = the parse nodes on the links that form the path form $st$ to $st'$
    REDUCER($st'$, goto(state($st'$), $A \rightarrow \alpha$), $A \rightarrow \alpha$, $kids$)
end for
```

Algorithm 19 The SHIFTER function of the SGLR algorithm

```
SHIFTER()
$active\text{-}stacks = \emptyset$
$termnode$ = new term node for $current\text{-}character$
for all $\langle s, st\rangle \in for\text{-}shifter$ do
    if $\exists st' \in active\text{-}stacks$: state($st$) = $s$ then
        add a link from $st'$ to $st$ with parse node $termnode$
    else
        $st'$ = new stack with state $s$
        add a link from $st'$ to $st$ with parse node $termnode$
        $active\text{-}stacks = \{st'\} \cup active\text{-}stacks$
    end if
end for
```

Figure 2.11 describes the parsing steps for grammar G_2 and input `foo`. Since there is a reject production `ID = "foo" {reject}`, the input is rejected and parsing fails.



(a) This configuration is reached after performing shifts for `f`, `o` and `o` and the end-of-file symbol is reached. So far no reductions are applied. Two stacks are active: F and G. The reject production applies on F, the regular identifier production applies on G. Start with reducing with the reject production. After applying this reduction the parser goes to state H. The link to H is marked as rejected.

(b) H is a new active stack, but since it is created using a reject production it is stored in *for-actor-delayed* and thus processed after all other active stacks. Thus continue by applying the reduction on G, reaching I afterwards.

(c) Now apply the reduction on I, reaching J. J ends up in *for-actor* and is thus processed before H.

(d) Continue by applying a reduction on J and go to state H afterwards. Since there already is an existing stack with state H, add an alternative derivation to its link.

(e) All stacks in *for-actor* are processed, so now the stack in *for-actor-delayed* is processed: the state with stack H. This stack is rejected, since all its outgoing links are rejected. It is thus skippend and since no other active stacks remain, parsing fails.

Figure 2.11: SGLR parsing `foo` with G_2.

Listing 2.2: Grammar G_2; identifiers of letters where `foo` gets rejected.

```
ID = [a-z]
ID = [a-z] ID
ID = "foo" {reject}
```

3

Syntax and Parsing in Spoofax

Spoofax [14, 15, 39] is a language workbench and its syntax related services are based on SGLR parsing. This work involves a reimplementation of the JSGLR parser used by Spoofax, and therefore this chapter describes Spoofax, its components and architecture related to syntax definition and the original parser implementation.

3.1. The Spoofax Language Workbench

Spoofax is a workbench for programming language designers that automatically generates the components of compilers based on declarative definitions using meta languages. One of such components is the parser, which takes the syntax definition of a language and text as input returning an abstract syntax tree as output. The meta language that is used in Spoofax for syntax definition is SDF3, which features character level LR parser generation. The parse tables are interpreted by JSGLR, an implementation of SGLR parsing in Java. Results of parses are abstract syntax trees (ASTs) that are propagated through the next phases of compilers like name binding, type analysis and transformations. These ASTs are also used for the derivation of presentational editor services in Spoofax like syntax highlighting, code folding and outlining. Furthermore they are used for semantic analysis allowing services as cross referencing and semantic checks.

3.1.1. Parsing Architeture

Spoofax allows modular syntax definition using SDF3 grammars. Multiple modules can be used to define a language and modules can depend on each other using imports. During parser generation these modules are combined and then transformed into a parse table. This process involves several normalization steps. Generating a parse table from a SDF3 grammar first transforms it into regular SDF constructs. Afterwards the SDF grammar is made more permissive by extending it with recovery rules (see Section 3.8). Also, extra rules are added to support code completion [1]. In the resulting SDF grammar both context-free and lexical productions are present in which context-free productions get the postfix `-CF` and lexical productions get the postfix `-LEX`. This intermediate format is called kernel syntax. After these normalization steps a parse table is generated using regular LR parser generation with character classes rather than tokens. JSGLR interprets such parse tables to parse actual input programs. It takes the generated tables and files or strings as input and outputs abstract syntax trees (represented with ATerms, see Section 3.2). The intermediate parse forest data structure is internal and is not exposed.

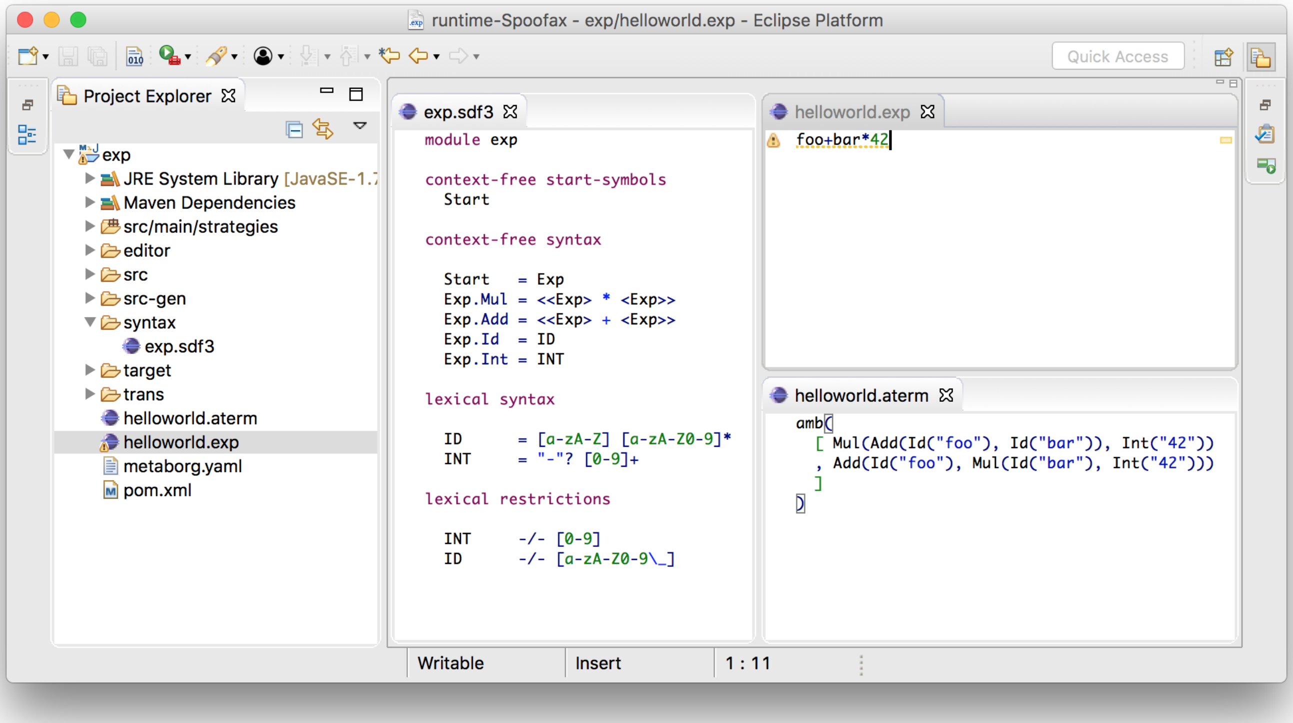
3.2. Abstract Syntax with Annotated Terms

Abstract syntax trees (ASTs) are represented in Spoofax using the Annotated Terms Format (ATerms) [22]. This format consists of the following constructs to represent tree data structures:

String A sequence of characters between double quotes. Strings are leafs in an ATerm tree. Example: `"foo"`.

Integer A constant integer. Like strings typically leafs in an ATerm tree. Example: 42.

Constructor Application An identifier followed by several other ATerms between parentheses and separated by commas. Example: `"Add(1, 2)"`.



```
module exp

context-free start-symbols
  Start

context-free syntax

  Start    = Exp
  Exp.Mul = <<Exp> * <Exp>>
  Exp.Add = <<Exp> + <Exp>>
  Exp.Id  = ID
  Exp.Int = INT

lexical syntax

  ID       = [a-zA-Z] [a-zA-Z0-9]*
  INT      = "-"? [0-9]+

lexical restrictions

  INT      -/- [0-9]
  ID       -/- [a-zA-Z0-9\_]
```

```
foo+bar*42
```

```
amb(
  [ Mul(Add(Id("foo"), Id("bar")), Int("42"))
  , Add(Id("foo"), Mul(Id("bar"), Int("42")))
  ]
)
```

Figure 3.1: Syntax definition and parsing of a simple expression language in Spoofax

List A sequence of ATerms between square brackets and separated by commas. Example: `[1, "foo", Add(2, 3)]`.

Tuple Similar to constructor applications and lists. Tuples however in contrast to lists have fixed sizes and use parentheses. They are basically constructor applications with empty constructors. Example: `"(1, 2, 3)"`.

ATerms are flexible and have several applications within Spoofax. Naturally they are used to represent parsed programs and further operations within the Spoofax pipeline (e.g. for semantic analysis) happen on the ATerm representations. Furthermore, parse tables are serialized to text files using the ATerm format. Grammars and their intermediate representations during normalization are also serialized to the ATerm format for portability purposes.

The format is textual and is used to represent trees. An abstract syntax tree in ATerms representing the example as seen in Figure 2.1 would be:

```
Add(Var("x"), Var("x"))
```

For Figure 2.2 it would be:

```
amb([
  Add(Lit("x"), Mul(Var("x"), Var("x"))),
  Mul(Add(Var("x"), Var("x")), Var("x"))
])
```

Ambiguous AST nodes are thus represented by an `amb` constructor containing a list with the ambiguous derivations.

3.3. Syntax Definition with SDF3

SDF3 [24] is the latest version of the syntax definition formalism SDF [10, 40] as used in Spoofax. It is a meta language to define the concrete syntax of a textual language. Typically these are (domain specific) programming languages. Besides the (context-free) grammar, it also defines lexical syntax instructing the parser how to implode concrete syntax to abstract syntax using constructors. While our definition of context-free grammars used the terms *terminals* and *non-terminals* to represent symbols in the language, SDF uses *sorts* to identify both. Since it is used for scannerless parsing, both context-free and lexical syntax can be present in a single file. Syntax definition is modular in the sense that multiple files can be used and import each other. Like context-free grammars is SDF closed under composition, meaning existing (parts) of grammars can be reused to extend languages or compose new ones. It features several high-level constructs for defining syntax like lists and optionals. Context-free disambiguation constructs are present to define safe operator precedence and associativity using priorities. Lexical ambiguities can be resolved using follow restrictions and reject productions. This makes the approach suitable for declarative, concise and expressive syntax definition. In the context of Spoofax it is also used to derive several IDE features: pretty printing, syntax highlighting and syntactic code completions [1].

SDF3 uses LR parser generation. In contrast to the traditional LR parse tables the goto actions are mapped on productions rather than non-terminal symbols. Thus where tradition LR parse tables have states as rows and terminals (for shifts and reduces) and non-terminals (for gotos) as columns, SDF parse tables have states as rows and character classes (for shifts and reduces) and productions (for gotos) as columns. Encoding goto actions for productions enables resolving priority and associativity disambiguation conflicts during parser generation.

The following sections describe the features supported by SDF3. They are grouped based on their context of applicability: on lexical syntax, on context-free syntax, on both (global), for kernel syntax or supportive to editor services.

3.3.1. Global constructs

These constructs can be used for both lexical and context-free syntax definition:

Sorts Identifiers represent the symbols in a grammar, which can be both non-terminals and terminals like literals. An example is `Exp` which could represent expressions.

Constructors Define the correspondence between the grammar's productions and abstract syntax trees. Constructors instruct during the imploding phase on how to transform a parse tree to an AST. An example is `Exp.Add` which would define the `Add` constructor for the `Exp` sort. Such notation is present on the left-hand side of a producton.

Productions Also called rules; the main building blocks of a grammar. Written in the productive form they describe the patterns of languages. They consists of a left-hand side indicating the sort it derives and a right-hand side containing the pattern it derives. They are separated with =. Sorts on the left-hand side can optionally contain a constructor. An example is `Exp.Add = «Exp> + <Exp»`. The left-hand side `Exp.Add` indicates that it derives the sort `Exp` and imploding a derivation of this production should be using the `Add` constructor ATerm. Productions can have several attributes appended to them using brackets and separated by commas. E.g. `Exp.Add = «Exp> + <Exp» {reject}` indicates a reject production.

Rejects Productions can be marked as reject productions using the `reject` attribute. This means that a derivation of the left hand side sort of such productions (and possible other derivations) when derived will be excluded form the resulting parse tree. See also Section 3.4.2.

Optionals In the right hand side of productions items can be marked optionally by appending the `?` operator. When used in context-free syntax the optional value is represented in the AST using the ATerms `None` or `Some(...)`.

Lists Repetition of sorts can be indicated with the postfix operators `*` (zero or more) and `+` (one or more). When used in context-free syntax such repetitions lead to ATerm lists when imploded to an AST.

3.3.2. Lexical syntax

The following constructs define lexical syntax which derive tokens from characters:

Character classes While scannerless parsing happens directly on the character level it would be inconvenient to define syntax using single characters, too. Therefore SDF uses character classes to concisely define a set of characters. E.g. `[a1_]` defines `a`, `1` and `_` and `[a-zA-Z]` defines all letters. Various operators can be used on character classes:

- **\/ (union)** Infix operator that results into a character class containing characters that are in either of the two classes it is called on.
- **/\(intersection)** Infix operator that results into a character class containing characters that are in both of the two classes it is called on.
- **/ (difference)** Infix operator that results into a character class containing characters that are in the first but not in the second class it is called on.
- **~ (complement)** Prefix operator that result into a character class containing all characters not in the class it is called on.

Sequences Character classes and sorts can be placed as a sequence to indicate the concatenation of several character classes and sorts.

Literals Using quotes is a shorthand for defining sequence of characters like literals or operators. E.g. `"null"` which is normalized to `[n][u][l][l]` (a sequence of character classes).

Case insensitivity When using single quotes for literal definitions they will be case insensitive. E.g. `'foo'` will be normalized to `[fF][oO][oO]`.

Follow restrictions Follow restrictions allows filtering of derivations that are followed by a certain lookahead, indicated with the `-/-` operator (“may not be followed by”). See also Section 3.4.1.

Alternatives Using the infix `|` operator alternatives can be encoded. E.g. `A | B` could be parsed both as `A` and `B`.

Layout The lexical syntax definition also contains the definition of layout. If defined, the SDF normalization process automatically adds optional layout between the elements of context-free productions such that the user does not have to.

3.3.3. Context-free syntax

In the context-free part of the grammar context-free productions can be defined which can also use the sorts of the lexical syntax. Additionally it features constructs to define context-free disambiguation:

Associativity Associativity of productions can be indicated using the attributes `left`, `right`, `non-assoc` and `assoc` (same as `left`). See also Section 3.4.2.

Priorities Priorities on productions are a transitive relation denoted with the > operator that indicate which alternative should be chosen over others with lower priority in case of an ambiguity. See also Section 3.4.2.

Prefer and avoid The keywords `prefer` and `avoid` can be used to prefer or avoid specific alternatives in case of an ambiguity. It is further explained in Section 3.4.5.

3.3.4. Kernel syntax

The normalization process of SDF3 applies on both lexical and context-free syntax and transforms the grammar into a uniform format in which there is only a single type of production: kernel syntax. The normalization step enables the usage of high level features in SDF3 which are then converted into lower level constructs and during imploding handled correctly to get the desired result. For example layout is added between elements of a context-free production and discarded during imploding. During the normalization all sorts are renamed to include the `-LEX` or `-CF` postfixes for lexical and context-free productions, respectively. In SDF3 one can also directly write kernel syntax by using sorts already having such postfixes. This is less convenient but gives the user more control over layout between elements.

	Parser Generation	Parsing	Post-parse filtering	Imploding
Productions	x	x		x
Constructors	x			x
Rejects	x	x		
Lists	x	x		x
Optionals	x	x		x
Character classes	x	x		
Sequences	x			
Literals	x			x
Case insensitivity	x			
Follow restrictions	x	x		
Alternatives	x			
Layout	x			x
Associativity	x			
Priorities	x			
Prefer and Avoid	x		x	

Table 3.1: An overview of SDF features and the phase of parsing in which they are handled or have impact on the implementation.

3.3.5. Editor services

Additional features can be used in SDF3 definitions to support the derivation of editor services from the generated parser like pretty printing and syntax highlighting.

Template productions Pretty printing can be assisted using template productions. Without template productions pretty printing an AST would simply concatenate all terms separated by some whitespace, e.g. spaces. With templates the whitespace used for pretty printing can be altered, e.g. with newlines or tabs.

Template options SDF3 documents contain a `template options` section. Here the `tokenize` option can be set to indicate which characters should be tokenized separately. Lets say an `if{` construct contains `if{` then this can be seen as a keyword `if` and the operator `{` or as a whole: `if{`. Forcing tokenization of `{` would enforce the tokenizer to use the first representation, independent of whether layout is present between the tokens. Then the tokenization heuristic can apply its classification as a keyword and operator instead of one keyword.

An overview of the features and in which parts of the parsing architecture they are handled or have impact on can be found in Table 3.1.

3.4. Disambiguation Filters

SGLR's scannerless characteristic to combine lexical and context-free analysis phases leads to the disadvantage that disambiguation conflicts on lexical syntax are introduced. Also, because of its ability to handle the full class of context-free grammars, parse results can be ambiguous and can contain undesired alternatives. Disambiguation filters [18, 37] are used to filter undesired alternatives from an ambiguous parse, solving the aforementioned problems. As seen in the previous section, SDF3 uses several declarative disambiguation constructs to address different classes of ambiguities. They are described in more detail in the following sections. Disambiguation in SDF3 is safe for operator precedence and associativity, which means that the constructs only remove alternatives from ambiguous derivations.

3.4.1. Follow Restrictions

Follow restrictions can be used to disambiguate the longest match problem. Lets assume we have the following declaration for identifiers that can contain lowercase letters and numbers:

