Faster Scannerless GLR Parsing

Giorgios Economopoulos, Paul Klint, Jurgen Vinju

Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract. Analysis and renovation of large software portfolios requires syntax analysis of multiple, usually embedded, languages and this is beyond the capabilities of many standard parsing techniques. The traditional separation between lexer and parser falls short due to the limitations of tokenization based on regular expressions when handling multiple lexical grammars. In such cases scannerless parsing provides a viable solution. It uses the power of context-free grammars to be able to deal with a wide variety of issues in parsing lexical syntax. However, it comes at the price of less efficiency. The structure of tokens is obtained using a more powerful but more time and memory intensive parsing algorithm. Scannerless grammars are also more non-deterministic than their tokenized counterparts, increasing the burden on the parsing algorithm even further.

In this paper we investigate the application of the Right-Nulled Generalized LR parsing algorithm (RNGLR) to scannerless parsing. We adapt the Scannerless Generalized LR parsing and filtering algorithm (SGLR) to implement the optimizations of RNGLR. We present an updated parsing and filtering algorithm, called SRNGLR, and analyze its performance in comparison to SGLR on ambiguous grammars for the programming languages C, Java, Python, SASL, and C++. Measurements show that SRNGLR is on average 33% faster than SGLR, but is 95% faster on the highly ambiguous SASL grammar. For the mainstream languages C, C++, Java and Python the average speedup is 16%.

1 Introduction

For the precise analysis and transformation of source code we first need to parse the source code and construct a syntax tree. Application areas like reverse engineering, web engineering and model driven engineering specifically deal with many different languages, dialects and embeddings of languages into other languages. We are interested in the construction of parsing technology that can service such diversity; to allow a language engineer to experiment with and efficiently implement parsers for real and complex language constellations.

A parser is a tool, defined for a specific grammar, that constructs a syntactic representation (usually in the form of a parse tree) of an input string and determines if the string is syntactically correct or not. Parsing often includes a scanning phase which first splits the input string into a list of words or tokens. This list is then further analyzed using a more powerful parsing algorithm. This

scanning/parsing dichotomy is not always appropriate, especially when parsing legacy languages or embedded languages. Scanners are often too simplistic to be able to deal with the actual syntax of a language and they prohibit modular implementation of parsers. Scannerless parsing [20, 21, 29] is a technique that avoids such issues that would be introduced by having a separate scanner [7]. Intuitively, a scannerless parser uses the power of context-free grammars instead of regular expressions to tokenize an input string.

The following Fortran statement is a notorious example of scanning issues [1]: `DO 5 I = 1.25`. This statement supposedly has resulted in the crash of the NASA Mariner 1.[1] It is not until the decimal point that it becomes clear that we are dealing here with an assignment to the variable `DO5I`.[2] However, in the slightly different statement: `DO 5 I = 1,25`, `DO` is a keyword and the statement as a whole is a loop construct. This example highlights that tokenization using regular expressions, without a parsing context, can easily be non-deterministic and even ambiguous. In order to restrict the number of possibilities, scanners usually apply several implicit rules like, e.g., *Prefer Longest Match*, *Prefer Keywords*, *Prefer First Applicable Rule*. The downside of such disambiguation is that the scanner commits itself to one choice of tokens and blocks other interpretations of the input by the parser. A scannerless parser with enough lookahead does not have this problem.

Another example is the embedding of Java code in AspectJ definitions and *vice versa*. If a scanner is needed for the combination of the two languages, you may end up with reserving the new AspectJ keywords from the Java code. However, existing Java code may easily contain such identifiers, resulting in parsing errors for code that was initially parsed correctly. One approach that could avoid this problem would be to use two separate scanners: one that is active while parsing pure AspectJ code and another that is active while parsing pure Java code. Once again, the parsing context would be used to decide which scanner is used in the tokenization. This problem does not exist when using a scannerless parser [8].

In a classical scanner/parser approach the scanner makes many decisions regarding tokenization. In a scannerless parser these decisions are postponed and have to be made by the parser. Consequently, scannerless parsers generally have to deal with more non-determinism than before, so the deterministic LR parsing algorithms can no longer be used. However, it turns out that the non-determinism introduced by the removal of the scanner can be gracefully handled by Generalized LR (GLR) parsing algorithms [24, 16, 19].