```
ID = [a-z0-9]
```

For example if a part of the grammar accepts multiple identifiers after each other with optional layout between them, then the string `abc` would be parsed ambiguously. Since layout is optional possible derivations are `a` followed by `bc` or `ab` followed by `c`. We probably like this to be parsed as a single identifier. This can be resolved by using the following follow restriction:

```
ID -/- [a-z0-9]
```

It restricts an identifier to be followed directly by a character in the identifier production. This filters the shorter derivations of identifiers and thus achieves the longest match disambiguation.

3.4.2. Reject Productions

Productions marked as reject productions filter derivations for the left-hand side symbol of the production. Consider a grammar with the following productions:

```
ID = [a-z0-9]
KEYWORD = "true"
```

The string `true` would be parsed ambiguous as both an `IDENTIFIER` and a `KEYWORD`. Usually we would like to reserve keywords which can be achieved by rejecting the derivation of the keyword as identifier:

```
ID = [a-z0-9]
KEYWORD = "true"
ID = "true" {reject}
```

During parsing then three productions for `true` are derived. First, the intended derivation of `KEYWORD`. Second, the unintended derivation of `ID`. Last, the reject production for `ID`. The reject production makes sure all derivation for this string for the sort `ID` are filtered and only the correct `KEYWORD` remains.

3.4.3. Priority

In a separate section of a SDF3 document priority relations between productions can be defined. This relation is transitive and indicated with the > operator between (groups) of sorts with constructors or complete productions. As seen in Chapter 2 the following grammar is ambiguous for input `x + x * x`:

```
Lit.Lit = "x"
Exp.Mul = <<Exp> * <Exp>>
Exp.Add = <<Exp> + <Exp>>
```

However if we define the following priority relation we resolve the ambiguity by giving the `*` operator precedence over +:

```
Exp.Mul > Exp.Add
```

3.4.4. Associativity

In the previous section we described how to disambiguate `x + x * x`, but the same approach would not work for `x + x + x` which is also ambiguous. Disambiguation for operators works with associativity. Associativity declarations filter derivations in which recursive productions have a derivation by the same production as a direct child. The following types of associativity are available:

`left` Derivations where the recursive direct child is the right-most child are filtered out.

`right` Derivations where the recursive direct child is the left-most child are filtered out.

`non-assoc` Derivations with any recursive direct child are filtered out.

`assoc` Same as `left`.

3.4.5. Prefer and Avoid

While the other constructs disambiguate the grammar during parser generation, prefer and avoid are handled after parsing as a post-parse filter. When an ambiguity is parsed this means several productions derived the same string. These ambiguous alternatives could be marked with `prefer`, `avoid` or not. Derivations for productions with `avoid` are filtered out of the resulting tree, but only if there are other derivations. If `prefer` productions are present, all other derivations are filtered out.

3.4.6. Deep Priority Conflicts

The priority, associativity, prefer and avoid constructs disambiguate ambiguities in the grammar where unintended alternatives are a direct child of the top node containing the ambiguity. However, regular LR parser generation can not handle the resolution of conflicts that require unbounded depth analysis, called deep priority conflicts. While such conflicts can be resolved by rewriting the grammar and adding extra productions, this is not convenient for the users and conciseness decreases. SDF3 is capable of handling deep priority conflicts and the approach is described in [5].

3.5. Parsing with JSGLR

JSGLR [23] is the Java implementation of SGLR parsing that is used in Spoofax. It was ported form the original C implementation and extended with various features supporting the language workbench environment. While the implementation is successfully used in Spoofax, users reported performance issues and performance of the implementation and SGLR in general is little studied. This formed the initial motivation of this work. Additionally, the implementation of JSGLR was considered messy and hard to extend which hinders new parsing related experiments in the context of Spoofax. Therefore a new implementation that is modular and extensible was desired.

3.5.1. Additions to SGLR

JSGLR naturally implements the original SGLR algorithm. However in the context of Spoofax the parser includes more:

Reducing with lookahead While follow restrictions with one character are handled completely in the parse table, follow restrictions with multiple characters lookahead must be processed during parsing. This results in the ACTOR function as described in Algorithm 20 in which the function checkLookahead(l) checks if the next $|l|$ characters of the input sequence match the action's lookahead. SDF generates parse tables with specialized reduce actions including the lookahead, when applicable.

Post-parse filtering After parsing the resulting parse forest is filtered to reduce ambiguities. This is required for implementing the SDF prefer and avoid constructs.

Imploding Transforms parse forests to ATerm ASTs, removing the irrelevant information from parse forests like layout, literals, keywords and operators. Typically language designers are mostly interested in abstract syntax and static/semantic analysis and transformations are performed on ASTs, too.

Tokenization The lexical parts of an AST (tokens) are classified in a process called tokenization. Using various heuristics a distinction is made between layout, strings, keywords, operators, etc. This classification is used for syntax highlighting where every class is styled differently.

Error reporting and recovery While regular SGLR parsing only works on and is able to produce ASTs for syntactically valid input programs, in interactive IDE usage often invalid inputs are parsed. Error reporting and recovery introduces better support for such scenarios as further described in Section 3.8.

Completions On interactive usage in an IDE environment JSGLR supports syntactic code completions using placeholders. This means that while the user is typing syntactic code completions are automatically suggested.

This work does not include the implementation and performance optimization of error reporting, error recovery and completions.

Algorithm 20 The ACTOR function of the SGLR algorithm with reduce actions with lookahead