Scannerless parsing remains a counter-intuitive notion, which is partly due to our education in compiler construction where scanner optimization was a central point of interest. So we emphasize its benefits here once more:

- Computational power: lexical ambiguity is a non-issue and full definition of lexical syntax for real languages is possible.

[1] Various (non-authoritative) sources mention that writing a "`.`" in instead of "`,`" caused the loss of the Mariner 1.

[2] Recall that Fortran treats spaces as insignificant, also inside identifiers.

– Modularity: languages with incompatible lexical syntaxes can be combined seemlessly.
– Scope: to generate parsers for more languages, including ambiguous, embedded and legacy languages.
– Simplicity: no hard-wired communication between scanning and parsing.
– Declarativeness: no side-effects and no implicit lexical disambiguation rules necessary.

So, on the one hand a language engineer can more easily experiment with and implement more complex and more diverse languages using a parser generator that is based on Scannerless GLR parsing. On the other hand there is a cost. Although it does not have a scanning phase, scannerless parsing is a lot more expensive than its two-staged counterpart. The structure of tokens is now retrieved with a more time and memory intensive parsing algorithm. A collection of grammar rules that recognizes one token type, like an identifier could easily have 6 rules, including recursive ones. Parsing one character could therefore involve several GLR stack operations, searching for applicable reductions and executing reductions. Consider an average token length of 8 characters and an average number of stack operations of 4 per character, a scannerless parser would do $4 * 8 = 32$ times more work per token than a parser that reads a pre-tokenized string. Furthermore, a scannerless parser has to consider all whitespace and comment tokens. An average program consists of more than 50% whitespace which again multiplies the work by two, raising the difference between the two methods to a factor of 64. Moreover, scannerless grammars are more non-deterministic than their tokenized counterparts, increasing the burden on the parsing algorithm even more.

Fortunately, it has been shown [7] that scannerless parsers can be implemented fast enough to be applied to real programming languages. In this paper we investigate the implementation of the Scannerless GLR (SGLR) parser provided with SDF [29, 7]. It makes scannerless parsing feasible by rigorously limiting the non-determinism that is introduced by scannerless parsing using disambiguation filters. It is and has been used to parse many different kinds of legacy programming languages and their dialects, experimental domain specific languages and all kinds of embeddings of languages into other languages. The parse trees that SGLR produces are used by a variety of tools including compilers, static checkers, architecture reconstruction tools, source-to-source transformers, refactoring, and editors in IDEs.

As SDF is applied to more and more diverse languages, such as scripting and embedded web scripting languages, and in an increasing number of contexts such as in plugins for the Eclipse IDE, the cost of scannerless parsing has become more of a burden. That is our motivation to investigate algorithmic changes to SGLR that would improve its efficiency. Note that the efficiency of SGLR is defined by the efficiency of the intertwined parsing and filtering algorithms.

We have succeeded in replacing the embedded parsing algorithm in SGLR—based on Farshi's version of GLR [16]—with the faster Right-Nulled GLR algorithm [22, 12]. RNGLR is a recent derivative of Tomita's GLR algorithm that,

intuitively, limits the cost of non-determinism in GLR parsers. We therefore investigated how much the RNGLR algorithm would mitigate the cost of scannerless parsing, which introduces more non-determinism. The previously published results on RNGLR can not be extrapolated directly to SGLR because of (A) the missing scanner, which may change trade-offs between stack traversal and stack construction and (B) the fact that SGLR is not a parsing algorithm *per se*, but rather a parsing and filtering algorithm.The benefit of RNGLR may easily be insignificant compared to the overhead of scannerless parsing and the additional costs of filtering.

In this paper we show that a Scannerless Right-Nulled GLR parser and filter is actually significantly faster on real applications than traditional SGLR. The amalgamated algorithm, called SRNGLR, requires adaptations in parse table generation, parsing and filtering, and post-parse filtering stages of SGLR. In Section 2 we analyze and compare the run-time efficiency of SGLR and the new SRNGLR algorithm. In Sections 3 and 4 we explain what the differences between SGLR and SRNGLR are. We conclude the paper with a discussion in Section 6.

2 Benchmarking SRNGLR

In Sections 3 and 4 we will delve into the technical details of our parsing algorithms. Before doing so, we first present our experimental results. We have compared the SGLR and SRNGLR algorithms using grammars for an extended version of ANSI-C—dubbed C'—, C++, Java, Python, SASL and Γ_1—a small grammar that triggers interesting behaviour in both algorithms. Table 1 describes the grammars and input strings used. Table 2 provides statistics on the sizes of the grammars. We conducted the experiments on a 2.13GHz Intel Dual Core with 2GB of memory, running Linux 2.6.20.

SGLR and SRNGLR are comprised of three different stages: parse table generation, parsing and post-parse filtering. We focus on the efficiency of the latter two, since parse table generation is a one-time cost. We are not interested in the runtime of recognition without tree construction. Note that between the two algorithms the parsing as well as the filtering changes and that these influence each other. Filters may prevent the need to parse more and changes in the parsing algorithm may change the order and shape of the (intermediate) parse forests that are filtered. Efficiency measurements are also heavily influenced by the shapes of the grammars used as we will see later on.

The SRNGLR version of the parser was tested first to output the same parse forests that SGLR does, modulo order of trees in ambiguity clusters.

Table 3 and Figure 1 show the arithmetic mean time of five runs and Table 4 provides statistics on the amount of work that is done. GLR parsers use a Graph Structured Stack (GSS). The edges of this graph are visited to find reductions and new nodes and edges are created when parts of the graph can be reduced or the next input character can be shifted. Each reduction also leads to the construction of a new parse tree node and sometimes a new ambiguity cluster. An ambiguity cluster encapsulates different ambiguous trees for the same substring.

Name	Grammar description	Input size (chars/lines)	Input description
C'	ANSI-C plus ambiguous exception handling extension	32M/1M	Code for an embedded system
C++	Approaches ISO standard, with GNU extensions	2.6M/111K	Small class that includes much of the STL
Java	Grammar from [8] that implements Java 5.0	0.5M/18k	Implementation of The Meta-Environment [5]
Python	Derived from the reference manual [28], ambiguous due to missing off-side rule implementation	7k/201	`spawn.py` from Python distribution
SASL	Taken from [26], ambiguous due to missing off-side rule implementation	2.5k+/114+	Standard prelude, concatenated to increasing sizes
Γ_1	$S ::= SSS \mid SS \mid a$; triggers worst-case behavior [12]	1–50/1	Strings of a's of increasing length

Table 1. Grammars and input strings used.

	NNT	NP	RNP	States	Shifts+Gotos	Reductions		LA Reductions	
						SGLR	SRNGLR	SGLR	SRNGLR
C'	71	93	94	182k	37k	18k	23k	5.9k	6.3k
C++	90	112	102	112k	18k	19k	19k	1.5k	1.5k
Java	81	112	116	67k	9.7k	5.9k	6.0k	1.0k	1.1k
Python	56	74	85	22k	3.4k	1.7k	1.9k	0	0
SASL	16	21	22	4.5k	0.9k	0.5k	0.6k	0	0
Γ_1	0	0	0	13	30	13	15	0	0

Table 2. Grammar statistics showing nullable non-terminals (NNT), nullable productions (NP), right-nullable productions (RNP), SLR(1) states, shifts and gotos, reductions and reductions with dynamic lookahead restriction (LA Reductions).

For both algorithms we count the number of GSS edge visits, GSS node creations, edge and node visits for garbage collection, and parse tree node and ambiguity cluster visits for post-parse filtering. Note that garbage collection of the GSS is an important factor in the memory and run-time efficiency of GLR.

For this benchmark, SRNGLR is on average 33% faster than SGLR with a smallest speedup of 9.8% for C and a largest speedup of 95% for SASL. Apparently the speedup is highly dependent on the specific grammar. If we disregard SASL the improvement is still 20% on average and if we also disregard Γ_1^{50} the average drops to a still respectable 16% improvement for the mainstream languages C, C++, Java and Python. The results show that SRNGLR parsing speed is higher (up to 95%) for grammars that are highly ambiguous such as SASL. SRNGLR also performs better on less ambiguous grammars such as Java (14% faster). The *parsing time* is always faster, and in most cases the *filtering time* is also slightly faster for SRNGLR but not significantly so.

The edge visit statistics (Table 4 and Figure 3) explain the cause of the improved parsing time. Especially for ambiguous grammars the SGLR algorithm traverses many more GSS edges. According to the time measurements this is significant for real world applications of scannerless parsing.