```
ACTOR($st$)
for all actions $a \in$ actions($s$, $current\text{-}token$) do
    switch $a$ do
        case Accept
            $accepting\text{-}stack = st$
        case Shift($s$)
            $for\text{-}shifter = \{\langle st, s\rangle\} \cup for\text{-}shifter$
        case Reduce($p$)
            DO-REDUCTIONS($st$, $p$)
        case ReduceLookahead($p$, $l$)
            if checkLookahead($l$) then
                DO-REDUCTIONS($st$, $p$)
            end if
end for
```

3.6. Representation of Characters

JSGLR and parse tables generated by SDF are capable of handling the 8 bit ASCII character set which means a range of 256 characters. Intuitively a byte representation ($[0,255]$, or $[-128,127]$ in the case of Java) would be the smallest and thus best representation for characters. However, we need an extra character to represent the end-of-file symbol. JSGLR and SDF solved this by using integers to have a single representation for both the ASCII characters ($[0,255]$) and EOF (256).

Entries in the parse tables are actions for classes of characters [41]. In JSGLR's internal parse table representation objects with arrays of integers represent such character classes. The arrays contain either a single integer or a multiple of two integers. An array with a single integer means it represents a class containing only the ASCII character the integer represents (e.g. `[120]` represents `x`). A multiple of two integers in the array means it represents ranges of characters where each pair of two integers is one range (e.g. `[65, 90, 97, 122, 256, 256]` represents `A-Z`, `a-z` and the end-of-file symbol, i.e. all letters and the end-of-file symbol).

3.7. Imploding

For JSGLR imploding means the internal representation of parse forests by means of Java objects are transformed to Java ATerm objects. It is intuitively a recursive process in which for a given node from the parse forest for a certain production imploding happens based on the type of the production:

Context-free Each child is recursively imploded. Only non-empty results for children are considered in the result. If a constructor is defined, an ATerm constructed is returned with the ASTs for the children. If no constructor is defined, a tuple is returned. In the cases of lists flattening takes place since lists are represented as trees in the parse forest.

Lexical For lexical parts of the parse tree a token is created and depending on the type an ATerm is returned, e.g. a string or number.

Literal Literals (including operators and keywords) are excluded from the AST and thus returns an empty AST.

Layout Layout is excluded from the AST and thus returns an empty AST.

The types of productions are derived during parser generation and parse table reading using heuristics.

3.8. Error Recovery

While parsing inputs in interactive environments like IDEs, the input is often in a syntactically invalid state. Parse error recovery is a technique to diagnose, report and recover from syntactically incorrect (parts of) programs. SGLR's nature makes extending it with recovery difficult. Since parsing is scannerless, errors are detected on the level of characters rather than tokens. This makes diagnoses harder since the space of possible errors is bigger. The generalized characteristic of SGLR parser makes recovery difficult, too. Naturally

GLR tries different parses in parallel where some of the attempts are allowed to fail in a successful parse. If all branches fail it is not trivial which of the branches caused the actual error.

JSGLR however does support error recovery [4, 16] and its approach is based on island grammars [25, 38]. Island grammars combines two types of productions: some are intended to precisely parse interesting parts of the input ("islands") and others are intended to skip parts of the input ("water"). This approach maps to error recovery because extra productions are added that can skip erroneous parts of the input. The approach automatically derives such productions from the original grammar in a process called relaxation, making it more permissive. The permissive grammars contains these productions that are then used to assist the parser in recovering from several types of errors. The algorithm is adjusted in such way to backtrack when an error is encountered. An increasing backward search space is then explored to find the minimal-cost recovery rule. A rule is then applied to skip over an invalid part of the input (consider it as "water") or to insert a literal to continue successful parsing afterwards.

4
A Modular SGLR Architecture

A modular architecture enables systematically evaluating the behaviour of its components. This work introduces such an architecture for SGLR parsing, which is found by decomposing the original algorithm into loosely coupled components. This chapter describes the identification of those components, the interfaces between them and their naive implementations from a software engineering point of view. The implementation is done in Java and named JSGLR2 [6], the successor of the original Spoofax parser implementation JSGLR.

4.1. Motivation

A modular architecture enables experimenting with several implemented variants for its components. This is desired in the context of performance optimizations since it enables systematic evaluation of variants of the architecture's components. One can replace a component with an (improved) variant, leave the rest of the architecture intact and evaluate changes in performance by repeating and comparing benchmarks for the different compositions. An additional benefit of a modular architecture is that multiple implementations for a component can be maintained within a single code repository. Also, modular architectures are more extensible and thus open possibilities for doing future work more easily.

This work introduces such an architecture for SGLR parsing and its related components in the context of Spoofax. It enables experimenting with various compositions of the parser's components, finally enabling to improve parsing performance within Spoofax. Additionally, this modular architecture opens future possibilities for doing parsing related research more easily. Finding such an architecture is non-trivial, since the SGLR algorithm and its implementation within Spoofax are only known as a whole, without an identification of underlying separate components and their dependencies.

The existing SGLR implementation in Spoofax, JSGLR, did not have a modular architecture. Its implementation was tightly coupled and hard to maintain or extend. Besides the original SGLR algorithm, the code contains extensions like error recovery and support for code completions which implementations are mixed up with each other without clear separation. This hindered code maintainability and performing new research with it. Additionally, users reported JSGLR failing on big inputs and its performance was little studied, which motivated performing performance optimization.

4.2. Decomposing SGLR

The modular architecture as introduced by this work was initially found by trying various decompositions of the original SGLR algorithm. The following decomposition by grouping the algorithm's functions was found most effective in making the architecture modular:

StackManager Handles creation of stack nodes, creation of stack links and rejecting of stack links.

ParseForestManager Handles creation of parse nodes, creation of derivations and adds derivations to parse nodes.

ReduceManager Handles the following functions from SGLR:

- DO-REDUCTIONS

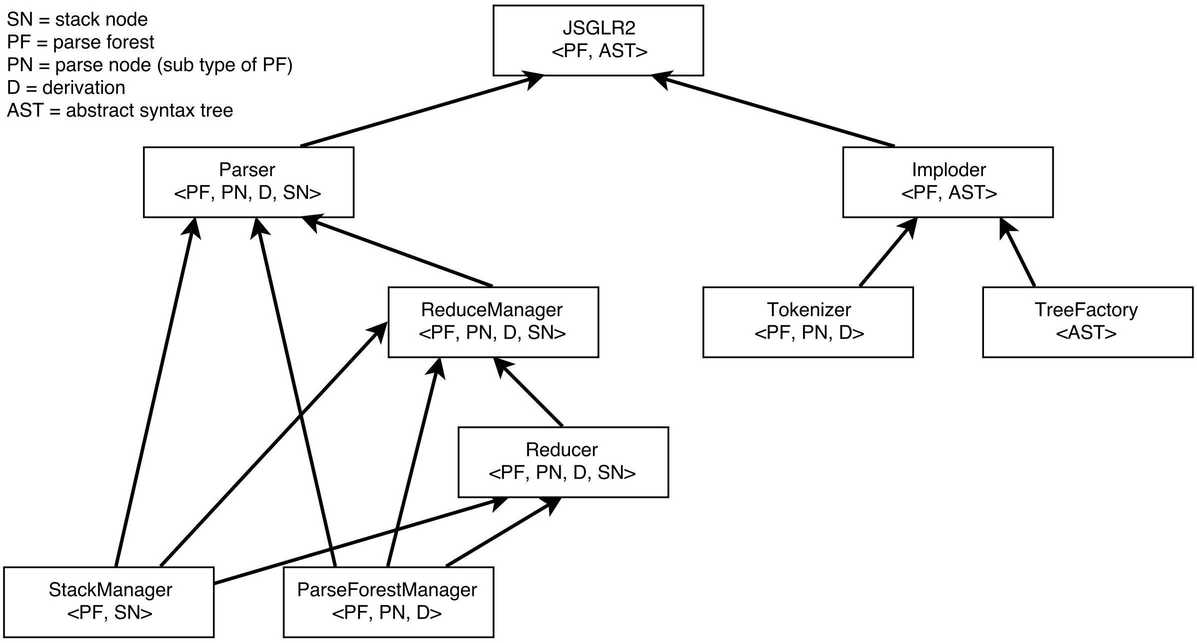


Figure 4.1: Components of the modular SGLR architecture and their dependencies on each other.

- DO-LIMITED-REDUCTIONS

Reducer Handles the following functions from SGLR:

- REDUCER

Parser Depends on implementations of StackManager, ParseForestManager and ReduceManager and handles the following SGLR functions:

- PARSE
- PARSE-CHARACTER
- ACTOR
- SHIFTER

The implementation of these functions is independent of the implementation of the components the parser depends on, but only depends on their interfaces.

Additionally to the SGLR parsing algorithm, in JSGLR2 the following components are present in the modular architecture:

Imploder Transforms parse forests into abstract syntax trees.

Tokenizer Labels the context-free leaf nodes of parse forests with tokens which represent a substring of the input and are mapped to lexical terms during imploding.

An overview of the components and their dependencies on each other can be found in Figure 4.1.

4.3. Components

While the previous section introduced a decomposition of the functions of the SGLR parsing algorithm, the following sections describe the components of SGLR and their requirements. Additionally, other components present in JSGLR2 are described. Besides parsing this includes the parse table, data structures the algorithm uses (like stack collections) and imploding. This chapter describes the components and a naive implementation for each of them. The naive implementations are considered the easiest to implement and require no post-processing of the parse table. The next chapter describes improved variants for some of the components.

4.3.1. Parse Forest

During parsing, the input's syntactic structure is stored in a parse forest. As earlier described, parse forests consists of three types of elements: parse nodes, derivations and character nodes. In the case of ambiguities, parse nodes have multiple derivations. Derivations are a list of other nodes (either parse nodes or character nodes), deriving the right hand side of a production as provided by the parse table via a reduce action. The naive implementation of the parse forest implements a class for each type, where parse nodes contain a Java `java.util.ArrayList` with derivations. While the parse forest is a data structure used during a parse, the modular architecture contains a ParseForestManager component. An implementation of its interface requires the operations of creating parse nodes, creating derivations, adding derivations to parse nodes and creating character nodes. For each variant of the parse forest data structure, a compatible ParseForestManager implementation is required to compose a parser working with such parse forests.

4.3.2. Stack

The parser uses a stack to maintain its state, which is a graph consisting of nodes and links. Stack nodes contain a LR state. Links are directed edges of the stack graph and contain a reference to a parse forest node. The naive implementation of the stack nodes contain a Java `java.util.ArrayList` with links to other nodes. Similarly to the parse forest, the stack requires an implementation of the StackManager for the parser to work with the type of stack. It requires the operations of creating stack nodes, creating stack links and adding links to nodes.

4.3.3. Reducing

For maximal modularity and extensibility, the parser has two components related to reducing. First, the ReduceManager component implements SGLR's DO-REDUCTIONS and DO-LIMITED-REDUCTIONS functions, handling the calling of Reducer for all applicable paths for a given active stack and reduce action. Second, the Reducer component implements SGLR's REDUCER function, handling the creation of parse nodes and derivations and possibly stack nodes and links for a given reduction path.

4.3.4. Character Classes

Parse tables for scannerless parsing use character classes to indicate the applicability of actions for a given parser configuration. Since during parsing the parse table is queried many times for applicable actions, the implementation of character classes is of importance for performant parsing. Ideally, character classes implementation use little memory and have fast character membership lookups. Section 3.6 described SDF and JSGLR using integers to represent the range of ASCII characters and the end-of-file symbol. ASCII characters are 8 bits and thus a byte representation for characters and classes would make sense since the memory footprint for their implementation could then be lower (8 bits for bytes versus 32 bits integers). Using this approach requires a specialized representation of the end-of-file symbol. A solution is to have two methods on the character class representation: `boolean containsCharacter(byte character)` and `boolean containsEOF()`. The latter is then used when the end of the input is reached and the first is used before, using characters from the input casted to bytes. Various experiments are carried out with this approach which in the end did not turn out to be beneficial which is expected to be due to the following reasons:

Java Byte Representation Only when using big arrays of bytes this actually saves memory with respect to using integers. In practice the parser works on single characters as tokens. If implemented as bytes they still use as much memory as an integer (32 bits).

Casting Characters to Bytes Java does not represent characters as bytes. Using a string from a file as input thus requires casting from characters to bytes during parsing.

Specialization of Classes Introducing the byte representation for characters requires implementing a specialized character class for the end-of-file symbol. This means more than two character class implementations are active during runtime. Then dynamic dispatch slows down since the JVM can only optimize when at most two types of classes are present[1].

JSGLR2 represents characters like JSGLR and SDF with the integer range $[0, 256]$. The character class implementation however is different. After empirically evaluating various combinations of implementations of character classes the most efficient turned out to be the combination of the following two specializations:

[1] https://shipilev.net/blog/2015/black-magic-method-dispatch/

Single Characters Single characters are represented using basically an integer wrapper class. Memory footprint is minimal and character membership lookup is fast, namely an integer comparison.

Multiple Characters A specialized class containing four 64 bit longs and a boolean represent the character classes containing more than one character[2]. Each long represents a quarter of the 256 ASCII range. The boolean represents whether the end-of-file marker is in the class. Lookup works with bit shifting. The `character ≫ 6` operation returns an integer value in the range $[0,4]$ for $0 \leq character \leq 256$. 6 indicates shifting to the 64 bits segment ($2^6 = 64$). A value in $[0,3]$ indicates a lookup in one of the longs. 4 means return the boolean value for end-of-file membership.

4.3.5. Applicable Actions on States

The main operation on states is the retrieval of applicable actions for a given configuration. We can distinguish two types of action retrieval here. First, if the parser handles an active stack in the ACTOR function it looks up the applicable shift, reduce and accept actions for the stack's state and the current input character. Second, after reducing, a new stack is added with the goto state. This goto state is retrieved from a state using a lookup by the production that was reduced. These types of retrieval of actions are conceptually mappings from characters to shift, reduce and accept actions and from productions to goto actions, respectively.

Shift, Reduce and Accept actions Several implementations for retrieving the applicable shift, reduce and accept actions are evaluated. The initial naive implementation created a new result list, looped over the actions in the state and for each state checked whether its character class applied to the given character and if it did, it was added to the result list. This is an inefficient approach since a new `java.util.ArrayList` is created. A slightly better approach is to prevent the creation of a new list. Returning a `java.lang.Iterable` that conceptually is a filter on the list of actions of the state does this, which is considered as the naive implementation in this work. Even better is using a for loop directly on the state's actions and do the filtering on the call site. The latter is most efficient. However, one could argue not to use this approach since code organisation would be better if a method in the state class implements the retrieval of actions. This latter is the reason for JSGLR2's interfaces to work with `java.lang.Iterable` rather than arrays.

Goto actions Since both productions and states have integer identifiers, the lookup for a goto state for a given production is a function or mapping from an integer to an integer. In the naive implementation the state consists of an array of goto actions where each goto representation has an array of production identifiers for which it applies. This representation matches the SDF parse table format. Performing the lookup for a given production involves a nested loop over both of the arrays.

4.3.6. Stack Collections

The parser maintains various collections of stacks, namely `activeStacks`, `forActor` and `forActorDelayed`, which each require a specific interface:

activeStacks Add stacks (in initialization, in REDUCER and in SHIFTER), add all stacks to forActor (in PARSE-CHARACTER), find by (goto) state (in REDUCER) and clear (in SHIFTER). The naive implementation is array based (using Java's `java.util.ArrayList`).

forActor Add stacks (in REDUCER), add all stacks from forActor (in PARSE-CHARACTER), pop/remove a stack (in PARSE-CHARACTER), contains stack check (in REDUCER) and clear (in PARSE-CHARACTER). The naive implementation is array based (using Java's `java.util.ArrayDeque`).

forActorDelayed Add stacks (in REDUCER), prioritized pop/remove a stack (in PARSE-CHARACTER), contains stack check (in REDUCER) and clear (in PARSE-CHARACTER). The naive implementation is a priority queue (using Java's `java.util.PriorityQueue`).

4.3.7. ForShifter Collection

The `forShifter` variable in the algorithm contains pairs of stacks and states indicating that after reductions in a parse round are applied, for each stack in a pair a shift can be executed into the state of the pair. The interface of this component is simple with only three operations: add (in ACTOR), iterate (in SHIFTER) and clear (in PARSE-CHARACTER). The naive variant is based on Java's `java.util.ArrayDeque`.

[2] This character class implementation is contributed by Michael Steindorfer.

4.4. Implementation

This section describes the Java implementation details of various aspects of JSGLR2, how this relates to the modular architecture and discusses software engineering choices.

4.4.1. Modular Architecture

The components of SGLR identified in the previous chapter are represented in Java using interfaces and abstract classes. An instance of the parser class is constructed based on implementations of the interfaces for the components it depends on. Different parser variants can be composed by constructing a parser with different variants of the components. The components depend on each other and the validity of a combination of components is ensured by using Java's generics mechanisms. For example, the parser class is instantiated with type parameters for the stack and parse forest representations: `IParser<StackNode extends AbstractStackNode<ParseForest>,`
`ParseForest extends AbstractParseForest>`. In its constructor it requires a StackManager and ParsForestManager for the same types `StackNode` and `ParseForest`, respectively. This implementation enables the modular architecture since abstract classes and interfaces represent the components and their characteristics and typing information ensures only valid compositions are allowed.

4.4.2. Stateless Parser Using a Parse Object

JSGLR extracts the state that is maintained during parsing out of the actual parser class into a separate parse class. This class contains the current parse position and the variables from the algorithm (`activeStacks`, `forActor`, etc.). A parser can be instantiated based on a parse table. After instantiation it can then be used for parsing various inputs without resetting. The latter was required after each parse with JSGLR and hindered concurrently parsing multiple inputs in parallel using a single parser instance. This contributes to the modular architecture since state is encapsulated in a single object and is not entangled with its components.

4.4.3. The Factory Pattern

Using the factory pattern achieves modularity for parse tables and their implementations of states, character classes and actions. While loading a parse table, variants for the factories for states, character classes and actions are provided. This enables the construction of parse tables with various (combinations of) implementations of the underlying data structures.

4.4.4. Imploding

A successful parse produces a parse forest. The parse forest represents the input by successive hierarchical applications of productions from the language grammar. For further processing of parse results we are interested in abstract syntax trees (AST's) rather than parse forests, since parse forest contain subtrees representing layout, punctuation, etc. Imploding in JSGLR2 works by recursively traversing the parse forest. It starts with the context free root of the forest and recursively operates on subtrees and combines implode results of subtrees. Once the boundary between context-free and the lexical part is reached it just takes the part of input covered by the concerned parse node on this boundary. Such parse node is the root node of a lexical subtree and taking the substring of the input it represents is sufficient, e.g. to create a token in the case of a lexical production or skipping in the case of layout or a literal. The post-parse filtering required for implementing SDF's prefer and avoid construct is handled during imploding, too. When retrieving derivations for a possibly ambiguous context-free parse node, the alternatives' prefer and avoid flags are taken into account.

4.4.5. Differences with Respect to Original JSGLR

Furthermore JSGLR2 has several implementation details handled differently then the original implementation:

- JSGLR2 supports reduce actions with lookahead where the lookahead is more than one character. This support was already present in SDF and generated parse tables but the original parser lacked support.

- Parse forest nodes contain the positions of the substring of the input they represent. This supports more efficient imploding.

4.5. Testing

To ensure a correct implementation of the SGLR parsing algorithm and (backwards) compatibility with JSGLR and SDF, JSGLR2 contains several types of automated and extensible tests. The valid implementation of SDF3's functionality is ensured by writing small tests with parse examples for all its features (see Chapter 3.3). Additionally, some larger tests are executed on real world programming languages parse tables and inputs.

For the implementation of SDF features imploding is important and does not have well defined behaviour. To validate backwards compatibility an automated test suite is created in which results of JSGLR and JSGLR2 are compared.

Tests are written using the JUnit[3] testing framework. Execution of tests is incorporated in the Maven build pipeline. Tests for SDF grammars are automated in the sense that grammars are converted to parse tables during test execution. This ensures new tests can easily be added because no manual generation of parse tables in Spoofax is required.

4.6. Observing

JSGLR2 implements the observer pattern. This allows the implementation of an interface that enables subscribing to the following events that occur during parsing:

- Start a parse.
- Parse a character.
- Create stack node.
- Create stack link.
- Reject stack link.
- Observe stack collections.
- Execute ACTOR function.
- Skip a rejected stack.
- Add an element to `forShifter`.
- Execute DO-REDUCTIONS function.
- Execute DO-LIMITED-REDUCTIONS function.
- Execute REDUCER function.
- Find direct link between stacks.
- Accept a stack.
- Create parse node.
- Create derivation.
- Create a character node.
- Add derivation to parse node.
- Execute SHIFTER function.
- Conclude successful parse.
- Conclude failing parse.

Several applications for observing are implemented in JSGLR:

[3] http://junit.org/junit5/

Logging The most simple implementation of the observer is one that logs every event. Adding this observer is helpful during development and debugging since then logs indicate the exact execution steps of the parser.

Measurements E.g for investigating the parser behaviour and confirming expectations about data structures it is helpful to measure (intermediate) parser configurations during parsing. Several types of these measurements are used to support claims that are made in this work. Performing measurements using the observer pattern is convenient because for a certain type of measurement the observer can hook into exactly the desired data structure on each step of the parser and than extract the required data from it.

Benchmarking An observer can monitor data structures at all states. This is useful for benchmarking since the operations on data structures can be collected and executed again after parsing to result in an isolated benchmark. This ensures that for several implementation of a data structure their performance can evaluated properly without being executed and impacted during actual parsing.

Visualization Since the observers can monitor each step a parser executes we can use this to reproduce parses in an interactive visualized environment. See also Section 4.7.

The aforementioned applications of the observer pattern on JSGLR2 validate its modularity. The parser can be extended and used for various applications while maintaining an organized implementation and code base. The original implementation had flags to enable logging or measurements and this decreased code organization.

4.7. Visualization

During development and debugging JSGLR2, visualization of stacks and parse forests turned out to be very useful. Using the observer pattern all actions executed during a parse are logged and afterwards serialized to a JSON file. A web app then interprets this JSON file. This web app contains two visualizations, one for both the stack and parse forest. Visualization is performed using the vis.js library [4] that supports visualizing graphs. Additionally the app contains several buttons to interactively step through parser execution. Every step extends the parser configuration by adding stack and parse nodes, links, mark stack links red if they are rejected or green if they are accepted, etc. Via the play button each step is applied automatically with a small time interval which results into an animation. A screenshot of this tool can be found in Figure 4.2. Several other tools exist for investigating GLR parsing algorithms, like the Grammar Tool Box (GTB) [11, 12] and Parser Animation Tool [9]. The visualization tool introduced by this work has less features, but it specific to SGLR and validates the modular architecture since it could be implemented as an extension of the parser without changing core parser code.

[4] http://visjs.org/

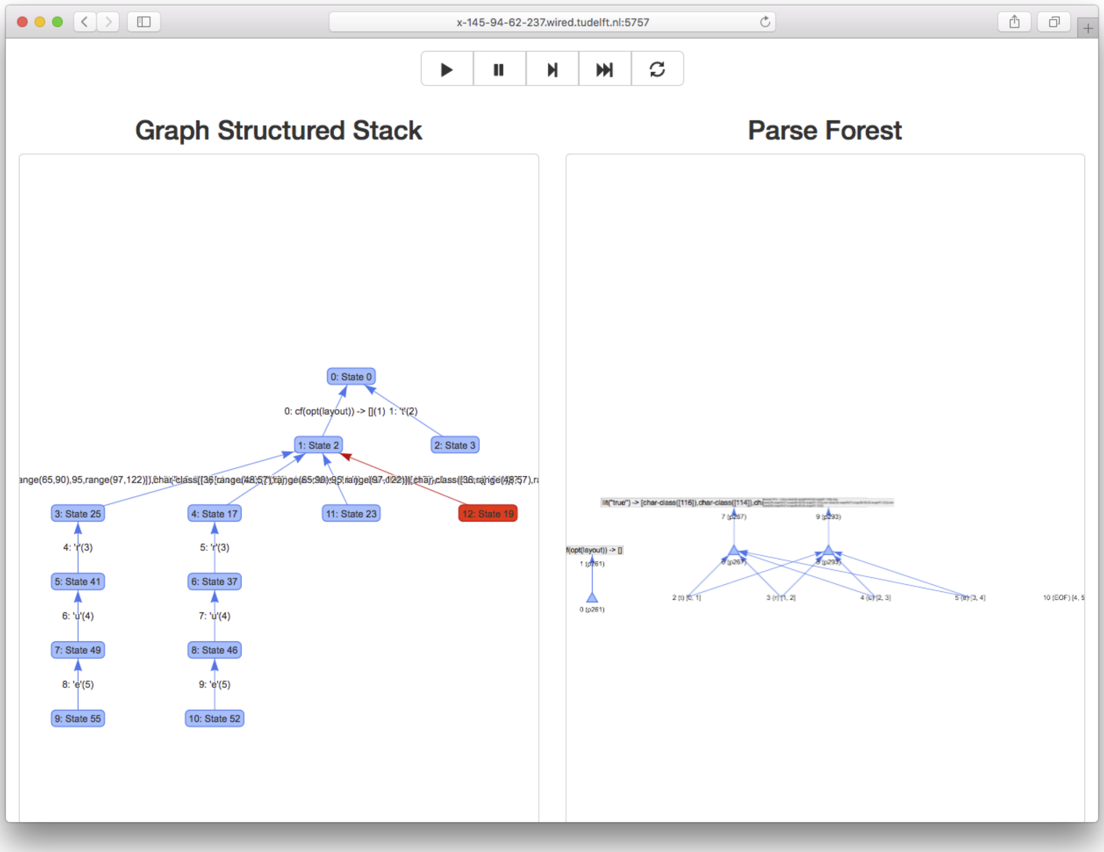


Figure 4.2: A screenshot of JSGLR2's web tool for visualization of stacks and parse forests.

5
Performance Optimizations

The implemented modular architecture in JSGLR2 enables experimenting with several variants for its components. This chapter describes the implementation of such variants that are intended to improve the overall parser performance.

5.1. Parse Table Data Structures

During many steps the parser takes the parse table is queried for e.g. the retrieval of actions. Therefore the internal representation of states and their applicable actions impacts performance. The improvements are discussed below:

Retrieval of Shift, Reduce and Accept Actions with Binary Search on Grouped Actions Character classes have an optimized representation with two specializations: one for single characters (an integer wrapper class) and one for ranges (containing four longs and a boolean to cover the [0,256] range). This increases efficiency of character class membership lookups. Additionally an improvement is made in the way character classes are grouped. A naive implementation contains lists of character classes and action pairs in which retrieving applicable actions requires a linear search. JSGLR2 improves on this by introducing a binary search based on grouping applicable actions and sorting the disjoint groups on ranges of the character classes they are applicable for. The grouping of actions by character ranges and sorting requires a processing step during parse table loading. The character classes are not used for these groups of actions. Every group is disjoint for a character range. The binary search finds an applicable range for the given character and returns all actions at once if it is found.

Retrieval of Goto Actions with HashMap for Production/State Relation Looking up a goto state for a given production is done with a HashMap lookup rather than nested for loops. This also requires a post processing step during parse table loading.

5.2. Elkhound: Hybrid LR/GLR Reducing

The Elkhound [21] paper presents several improvements for GLR parsing. Besides general software engineering improvements, it introduces a concept characterised as a hybrid form of LR and GLR. Essentially this idea means that the parser switches to a regular LR parser when possible and thus discards the overhead involved by GLR parsing as much as possible. Since LR parsing only works for deterministic inputs, Elkhound requires a mechanism to detect when the currently processed part of the input is deterministic and GLR overhead can skipped. Elkhound does this by maintaining the deterministic depths of stack nodes. While the stack used for GLR parsing is a graph, some active stacks might actually be linear for a certain length. This length is called the deterministic depth. When a reduction is applied and it requires a number of child parse nodes (i.e. the length of the right-hand side of the production) that is equal to or lower than the deterministic path depth, a shortcut can be taken regarding the retrieval of stack paths. They can be retrieved linearly and put directly in an array rather than via a breadth-first search through the graph-structured stack searching for paths of the required length in a linked list like representation. Furthermore, if there is only a single active stack, the parser is really in a deterministic part of the input and the relatively expensive checks that happen in REDUCER can be skipped. Thus the Elkhound parser can apply LR reducing on a stack when the following two criteria are met:

1. The deterministic depth of the stack should be at least as long as the number of parse nodes that are required for the reduction (i.e. the number of elements on the right hand side of the production which is used for the reduction).

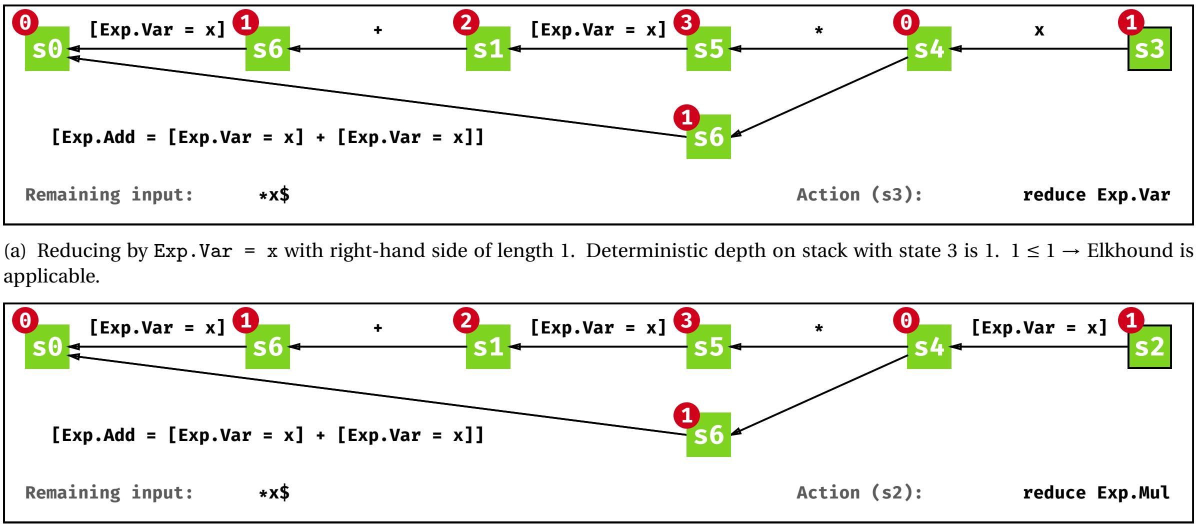
2. There are no other active stacks.

If only the first criterium is met the approach still has a benefit, because the path retrieval is faster. While LR parsing actually pops stack nodes when reducing, GLR does not since the active stack from which is reduced might be merged into by another reduction in the same parse round. Therefore, when GLR parsing performs a reduction in a parse round with a single active stack, afterwards it will have two active stacks. Then, when reducing on the second stack Elkhound would not apply since multiple active stacks are present. However, there was only one active stack and the parse could thus stay LR. To overcome this, implementing Elkhound requires an additional change: when a single action is performed on a single active stack, the active stack on which the action was performed is discarded and only a single active stack remains (with the goto state from the production). This is essentially the same as popping stacks from a linear as in regular LR parsing.

5.2.1. Deterministic Depth

The key change required for the Elkhound approach is the maintenance of the deterministic depths of stacks, which works as follows. The initial stack node has deterministic depth of 0. Every time a stack node is create with a link to its parent, its deterministic depth is set to the deterministic depth of its parent increased by one. When a second link is added to a stack node in the future, its deterministic depth is reset to 0. Additionally, the stack nodes connected to it from the future now need to be recomputed. This could be realised by maintaining bi-directional links on stacks. Adding the links in the direction of the future allows updating deterministic depths by traversing the stack paths to the future. Without bi-directional links, only traversing from top stack nodes to the root is possible. Since maintaining bi-directional links is expensive, deterministic depths are updated by a recursive process, only on the active stacks. The recursion stops when the root stack node is reached or when a stack node with deterministic depth 0 is reached.

Figure 5.1 shows the deterministic depth for each stack node in the red circles for two configurations of the stack as seen in Figure 2.9. In the left configuration (Figure 5.1(a)) the next step is reducing on stack node *s*3 by a production with a right-hand side of length one. Elkhound is applicable here since the deterministic depth on stack node *s*3 is also one and thus large enough. On the right configuration (Figure 5.1(b)) Elkhound is not applicable. The next step is reducing by a production with a right-hand side of length three which is bigger than the deterministic depth of stack *s*2, which is one. This case indeed is not deterministic, since there are two paths of length three from stack node *s*2 on which the reduction can be applied.



(a) Reducing by `Exp.Var = x` with right-hand side of length 1. Deterministic depth on stack with state 3 is 1. $1 \leq 1$ → Elkhound is applicable.

(b) Reducing by by `Exp.Mul = Exp * Exp` with right-hand side of length 3. Deterministic depth on stack with state 2 is 1. $3 \nleq 1$ → Elkhound is not applicable.

Figure 5.1: GLR configurations from Figure 2.9 with deterministic depths added.

5.2.2. Implementation

Elkhound is implemented in JSGLR2 with variants for the following components:

Parser When only a single stack is active and only one shift or reduction path is applicable, the active stack is popped and only one active stack remains active. In all other cases, fall back to the regular GLR parser.

Stack An extended stack representation that includes the deterministic depth counting. For the parser to use this representation, a different implementation for the StackManager is used.

ReduceManager A subclass of the ReduceManager is implemented which only overrides the DO-REDUCTIONS method. It requires the parser to use the extended stack representation. The adjusted DO-REDUCTIONS functions is described in pseudo code in Algorithm 21.

Algorithm 21 The updated DO-REDUCTIONS function for Elkhound

```
DO-REDUCTIONS($st$, $A \to \alpha$)
if deterministic-depth($st$) ≥ $|\alpha|$ then
    $deterministic\text{-}path$ = deterministic path from stack $st$ to stack $st'$ of length $|\alpha|$
    $kids$ = the parse nodes on the links of $deterministic\text{-}path$
    if $|active\text{-}stacks| = 1$ then
        REDUCER-LR($st'$, goto(state($st'$), $A \to \alpha$), $A \to \alpha$, $kids$)
    else
        REDUCER($st'$, goto(state($st'$), $A \to \alpha$), $A \to \alpha$, $kids$)
    end if
else
    for all paths from stack $st$ to stack $st'$ of length $|\alpha|$ do
        $kids$ = the parse nodes on the links which form the path from $st$ to $st'$
        REDUCER($st'$, goto(state($st'$), $A \to \alpha$), $A \to \alpha$, $kids$)
    end for
end if
```

The Elkhound version of DO-REDUCTIONS identifies multiple scenarios for reducing based on the deterministic depth of the stack *st* it is called on and on the number of active stacks. Assume we are reducing for production $A \to \alpha$ with $|\alpha|$ elements on its right-hand side. LR reducing is only possible when the deterministic depth of the current stack is at least $|\alpha|$ and when there is only one active stack. Therefore, the following scenarios for reducing arise:

LR: deterministic-depth(st) ≥ $|\alpha|$, $|active\text{-}stacks| = 1$ LR reducing can be applied by calling REDUCER-LR (Algorithm 22). This function skips the overhead GLR requires and just creates a new stack node, a new stack link and with on the stack link a new symbol node with a derivation. The newly created stack is also added to *active-stacks* (for processing in the next parse round) and to *for-actor* (for processing in the current parse round. Lastly, if the concerned production is a reject production, the stack link is marked as rejected.

Determinstic-GLR: deterministic-depth(st) ≥ $|\alpha|$, $|active\text{-}stacks| > 1$ Because there are multiple active stacks, we can not apply LR here, since the reduction of the deterministic path can influence the handling of other active stacks. However, since the path is deterministic we can use a faster path retrieval method here.

Non-deterministic-GLR: deterministic-depth(st) < $|\alpha|$ Since the deterministic depth is lower than the length of the path that is required for the reduce, there are multiple paths of length $|\alpha|$ and thus we need to apply regular GLR.

Algorithm 22 The updated REDUCER-LR function for Elkhound