	C'		C++		Java		Python		$SASL^{80}$		$\Gamma_1{}^{50}$	
	S	SRN	S	SRN	S	SRN	S	SRN	S	SRN	S	SRN
Speed (chars/sec.)	385k	443k	121k	175k	404k	467k	178	904	78	1k	4.7	24
Parse time (sec.)	84.2	73.2	21.5	14.9	2.1	1.8	39.2	7.7	4.8k	202.2	10.8	2.1
Filter time (sec.)	102.9	95.5	5.7	5.6	0.8	0.7	327.3	298.8	1.6	1.6	7.7	9.5
Total time (sec.)	187.2	168.8	27.3	20.6	2.9	2.5	366.5	306.5	4.8k	203.9	18.5	11.6
Speedup (%)		9.8		24.5		13.8		16.4		95		37.6

Table 3. Speed (characters/second), Parse time (seconds) , Filter time (seconds), Total time (seconds) and Speedup (%) of SGLR (S) and SRNGLR (SRN). $k = 10^3$.

	C'		C++		Java		Python		$SASL^{80}$		Γ_1^{50}	
	S	SRN	S	SRN	S	SRN	S	SRN	S	SRN	S	SRN
ET	149M	44M	26M	6.6M	3.2M	0.9M	90M	3.4M	71B	165M	48M	0.7M
ES	81M	18M	145M	27M	5.0M	0.9M	1.8B	234M	16B	14B	28M	14M
NC	141M	143M	19M	20M	3.0M	3.0M	157k	157k	2.4M	2.4M	252	252
EC	154M	157M	30M	31M	3.5M	3.4M	962k	962k	44M	44M	3.9k	3.9k
GC	13M	13M	6.2M	6.8M	0.7M	0.6M	2.0M	2.0M	88M	88B	14k	14k
FAC	30k	30k	5.6k	5.6k	0	0	83k	83k	48k	48k	1.2k	2.1k
FNC	241M	241M	13M	13M	1.6M	1.6M	707M	707M	3.1M	3.1M	1.1M	1.3M

Table 4. Workload data. Edges traversed searching reductions (ET), edges traversed searching existing edge (ES), GSS nodes created (NC), GSS edges created (EC), edges traversed for garbage collection (GC), ambiguity nodes created while filtering (FAC), and parse tree nodes created while filtering (FNC). $k = 10^3$, $M = 10^6, B = 10^9$

Filtering time is improved in all but the Γ_1 case, although the improvement is not greater than 10%. The workload statistics show that about the same number of nodes are created during filtering. The differences are lost in the rounding of the numbers, except for the Γ_1 case which shows significantly more node creation at filtering time. This difference is caused by different amounts of sharing of ambiguity clusters between the two versions. The amount of sharing in ambiguity clusters during parsing, for both versions, depends on the arbitrary ordering of reduction steps. I.e. it is not relevant for our analysis.

Notice that the parse time versus filtering time ratio can be quite different between languages. This highly depends on the shape of the grammar. LR factored grammars have higher filtering times due to the many additional parse tree nodes for chain rules. The Python grammar is an example of such a grammar, while SASL was not factored and has a minimum number of non-terminals for its expression sub-language. Shorter grammars with less non-terminals have better filtering speed. We expect that by "unfactoring" the Python grammar a lot of speed may be gained.

Figure 2 depicts how SRNGLR improves parsing speed as the input length grows. For Γ_1 it is obvious that the gain is higher when the input gets larger. Note that although Γ_1 does not have any right-nullable productions (see Table 2) there is still a significant gain. The reason for this is that SRNGLR prevents work from being done for all grammars (see Section 3).

From these results we may conclude that SRNGLR clearly introduces a structural improvement that increases the applicability of scannerless GLR parsing to