```
REDUCER-LR($st$, $s$, $A \to \alpha$, $kids$)
$rulenode$ = new rule node with production $A \to \alpha$ and child parse nodes $kids$
$symbolnode$ = new symbol node for symbol $A$ with $rulenode$ as first derivation
$st'$ = new stack with state $s$
add a link $l$ from $st'$ to $st$ with parse node $symbolnode$
$active\text{-}stacks = \{st'\} \cup active\text{-}stacks$
$for\text{-}actor = \{st'\} \cup for\text{-}actor$
if $A \to \alpha$ is a reject production then
    mark link $l$ as rejected
end if
```

5.3. Parse Forest Reduction

Since SGLR parsing is scannerless, for each character of the input a parse node is created. Lexical tokens in the imploded AST are represented in the parse forests by lexical subtrees. Its lexical structure and the corresponding parse trees are convenient during debugging but often not required during parsing in practice, e.g. when an AST is desired as final output. Therefore, the creation for lexical subtrees can be limited to only create parse nodes for the parent nodes of lexical subtrees. This is sufficient for the imploding phase to determine when to create lexical terms in the AST and when to skip a subtree as being layout or literals. In this way the number of parse nodes can be significantly reduced.

5.4. Stack Collections

SGLR uses three stack collections: `activeStacks`, `forActor` and `forActorDelayed`. No improved variants for `forActorDelayed` are considered, since its maximal size over all test sets was only 2 and thus no significant improvements are expected to be made here. Both `activeStacks` and `forActor` are collections that grow (they should have an add operation), that are iterated over and have membership operations (REDUCER looks up active stacks by state and checks whether stacks are in `forActor` of `forActorDelayed`). The naive implementations are array based which requires linear search for the membership operations. The ideal data structure for these collections has fast add, iterate and lookup operations. One approach is to maintain both an array and a hashmap. This goes at the cost of maintaining two collections, but introduces $O(1)$ lookups. This works for `activeStacks`, but not for `forActor` since it requires removing stacks one by one. `activeStacks` is only cleared once per parse round. The third variant considered is a linked hashmap. Each value of such maps contains a wrapper around the actual value that also links to the previous value. In this way only one collection has to be maintained and the add, remove and lookup operations are $O(1)$. The cost is that iteration now works on a linked list rather than an array.

5.5. Hybrid Stack and Parse Forest Data Structures

Both the stack and parse forest consist of nodes and connections between nodes. The stack is a graph, the parse forest a tree. The naive implementation represents nodes with classes containing a field with a list of links to other nodes. In practice the number of links to other nodes can often be one. In such cases the link could be embedded as a field directly in the class rather than as element in a list. To still support nodes with multiple links JSGLR2 contains a hybrid parse forest and stack node implementation. These implementations contain a field for a link and an initially uninitialized lists with other links. The list is only instantiated when the second link for a node is added.

6
Results

Previous chapters described the modular architecture JSGLR2 implements and the implemented variants of its components intended to improve performance. This chapter describes the benchmarking approach for systematically measuring the performance of the (combinations of) improved components and the results.

6.1. Setup

The implemented benchmarks for a given test set typically work by performing the following steps. First, the parse table is loaded in the setup phase. Second, a variant of the parser is instantiated. Third, a set of input programs is loaded. Then, in the actual benchmark, after several warmup rounds all inputs are parsed multiple times and the execution time is measured. These steps are then repeated for several variants of the parser and optionally for a range of input sizes. Benchmarks are executed both on variants of JSGLR2 and the original JSGLR implementation.

6.1.1. Java Benchmarking with JMH

OpenJDK's Java Microbenchmark Harness (JMH[1]) is the framework used to perform benchmarks. It includes separation of setups of benchmarks which execution times are excluded from the benchmarks and it includes prepending warmup iterations before executing benchmarks to make sure time spent on compiler optimizations are not influencing the benchmarks. The benchmarks are executed on a computer with an Intel Core i7 processor with base frequency of 2.3 GHz and 8 GB RAM. Its operating system is Apple's macOS version 10.13.2 running Oracle's JVM build 1.8.0_45-b14. While performing benchmarks, other software running on the machine is reduced as much as possible.

6.1.2. Test Sets

The following artificial grammars are constructed for analysis on their specific determinism characteristics:

Ambiguous sums A grammar accepting sums of x's with increasing input sizes. It basically consists of two productions: `Exp.Add = «Exp> + <Exp»` and `Exp.X = <x>`. No associativity is defined, so parses of more than two x's will be ambiguous.

Non-Ambiguous sums Similar as the previous grammar, but then with left associativity defined. This makes the grammar non-ambiguous.

Lexical strings A simple grammar consisting of a single production `Exp = [a-z]+`. It accepts inputs of single strings of lowercase letters without whitespace and with increasing size. No context-free productions are present.

The ambiguous sums test set is expected to be mostly non-deterministic since no associativity is defined on the production reducing additions. In contrast, the non-ambiguous sums and lexical strings test sets are expected to be completely deterministic. Additionally, the following programming languages are considered for getting insight on more practical performance:

[1] http://openjdk.java.net/projects/code-tools/jmh/

Java 8 Java 8 grammar with as input set the complete Java 8 standard library (7715 files, ±86MB).

GreenMarl GreenMarl is a domain specific language for efficient graph analysis[2]. Due to a lack of source code available, the input for this test set is a single program (27 KB).

WebDSL A domain specific language for web applications[3]. Several projects implemented with WebDSL are used as input: WebLab[4], Researchr[5] and YellowGrass[6] (268 files, ±2,1 MB)).

The following sections present benchmarking results for the different test sets per improved variant. JSGLR1's performance is added to the plots for reference.

6.2. Parse Table Representation

Figure 6.1 and Figure 6.2 show the benchmarking results for the test sets for the various implementations of the parse table regarding the mappings from characters and productions to shift, reduce and accept actions and goto actions, respectively. For all other components the naive variants are used. The improvement in parse time is bigger for the programming languages than for the artificial test sets. While both improvements have a positive impact in all cases, the impact of the improved lookup of goto actions is larger than that of the improved lookup of applicable actions.

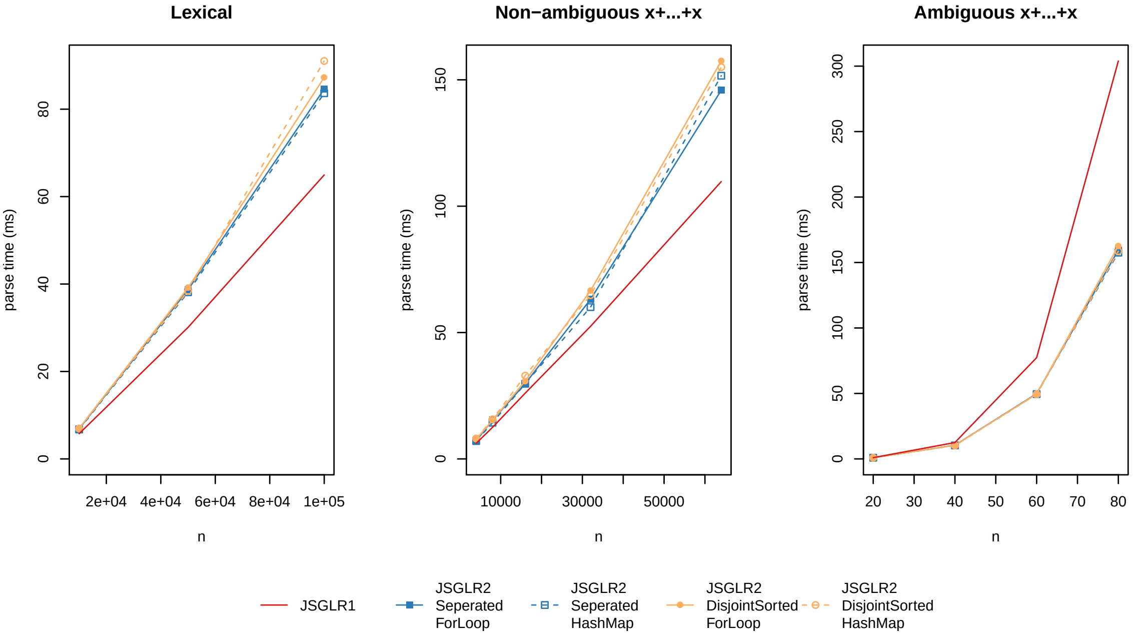


Figure 6.1: Benchmarking results with the artificial test sets for the variants with improved data structures in the parse tables representing the mapping from characters and productions to actions and gotos, respectively, compared to their naive counterparts and JSGLR1.

6.3. Elkhound

In Figure 6.3 we see the benchmarking results for the artificial test sets for Elkhound using the hybrid stack and parse forest. Besides the variant of JSGLR with Elkhound applied, the variant that does use the Elkhound stack but without actually performing the Elkhound LR reductions is added. This allows us to get insight in the overhead of maintaining deterministic depths in the stack.

Since Elkhound is applicable on deterministic parts of the input, we expect a benefit especially on the lexical and non-ambiguous sums test sets. The plots indeed confirm this expectation: for both test sets a

[2] https://github.com/stanford-ppl/Green-Marl
[3] http://webdsl.org
[4] https://weblab.tudelft.nl
[5] https://researchr.org/
[6] https://github.com/webdsl/yellowgrass

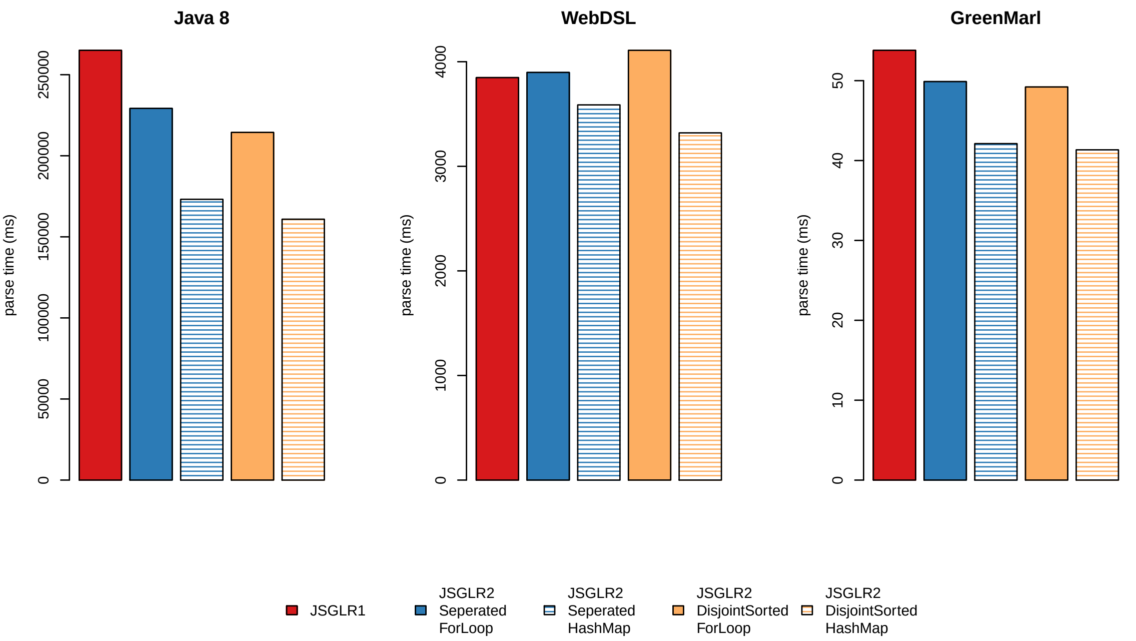


Figure 6.2: Benchmarking results with the programming languages test sets for the variants with improved data structures in the parse tables representing the mapping from characters and productions to actions and gotos, respectively, compared to their naive counterparts and JSGLR1.

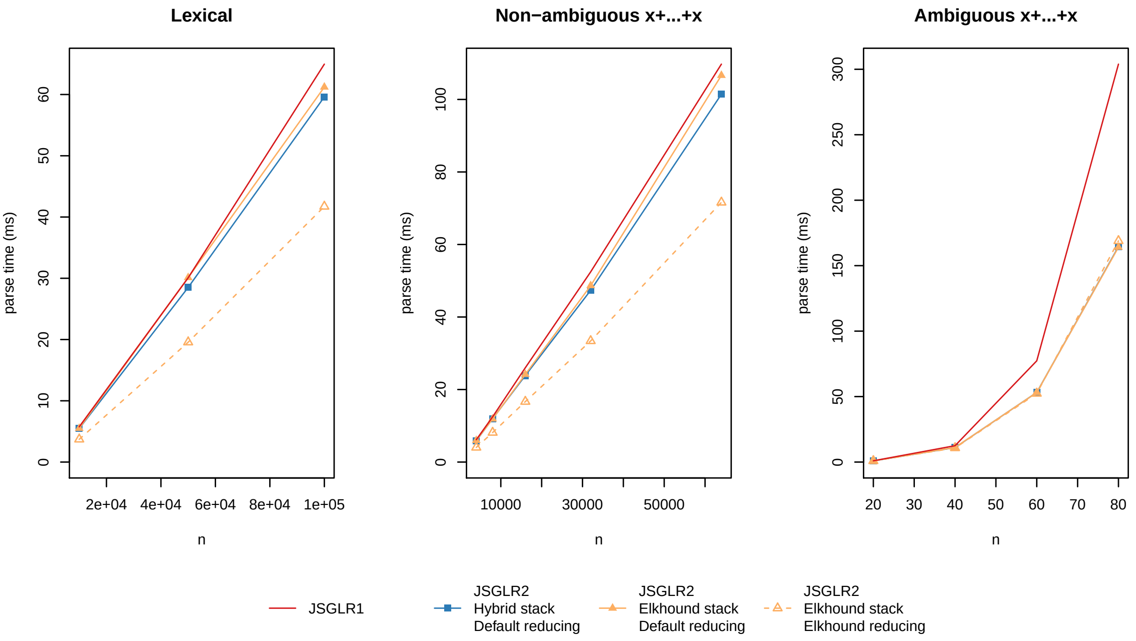


Figure 6.3: Benchmarking results with the artificial test sets for the variants with only the Elkhound stack, the Elkhound stack combined with Elkhound reducing compared to the variant with hybrid data structures and JSGLR1.

decrease of approximately 30% in parse time is found compared to the variant without Elkhound. Due to the non-deterministic characteristics of the ambiguous sums test set applying Elkhound has no benefit there.

Figure 6.3 shows the impact of Elkhound on the programming languages test sets. For all three test sets the overhead of maintaining deterministic depths on the stack is small. Also, applying Elkhound reducing positively effects all parse times. The improvement is bigger for WebDSL and GreenMarl, which indicates that their syntax and input is more deterministic than that of Java.

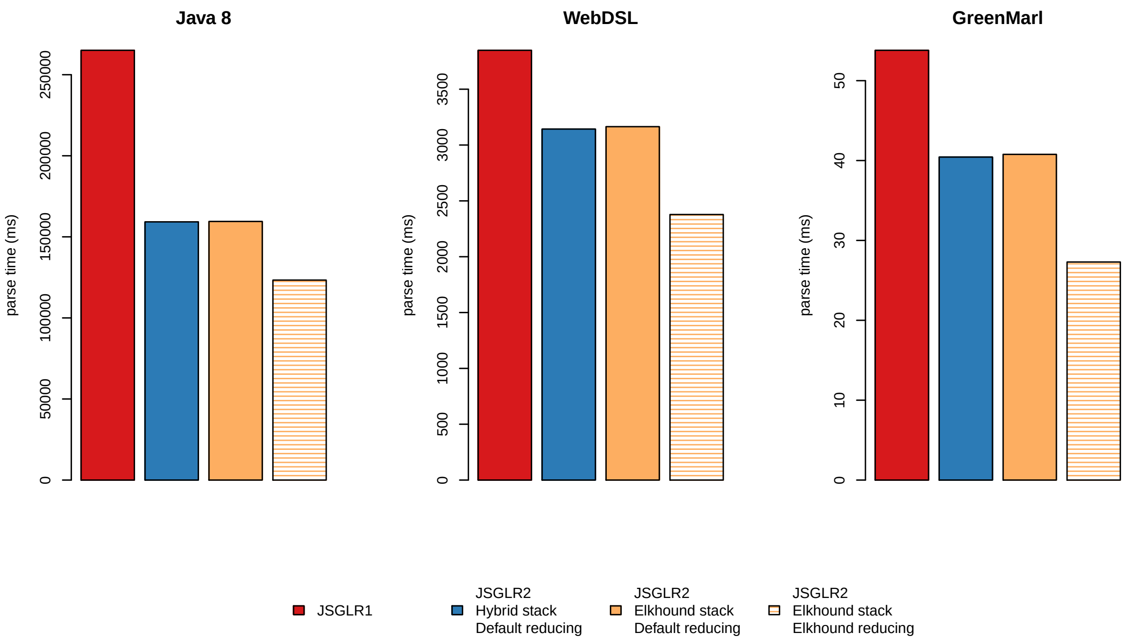


Figure 6.4: Benchmarking results with the programming language test sets for the variants with only the Elkhound stack, the Elkhound stack combined with Elkhound reducing compared to the variant with hybrid data structures and JSGLR1.

6.4. Parse Forest Reduction

The improved variant for parse forest creation prevents the instantiation of parse node objects for lexical, layout and rejected sub trees and just applies recognizing without parse tree construction in such subsets of the input. Figure 6.5 indicates the number of parse nodes for the artificial test sets with and without the optimization applied.

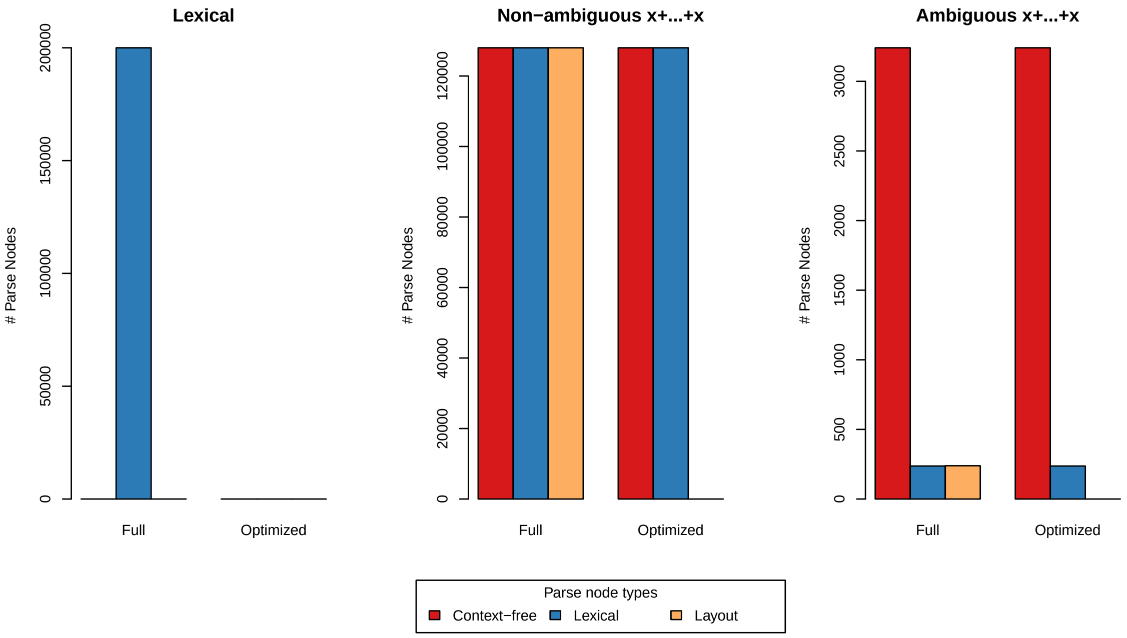


Figure 6.5: Number of parse nodes per type for the full and optimized parse forest for the artificial test sets.

Figure 6.6 shows the benchmarking results on the artificial test sets with the parse forest reduction applied compared to the variant of JSGLR2 with hybrid data structures. The mostly deterministic tests (lexical strings and non-ambiguous sums) parse with bigger throughput than the ambiguous sums test set, which also indicates the time spent on parse forest creation is relatively larger. The benchmarking results reflect this: the reduction of parse forests has a clear benefit on the mostly deterministic test sets but the benefit is only minor for the ambiguous sums test set.

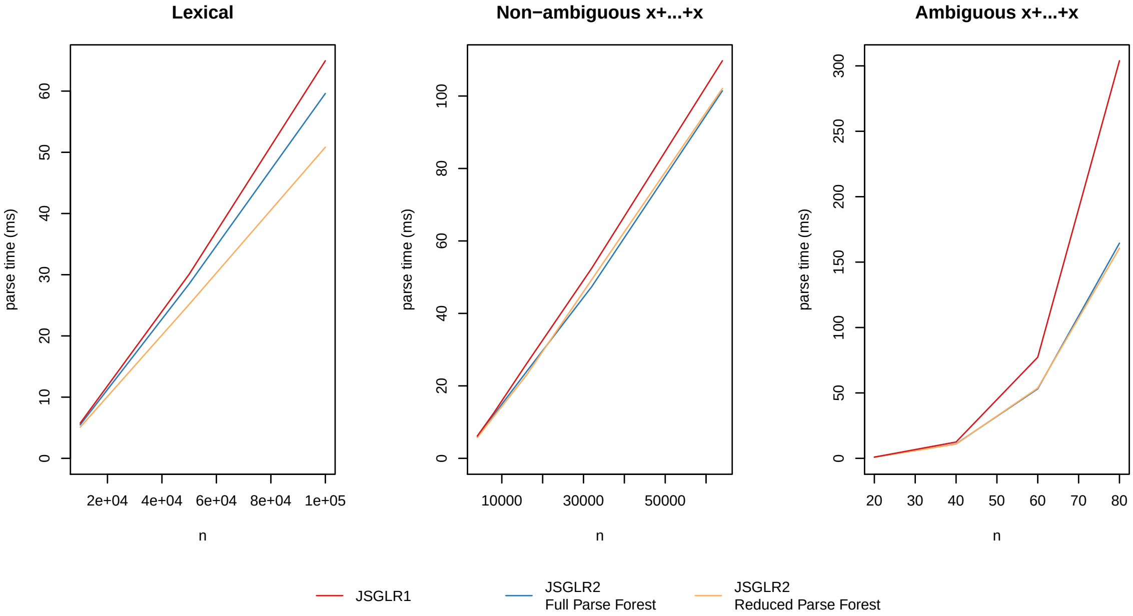


Figure 6.6: Benchmarking results with the artificial test sets for the variants with the hybrid data structures and with/without parse forest construction optimization.

Figure 6.7 shows the distribution of the number of parse nodes for the programming languages test sets with and without the optimized parse forest. In Figure 6.8 the effect on parse time for the reduction of the parse table is visible for the programming languages test sets. Since time spent during parsing on parse forest creation is relatively small, the improvements of parse times are also relatively small.

6.5. Stack Collections

For the two stack collection components (`forActorStacks` and `activeStacks` several data structures are benchmarked. For both collections, an array based and linked hashmap collection variant is present. The `activeStacks` collection additionally has a variant which maintains both an array and hashmap. The benchmarking results are shown in Figures 6.9 and 6.10.

For all improved variants the overhead of maintaining the more expensive data structures is not compensated by a bigger improvements on its operations. For both collections the simple array based implementations performs best.

6.6. Hybrid Data Structures

The hybrid data structures prevent list instantiations on stack nodes with only a single link or parse forest nodes with only one derivation. Stack nodes only have one link in deterministic configurations and parse forest nodes have only one derivation in non-ambiguous parse forests. From the artificial test sets we expect the non-ambiguous sums and lexical strings to have such characteristics. This is confirmed by measurements performed during parsing (Appendix B): while parsing lexical strings and the non-ambiguous sums all stack nodes and parse nodes have a single link or derivation. Figure 6.11 shows benchmarking results for the artificial test sets. As expected, we see an improvement of parse times in the plots by the hybrid counterparts for both of the data structures. Parsing inputs in the ambiguous sums test set returns ambiguous parse forests and the hybrid approach will not have a positive impact since lists are still instantiated for the ambiguous parse nodes. We thus expect no improvement in parse times by both improved variants of the data structures. The plot indeed confirms this: the naive and hybrid variants are close and there even is a

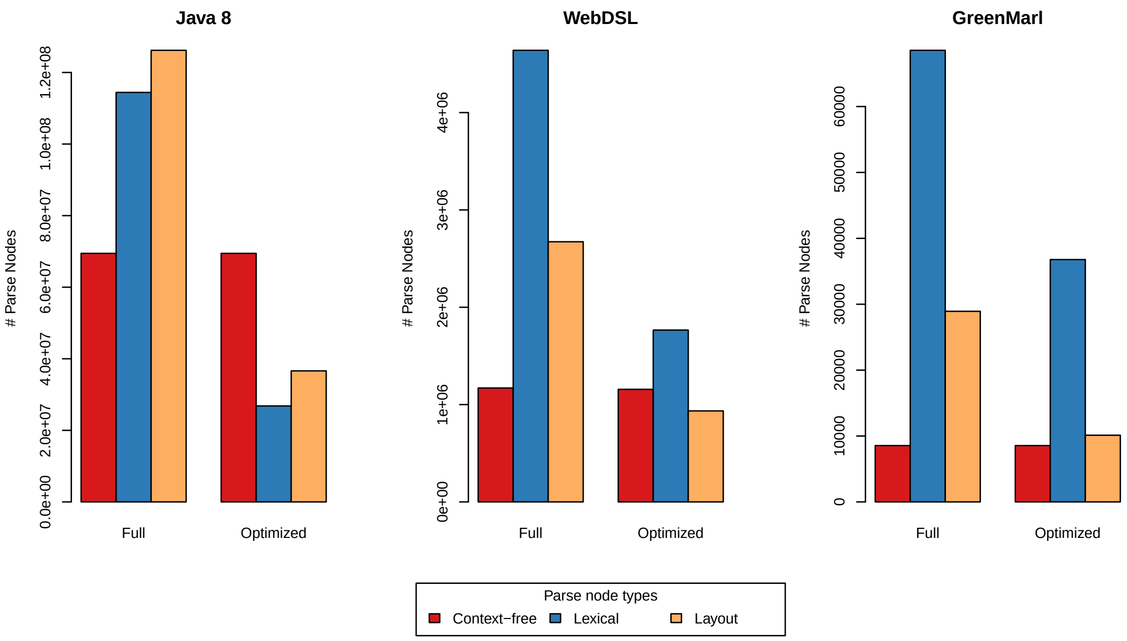


Figure 6.7: Number of parse nodes per type for the full and optimized parse forest for the programming language test sets.

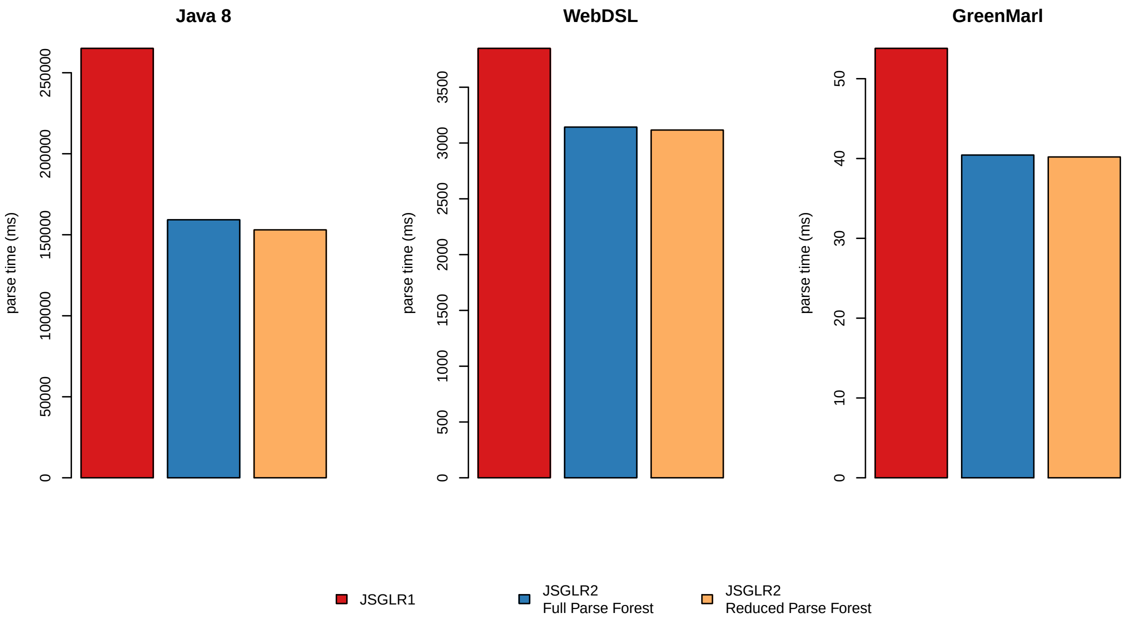


Figure 6.8: Benchmarking results with the programming language test sets for the variants with the hybrid data structures and with-/without parse forest construction optimization.

small overhead for the hybrid variants. This overhead is due to the branching they perform on the single and multiple links/derivations scenarios when their child nodes or derivations are requested. The overhead for the hybrid stack with respect to the hybrid parse forest seems bigger, which is expected to be due to the ratio of stack nodes having a single link that is bigger than that of parse nodes having a single derivation (67% vs. 17%, respectively).

Figure 6.12 shows the effect of the hybrid data structures for the programming languages test sets. On these test sets only a minor improvement by the hybrid parse forest is noticeable. For WebDSL and GreenMarl, the hybrid stack even seems to have negative effect. This is expected to be due to the same reason as for

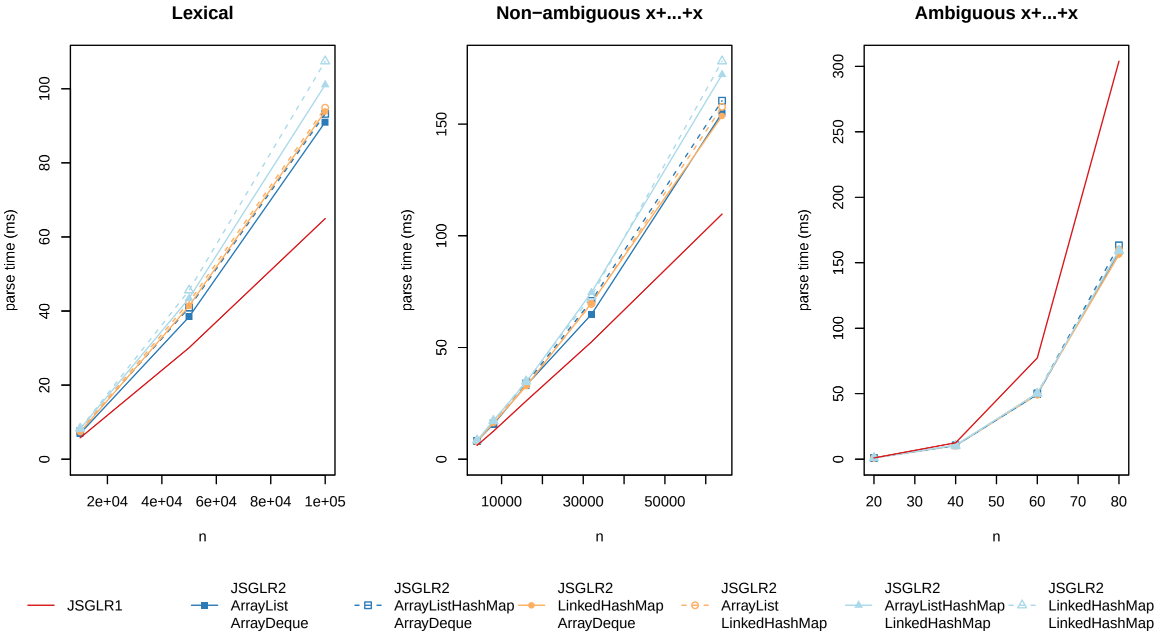


Figure 6.9: Benchmarking results for the artificial test sets for the variants with alternative stack collections implementations.

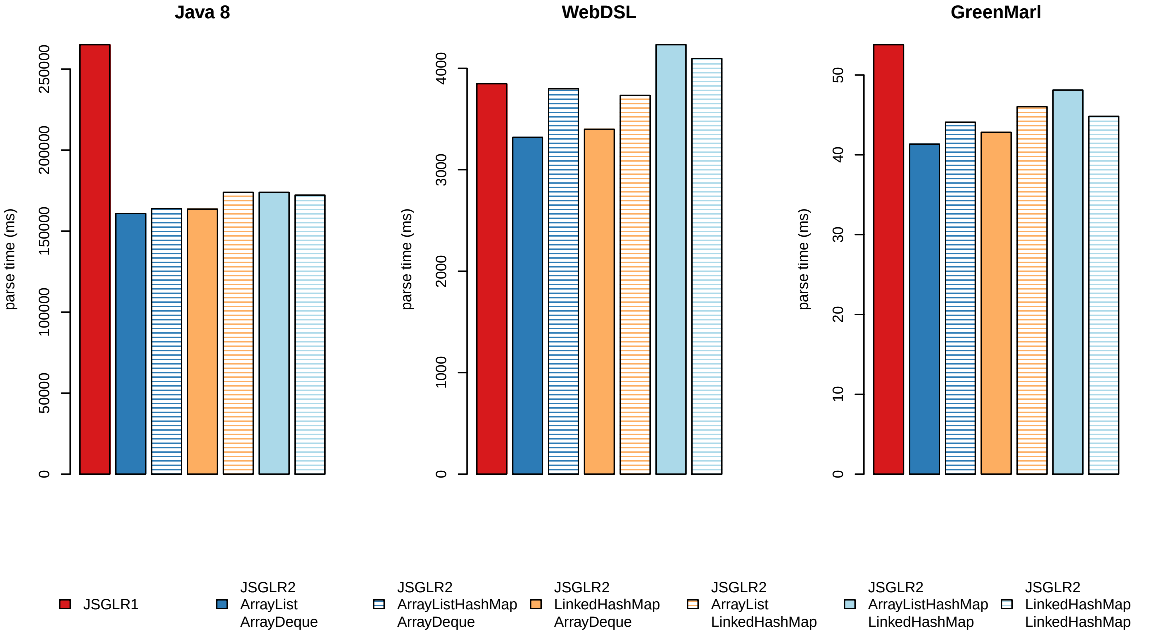


Figure 6.10: Benchmarking results for the programming languages test sets for the variants with alternative stack collections implementations.

the ambiguous sums test set: the overhead of branching between the cases of single and multiple links does not weigh up against the prevention of list instantiations.

6.7. Best Variants

At last the improved variants are combined and benchmarked to compare their impact with respect to the naive implementations. For the artificial and programming language test sets, Figure 6.13 and 6.14 show the benchmarking results, respectively. From the artificial test sets is clear that the combinations of improve-

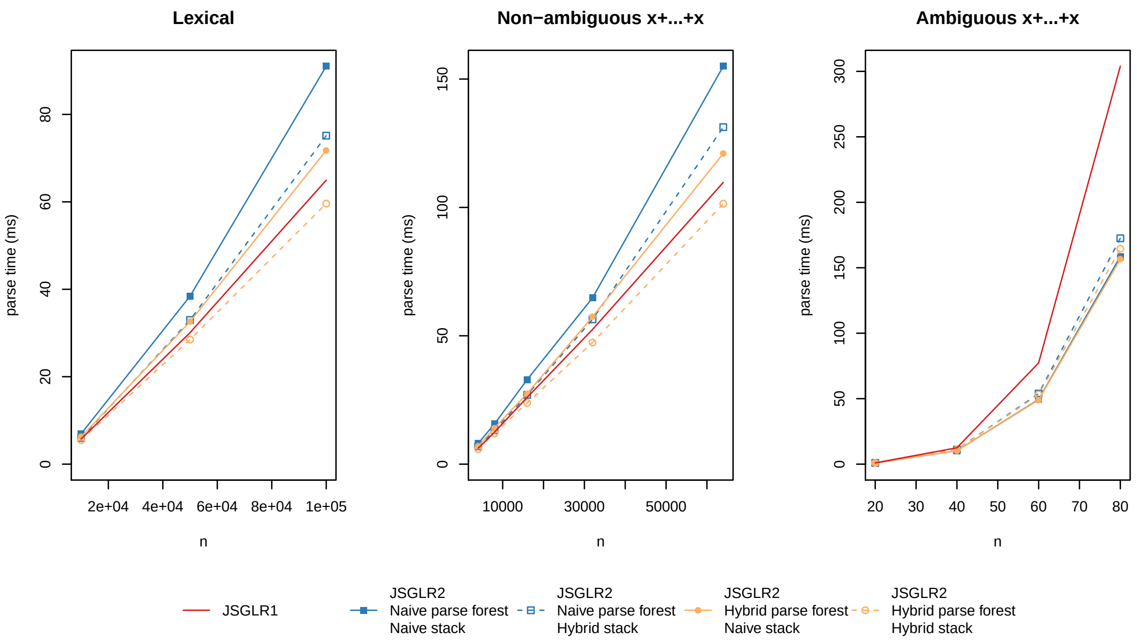


Figure 6.11: Benchmarking results with the artificial test sets for the variants with hybrid stacks and parse forests compared to their naive counterparts.

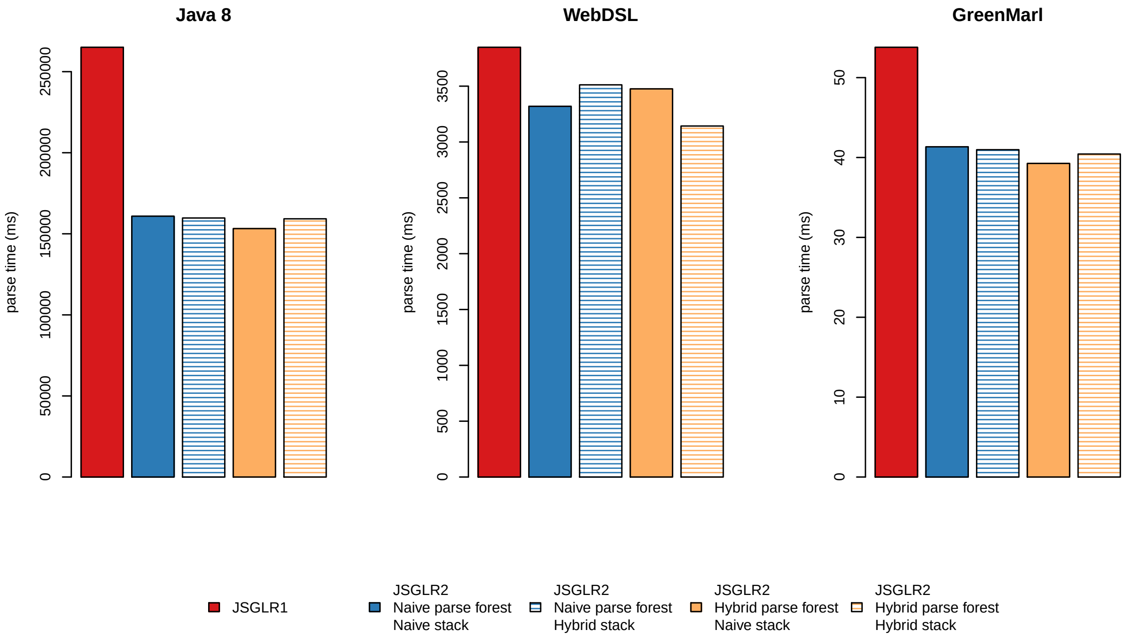


Figure 6.12: Benchmarking results with the programming language test sets for the variants with hybrid stacks and parse forests compared to their naive counterparts.

ments have positive impact on the deterministic test sets, with Elkhound applied reaching a speedup of more than $2x$. For the mostly non-deterministic ambiguous sums test set the improved variants do not lead to significant reduction in parse times compared to the naive implementations. For the programming language test sets the impact is significant, too. For every test set the improved variants perform better than JSGLR1.

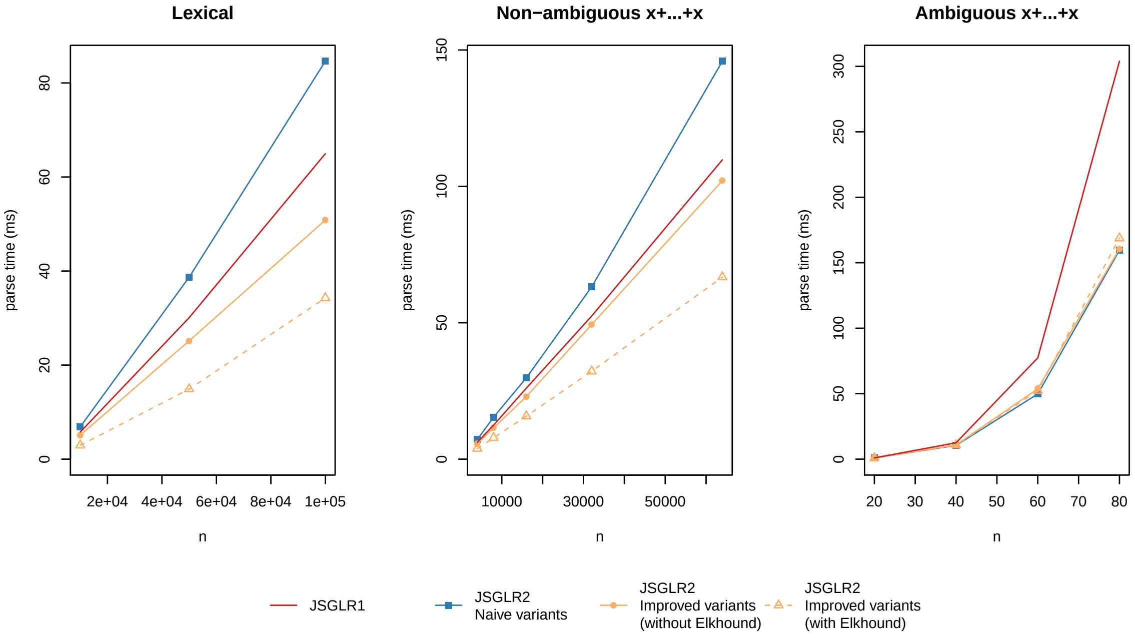


Figure 6.13: Benchmarking results with the artificial test sets for the naive and the best combined variants (with and without Elkhound).

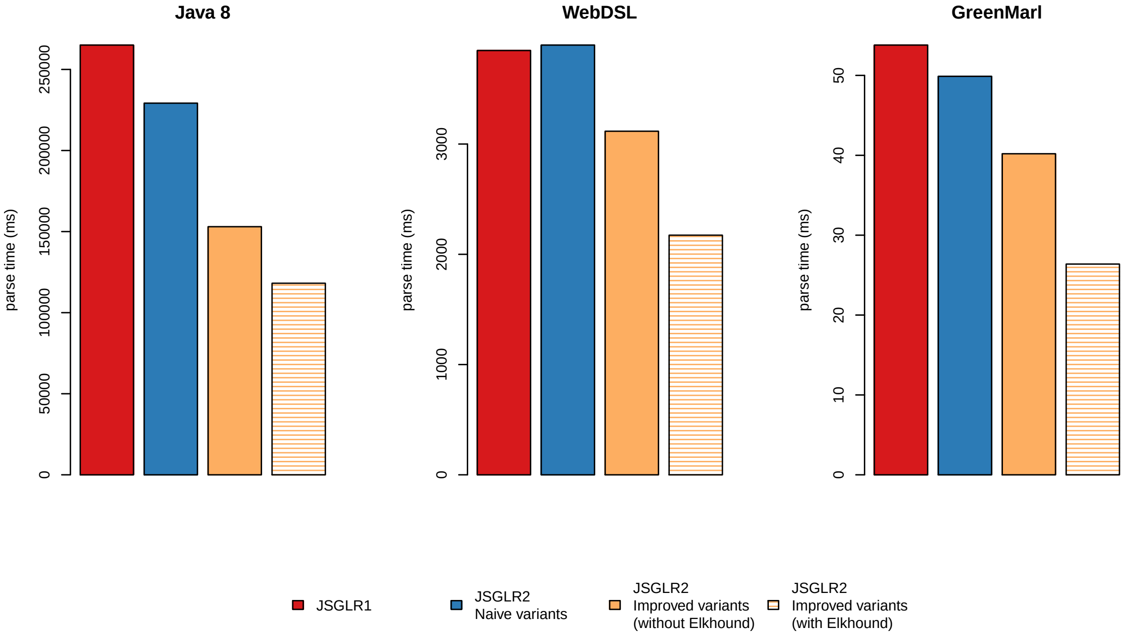


Figure 6.14: Benchmarking results with the programming language test sets for the naive and the best combined variants (with and without Elkhound).

6.8. Comparing Languages

The previous sections reported results using parse times for the various test sets. To compare parsing performance for various languages against each other, we need a different measure since comparison of test sets of varying sizes would be unfair. Figure 6.15 makes a fair comparison by showing benchmark results as throughput. Throughput means the amount of input parsed by time and is a method of normalizing results that enables comparing languages with each other. From the results is clear that there are differences between languages. Parsing GreenMarl has the highest throughput, Java the lowest and WebDSL is somewhere in the middle.

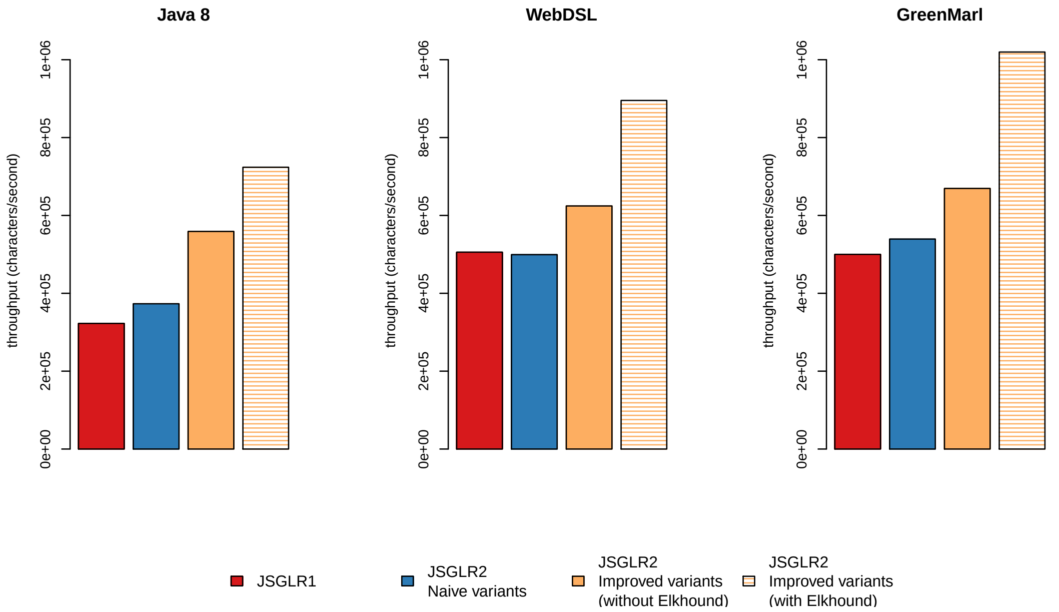


Figure 6.15: Throughput for parsing the programming languages test sets..

6.9. Imploding

The previous sections considered performance of just parsing an input. While the result of such parses are parse trees, in practice we are often interested in an imploded result with an AST as output. Figure 6.16 shows the benchmarking results as throughput for the same test sets as in Figure 6.15, but then with imploding. These results indicate imploding for JSGLR2 only has a small overhead. JSGLR1 has the biggest slowdown with imploding in case of GreenMarl. Findings from manual investigating that this is due to inefficient lexical imploding (where each character node is visited while this is not required) are confirmed by this results combined with the relatively large number of lexical parse nodes in the parse forest (Figure 6.7). When imploding is considered, the speedup of JSGLR2 compared to JSGLR1 is more than $3x$ for Java and more than $10x$ for GreenMarl.

6.10. Threats to Validity

Several threats to validity exist regarding the chosen method for performance evaluation, like correctness and representativeness. Regarding correctness, the benchmarks are implemented to only allow syntactically valid input programs. If an input is not valid, the complete benchmark fails. Together with an extensive set of unit tests for JSGLR2 covering SDF3 features most threats to correctness are expected to be countered. Regarding representativeness, a remaining threat to validity is the size of the test sets chosen for GreenMarl, which is small. For other, bigger, test sets the results might be different. For Java and WebDSL this threat is smaller, since the test sets are relatively large. However, it remains arguable to draw conclusions based on this test set for the languages Java and WebDSL in general.

6.11. Modularity of Architecture

Table 6.1 gives an overview of the presented improvements, the components they affect and the required changes in addition of files and lines of code. The changes are measured by only looking at the code that is used to implement the actual improvements itself. A small amount of additional code is needed per variant, e.g. add an extra branch for the new variant in a switch statement of the factory where a component is instantiated, but these are considered as part of the modular architecture and thus excluded from the numbers. The numbers validate the modular architecture in the sense that they are small with respect to the total code base, which indicates it requires relatively little effort to change or extend a component. Additionally, with all

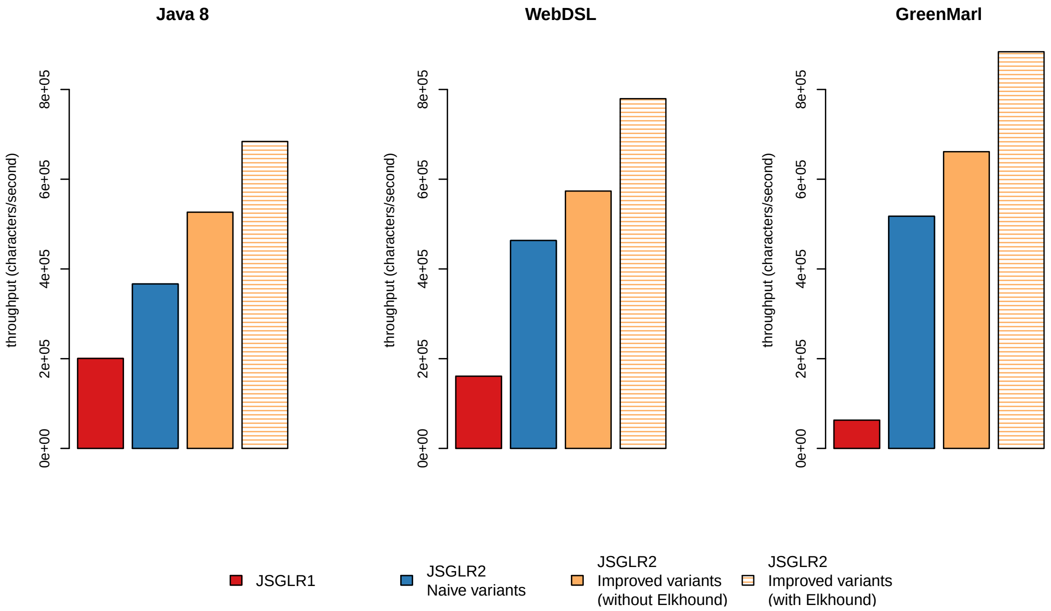


Figure 6.16: Throughput for parsing and imploding the programming languages test sets.

possible combinations of variants of the components a working parser can be composed. Elkhound requires the biggest changes: it involves variations of four components leading to 589 lines of code added spread over 9 files. Still, it can be used independently of e.g. the parse forest implementation and parse forest construction method. The modular architecture as introduced by this work is focused on supporting the variation points required for the investigated improvements. However, there are variation points that would not be directly supported by the architecture. An example is supporting parsing of Unicode inputs, which involves characters with an encoding of more than 8 bits. In contrast to supporting the variation points as discussed, supporting different types of character encodings in current JSGLR2 requires the change of core parser code.

Improved Variant	Affected Components	Δ Files	Δ LoC
Retrieval of shift/reduce/accept actions: binary search on disjoint sorted character ranges	IActionsForCharacter	1	142
Retrieval of goto actions: hashmap	IProductionToGoto	1	35
activeStacks collection (ArrayList+HashMap and LinkedHashMap)	IActiveStacks	2	169
forActor collection (LinkedHashMap)	IForActorStacks	1	63
Hybrid stack	Stack, StackManager	2	75
Hybrid parse forest	ParseForest, ParseForestManager	5	247
Elkhound reducing	Parser, Stack, StackManager, ReduceManager	9	589
Parse forest reduction	Reducer	1	92

Table 6.1: An overview of JSGLR2's improvements, the components they affect and the added files and lines of code (LoC) they require.

7
Conclusions

This work introduces a modular SGLR parsing architecture, several implementations for its components and a systematic benchmark approach to evaluate its performance. The implementation in Java is called JSGLR2 and is integrated in and released with Spoofax, succeeding its original parser implementation JSGLR. Also, it is backwards compatibility with its predecessor and fully supports parse tables generated by SDF3.

The modular architecture is validated in several ways. First, since the parser is observable, various extensions were implemented without having to change core parser code like logging for debugging purposes, performing measurements during parsing and making visualizations. Second, the components of the parser can be easily replaced by improved variants. With little extra code added, the following variation points were introduced by implementing variants of several components: retrieval of actions on states, stack collections representation, stack representation, parse forest representation, parse forest construction and reducing. Third, the parser is extensible, since e.g. another AST output format can be accomplished by replacing the imploder component with a custom one.

The improved variants included data structure (representation/implementation of parse table, stack and parse forest) and algorithmic (hybrid LR/GLR Elkhound reducing, parse forest construction reduction) improvements. Their performance is evaluated using systematic benchmarking on various test sets. Using artificial test sets with specific properties, it became clear that several improvement mostly have impact on deterministic inputs. For the considered programming languages, the combination of improved variants have an improvement of at least $\pm 2x$ on parse times with respect to their naive counterparts. When imploding is also considered, this speedup increases to up to $10x$ for the case of GreenMarl with respect to JSGLR1, mainly due to inefficient imploding of lexical parse forests in JSGLR1. Being able to benchmark all possible combinations of the implemented variants validates the claim that JSGLR2 has a modular architecture. However, the modular architecture is focused on supporting the variation points as discussed in this work and some extensions would not be directly supported. An example is the support for Unicode input strings, which would require adjusted character class representations and changes to the core parser implementation. Due to the clean code organization, these adjustments are expected to require minimal effort.

7.1. Discussion

Much time was invested to rewrite JSGLR and make JSGLR2 backwards compatible with its predecessor and be integrated in Spoofax. The disadvantage of this approach was that it took a lot of time which reduced the amount of time being spent on the actual performance optimizations. However, the advantage is that the result of this project is a working modular parser implementation integrated in and released with Spoofax which can actually be used and form a basis for future work. Some features are missing, but possibly new research projects can continue with this work. Examples include the implementation or extensions of error recovery or incremental parsing. See also the future work section.

In its current state the lack of error reporting and recovery is a limitation probably too big for interactive usage in Spoofax since users are used to JSGLR with recovery. However in non-interactive environments JSGLR2 is ready to parse syntactically valid programs and its applications can benefit from the improved performance.

The two main contributions of this work are the modular SGLR architecture and performance optimizations. Interestingly, these contributions are conflicting. By making the architecture modular, overhead is introduced. Verifying correctness of a composition of parser components is ensured by splitting them up and using type parameters in Java that should match. The interfaces between all these components ensure the composed parsers are correct or do no compile otherwise. However, for the optimal variants some of these interfaces can be removed. An example are the interfaces that requires Java's lists or iterables as types, where an improved variant actually could use arrays. Since the biggest contribution of this work is the modular architecture, the interfaces are maintained and no time is spent on creating the optimal performing parser with this overhead removed.

8
Future Work

This section describes several suggestions for future work that can build upon SDF3 and JSGLR2:

Update and Improve SDF3 Parse Table Format The current parse table format as produced by SDF3 is outdated and inconvenient. It is outdated since unused older data is present like priorities between productions, which in the past were used to resolve priorities between productions during parse time but are now handled during parser generation and thus are not required to be explicitly in the parse table. The format is inconvenient since loading a parse table requires several post processing steps like deriving rejectable states, finding parse node or tokenization types for productions, etc. Several of these types of information could be generated more precisely during parser generation and adopted in the parse table. This would enable JSGLR2 to be more precise in determining the boundary between context-free and lexical/layout parts of the parse forest and optimize its construction even further. Also, no heuristics are required anymore for productions to determine their type or class of tokens for syntax highlighting. Within Spoofax, the serialization step where the parse table is written to a textual ATerm on a file and then read again by the parser could be skipped. An in-memory representation of the parse table would reduce the overhead of writing disk IO in Spoofax usage.

Add Error Recovery, Syntactic Code Complementions and Incremental Parsing While JSGLR2 is a full implementation of the SGLR algorithm and integrated in Spoofax, it is not fully backwards compatible with its predecessor JSGLR. As discussed in Chapter 3, JSGLR features extra features like error reporting, error recovery and syntactic code completions. Especially in interactive environments like IDEs these features are desired. JSGLR2 could be extended with these features in the same modular way as the current architecture. This would enable experimenting with and evaluation of different types of error recovery using the same testset as already available for JSGLR2. Another extension that would be valuable in the context of interactive environments is incremental parsing in which only subsets of already parsed files are parsed again when changed.

Investigate Different LR Parser Generation Techniques In this work the experiments are carried out with only a single parser generation technique. Different LR parser generation techniques produce parse tables with different properties and thus influence performance. Ideally the parser generation algorithm is modular and an extra variation point for the type of algorithm (LR, SLR, LALR) could be introduced while performing benchmarks. This could lead to interesting new insights, since e.g. typically SLR parse tables have more conflicts and LR parse tables have more states.

Hybrid Scannerless Parsing The scannerless nature of SGLR parsing impacts performance in the sense that parse tables are bigger, there are more productions, more reductions are performed during parsing and naturally parse forests become bigger. An idea to overcome parts of this overhead is to introduce a hybrid scannerless parsing approach. This means that based on the context of the current parsing configuration the parser could switch to a more optimized lexical parser, e.g. using regular expressions. The most obvious approach would probably be improving on scanning layout. Often layout is defined as spaces, tabs and newlines. When the parser encounters layout, the corresponding characters could be parser quickly using regular expressions until the end of that part of layout is reached. Afterwards, the parser continues using its default approach. This hybrid approach is not trivial since the interface between the default and optimized lexical parser should be implemented. After ending the parse of a lexical substring the parser should switch back using the same state as when parsed normally. While the application of this approach would probably be most obvious on layout, in theory it should also be possible on other lexical constructs of the grammar. Since it could reduce the number of reductions that are required to be applied significantly this idea is worth to investigate when considering performance.

Iterative Imploding for Bigger Inputs A limitation of JSGLR1 is the imploding of parse forest. Since it involves a recursive traversal of a tree, this sometimes led to stack overflow problems for big inputs. While such problems did not yet arise for JSGLR2, if they did they could probably be fixed or reduced by introducing an iterative algorithm for imploding. Additionally, it would be interesting to see performance impact by using an iterative approach for imploding compared to recursive. The modular architecture already supports replacing the current imploding technique by different ones.

SRNGLR As Chapter 2 describes, several extensions are known for the original GLR algorithm as introduced by Tomita to make it support the full class of context-free grammars. This work mainly involves investigating the algorithm containing the improvements by Nozohoor-Farshi and Rekers. Additionally, the application of right-nulled parse tables to GLR parsing could be investigated. RNGLR is applied to scannerless parsing by Economopoulos [8] et al. leading to SRNGLR and they found an improvement of 33% in parse times on average compared to SGLR. This work could be applied to JSGLR2 to evaluate whether its performance impact is similar, or possibly different due to the parser or SDF3 parser generator implementation.

Bibliography

[1] Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 163–175. ACM, 2016.

[2] John Aycock, Nigel Horspool, Jan Janoušek, and Bořivoj Melichar. Even faster generalized lr parsing. *Acta Informatica*, 37(9):633–651, 2001.

[3] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *ACM SIGPLAN Notices*, volume 39, pages 365–383. ACM, 2004.

[4] Maartje de Jonge, Lennart CL Kats, Eelco Visser, and Emma Söderberg. Natural and flexible error recovery for generated modular language environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(4):15, 2012.

[5] Luis Eduardo de Souza Amorim, Timothée Haudebourg, and Eelco Visser. Declarative disambiguation of deep priority conflicts. Technical Report TUD-SERG-2017-014, Delft University of Technology, 2017.

[6] Jasper Denkers. Jsglr2, January 2018. URL `https://doi.org/10.5281/zenodo.1154156`.

[7] Franklin L DeRemer. *Practical translators for LR (k) languages*. PhD thesis, MIT Cambridge, Mass., 1969.

[8] Giorgios Economopoulos, Paul Klint, and Jurgen Vinju. Faster scannerless glr parsing. In *International Conference on Compiler Construction*, pages 126–141. Springer, 2009.

[9] Giorgios Robert Economopoulos. *Generalised LR parsing algorithms*. PhD thesis, University of London, 2006.

[10] Jan Heering, Paul Robert Hendrik Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism sdf—reference manual—. *ACM Sigplan Notices*, 24(11):43–75, 1989.

[11] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. The grammar tool box: a case study comparing glr parsing algorithms. *Electronic notes in theoretical computer science*, 110:97–113, 2004.

[12] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. The gtb and pat tools. *Electronic Notes in Theoretical Computer Science*, 110:173–175, 2004.

[13] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Evaluating glr parsing algorithms. *Science of Computer Programming*, 61(3):228–244, 2006.

[14] Karl Trygve Kalleberg and Eelco Visser. Spoofax: An extensible, interactive development environment for program transformation with stratego/xt. Technical report, Delft University of Technology, Software Engineering Research Group, 2007.

[15] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench.Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, 2010.

[16] Lennart CL Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. *Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing*, volume 44. ACM, 2009.

[17] Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *Pure and declarative syntax definition: paradise lost and regained*, volume 45. ACM, 2010.

[18] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20. Milan, Italy, 1994.

[19] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[20] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages and Programming*, pages 255–269. Springer, 1974.

[21] Scott McPeak and George C Necula. Elkhound: A fast, practical glr parser generator. In *International Conference on Compiler Construction*, pages 73–88. Springer, 2004.

[22] MetaBorg. ATerms documentation. `http://www.metaborg.org/en/latest/source/langdev/meta/lang/aterm/terms.html`, 2017.

[23] MetaBorg. JSGLR and JSGLR2 source code repository. `https://github.com/metaborg/jsglr`, 2017.

[24] MetaBorg. SDF3 documentation. `http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/index.html`, 2017.

[25] Leon Moonen. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22. IEEE, 2001.

[26] Rohman Nozohoor-Farshi. Handling of ill-designed grammars in tomita's parsing algorithm. In *International Workshop on Parsing Technologies*, pages 182–192, 1989.

[27] David Pager. A practical general method for constructing lr (k) parsers. *Acta Informatica*, 7(3):249–268, 1977.

[28] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll (*) parsing: the power of dynamic analysis. In *ACM SIGPLAN Notices*, volume 49, pages 579–598. ACM, 2014.

[29] Joan Gerard Rekers. *Parser generation for interactive environments*. PhD thesis, Citeseer, 1992.

[30] Daniel J Salomon and Gordon V Cormack. Scannerless nslr (1) parsing of programming languages. In *ACM SIGPLAN Notices*, volume 24, pages 170–178. ACM, 1989.

[31] Daniel J Salomon and Gordon V Cormack. The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical report, Technical Report 95/06, Department of Computer Science, University of Manitoba, Winnipeg, Canada, 1995.

[32] Elizabeth Scott and Adrian Johnstone. Right nulled glr parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):577–618, 2006.

[33] Elizabeth Scott, Adrian Johnstone, and Shamsa Sadaf Hussain. Tomita-style generalised lr parsers. *Royal Holloway University of London*, 2000.

[34] Masaru Tomita. An efficient context-free parsing algorithm for natural languages. In *IJCAI*, volume 2, pages 756–764, 1985.

[35] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Computational linguistics*, 13 (1-2):31–46, 1987.

[36] Masaru Tomita. Graph-structured stack and natural language parsing. In *Proceedings of the 26th annual meeting on Association for Computational Linguistics*, pages 249–257. Association for Computational Linguistics, 1988.

[37] Mark GJ Van den Brand, Jeroen Scheerder, Jurgen J Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized lr parsers. In *International Conference on Compiler Construction*, pages 143–158. Springer, 2002.

[38] Arie Van Deursen and Tobias Kuipers. Building documentation generators. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 40–49. IEEE, 1999.

[39] Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabrieël Konat. A language designer's workbench: a one-stop-shop for implementation and verification of language designs. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 95–111. ACM, 2014.

[40] Eelco Visser et al. *Syntax definition for language prototyping*. Eelco Visser, 1997.

[41] Eelco Visser et al. *Character classes*. Universiteit van Amsterdam. Programming Research Group, 1997.

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

[43] Tobi Vollebregt, Lennart CL Kats, and Eelco Visser. Declarative specification of template-based textual editors. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, page 8. ACM, 2012.

Appendices

A

Parse Table Measurements

Table A.1 shows the characteristics of the parse for the test sets as considered in this work.

Test set	Lexical	Sums Ambiguous	Sums Non-ambiguous	GreenMarl	WebDSL	Java 8
states	10	15	14	2672	10761	4687
characterClasses	9	15	13	11449	48378	17435
characterClassesUnique	3	5	5	285	727	274
statesDisjointSortableCharacterClasses	10	14	14	1555	5386	3170
gotos	14	27	21	51148	189186	138274
gotosPerStateMax	9	5	5	431	522	771
actionGroups	9	15	13	10906	44647	16656
actionDisjointSortedRanges	12	26	24	22468	94510	32315
actions	0	0	0	0	0	0
actionGroupsPerStateMax	2	2	2	39	37	33
actionDisjointSortedRangesPerStateMax	2	3	3	57	58	51
actionsPerStateMax	4	6	4	494	214	2244
actionsPerGroupMax	1	2	1	8	6	62
actionsPerDisjointSortedRangeMax	1	2	1	8	6	62

An explanation of the parse table measurements below. Note that the parse table format contains state, with per state groups of actions with a character class. These groups of actions can have overlapping character classes.

states The number of states.

characterClasses The number of character classes.

characterClassesUnique The number of unique character classes.

statesDisjointSortableCharacterClasses The number of states that contain groups of actions per character classes which character classes can be sorted disjoint without further processing.

gotos The number of goto actions.

gotosPerStateMax The maximum number of goto actions in a state.

actionGroups The number of groups of actions.

actionsDisjointSortedRanges The number of groups of actions, after making them grouped by disjoint sorted ranges.

actions The number of actions (reduce, shift, accept).

actionGroupsPerStateMax The maximum number of groups of actions in a state.

actionDisjointSortedRangesPerStateMax The maximum number of groups of actions in a state after making them disjoint sorted by character ranges.

actionsPerStateMax The maximum number of actions in a state.

actionsPerGroupMax The maximum number of actions in a group of actions with a character class.

actionsPerDisjointSortedRangeMax The maximum number of actions in a group of actions with a character class after making them disjoint sorted by character ranges.

B

Parsing Measurements

Table B.1 shows the characteristics of the parsing of the test sets as considered in this work with Elkhound. For some test sets benchmarks are carried out with multiple input sizes. The measurements in this appendix are carried out for the biggest evaluated size for these test sets.

Test set	Lexical	Sums Non-ambiguous	Sums Ambiguous	GreenMarl	WebDSL	Java 8
size	100000	64000	80	-1	-1	-1
files	1	1	1	1	268	7715
characters	100000	127999	159	26902	1945813	85525078
activeStacksAdds	300002	511999	796	132322	10103323	383201934
activeStacksMaxSize	1	1	5	24	58	88
activeStacksIsSingleChecks	500003	895998	726	171065	12267582	422336441
activeStacksIsEmptyChecks	500003	895998	726	171065	12267582	422336441
activeStacksFindsWithState	0	0	85947	86430	6789216	327167986
activeStacksForLimitedReductions	0	0	3159	8385	853038	37933723
activeStacksAddAllTo	0	0	0	3370	184803	11748414
activeStacksClears	300002	511999	402	86098	5825577	193846091
activeStacksIterators	0	0	78	0	611	12191
forActorAdds	0	0	394	49141	4123714	198193566
forActorDelayedAdds	0	0	0	453	338835	2910691
forActorMaxSize	0	0	2	8	17	33
forActorDelayedMaxSize	0	0	0	1	5	2
forActorContainsChecks	0	0	9321	35247	3234145	180362253
forActorNonEmptyChecks	200001	383999	793	134658	9732450	350733853
stackNodes	300002	511999	718	128105	9769974	372996332
stackLinks	300001	511998	3954	141401	10871611	467456276
stackLinksRejected	0	0	0	225	17896	2036645

deterministicDepthResets	0	0	78	0	611	12191
parseNodes	200001	383999	3717	106020	8482959	310069099
parseNodesAmbiguous	0	0	0	0	0	0
parseNodesContextFree	1	128000	3241	8552	1170291	69454917
parseNodesContextFreeAmbiguous	0	0	0	0	0	0
parseNodesLexical	200000	127999	237	68532	4639687	114423517
parseNodesLexicalAmbiguous	0	0	0	0	0	0
parseNodesLayout	0	128000	239	28927	2672981	126190665
parseNodesLayoutAmbiguous	0	0	0	0	0	0
characterNodes	100000	127999	160	26903	1946019	85532793
actors	0	0	394	49451	4445279	200321441
doReductions	200001	383999	558	96531	7409752	259685751
doLimitedReductions	0	0	6162	13287	1205377	67333128
doReductionsLR	0	0	79	18235	871165	19291492
doReductionsDeterministicGLR	200001	383999	323	70949	5904189	219404460
doReductionsNonDeterministicGLR	0	0	156	7347	634398	20989799
reducers	0	0	85713	52083	4956178	213997304
reducersElkhound	200001	383999	164	54978	3546147	98107696

An explanation of the parsing measurements below. The measurements are gathered while parsin

size For the artificial test sets, its size. For the test set with sums the size is the number of x's, for the lexical test set it is the length of the string.

files The number of files in the test set.

characters The total number of characters parsed in the test set.

activeStacksAdds The number of times a stack is added to the active stacks collection.

activeStacksMaxSize The maximum size the active stacks collection gets.

activeStacksIsSingleChecks The number of times the active stacks collection is checked for containing a single stak.

activeStacksIsEmptyChecks The number of times the active stacks collection is checked for being empty.

activeStacksFindsWithState The number of times a stack with a given state is looked up in the active stacks collection.

activeStacksForLimitedReductions The number of times the active stacks collection is searched for stacks applicable for DO-LIMITED-REDUCTIONS.

activeStacksAddAllTo The number of times all stacks in the active stacks collection are added to the forActor collection.

activeStacksClears The number of times the active stacks collection is cleared.

activeStacksIterators The number of times the active stacks collection is iterated over.

forActorAdds The number of times a stack is added to the for actor stacks collection.

forActorDelayedAdds The number of times a stack is added to the for actor delayed stacks collection.

forActorMaxSize The maximum size the for actor stacks collection gets.

forActorDelayedMaxSize The maximum size the for actor delayed stacks collection gets.

forActorContainsChecks The number of times the active stacks collection is checked for containing a stack.

forActorNonEmptyChecks The number of times the active stacks collection is checked for being non-empty.

stackNodes The number of stack nodes created.

stackLinks The number of stack links created.

stackLinksRejected The number of stack links rejected.

deterministicDepthResets The number of times stacks are merged and for Elkhound, the deterministic depths of active stacks need to be recomputed.

parseNodes The number of parse nodes created.

parseNodesAmbiguous The number of parse nodes created that are ambiguous.

parseNodesContextFree The number of context-free parse nodes created.

parseNodesContextFreeAmbiguous The number of context-free parse nodes created that are ambiguous.

parseNodesLexical The number of lexical parse nodes created.

parseNodesLexicalAmbiguous The number of lexical parse nodes created that are ambiguous.

parseNodesLayout The number of layout parse nodes created.

parseNodesLayoutAmbiguous The number of layout parse nodes created that are ambiguous.

characterNodes The number of character nodes created.

actors The number of times the ACTOR functions is executed, i.e. an active stack is processed.

doReductions The number of times the DO-REDUCTIONS functions is executed.

doLimitedReductions The number of times the DO-LIMITED-REDUCTIONS functions is executed.

doReductionsLR The number of executions of DO-REDUCTIONS where full LR is applicable.

doReductionsDeterministicGLR The number of executions of DO-REDUCTIONS where full LR is not applicable, but the deterministic depth is large enough.

doReductionsNonDeterministicGLR The number of executions of DO-REDUCTIONS LR is not applicable and the parser falls back to GLR.

reducers The number of times a path is reduced without Elkhound applied.

reducersElkhound The number of times a path is reduced with Elkhound applied.