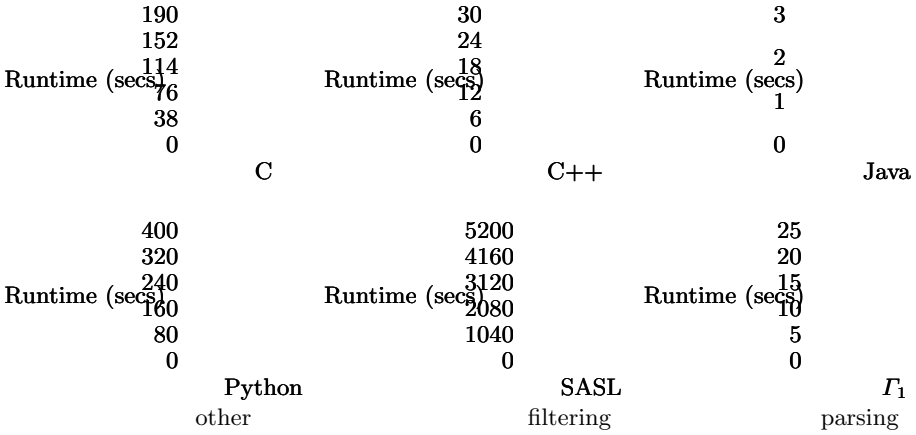

Fig. 1. Runtime comparison between SGLR (first col.) and SRNGLR (second col.). The *other* bar accounts for the time taken to read and process the input string and parse table.

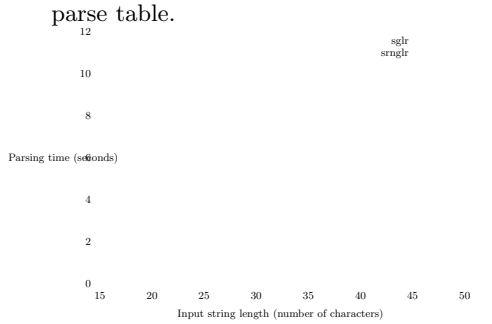

Fig. 2. Comparison of SGLR and SRNGLR parsing time for Γ_1.

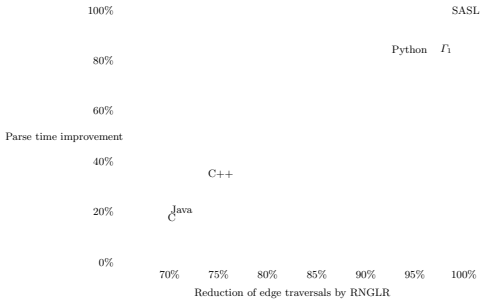

Fig. 3. Correlation between saving of edge traversals and parsing speedup.

large programs written in highly ambiguous scripting languages such as Python and SASL. Also, we may conclude that it introduces a significant improvement for less ambiguous or non-ambiguous languages and that the shape of a grammar highly influences the filtering speed.

3 SGLR and RNGLR

In this section we outline the RNGLR and SGLR algorithms and highlight the main differences between them. There are four main differences between the SGLR and RNGLR algorithms:

- Different parse tables formats are used; SLR(1) [29] versus RN [12].
- SGLR does more traversals of the GSS during parsing than RNGLR.
- Different parse forest representations are used; maximally shared trees [27] versus SPPF's [19].

– SGLR implements disambiguation filters [7] whereas RNGLR does not.

The RNGLR algorithm combines adaptations in the parse table generation algorithm with simplifications in the parser run-time algorithm. It is based on Tomita's algorithm, called Generalized LR (GLR) [24]. GLR extends the LR parsing algorithm to work on all context-free grammars by replacing the stack of the LR parsing algorithm with a Graph Structured Stack (GSS). Using the GSS to explore different derivations in parallel, GLR can parse sentences for grammars with parse tables that contain LR conflicts rather efficiently. However, the GLR algorithm fails to terminate on certain grammars. Farshi's algorithm fixes the issue in a non-efficient manner, by introducing extra searching of the GSS [16]. This algorithm is the basis for SGLR. The RNGLR algorithm fixes the same issue in a more efficient manner.

RNGLR introduces a modified LR parse table: an RN table. RN tables are constructed in a similar way to canonical LR tables, but in addition to the standard reductions, reductions on right nullable rules are also included. A right nullable rule is a production rule of the form $A ::= \alpha\beta$ where $\beta \stackrel{*}{\Rightarrow} \varepsilon$[3]. By reducing the left part of the right nullable rule (α) early, the RNGLR algorithm avoids the problem that Tomita's algorithms suffered from and hence does not require Farshi's expensive correction. However, since the right nullable symbols of the rule (β) have not been reduced yet it is necessary to pre-construct the parse trees of those symbols. These nullable trees are called ε-trees and since they are constant for a given grammar, they can be constructed at parse table generation time and included in the RN parse table. The early RN reduction will construct a full derivation simply by including the pre-constructed trees.

It is well known that the number of parses of a sentence with an ambiguous grammar may grow exponentially with the size of the sentence [9]. To avoid exponential complexity, GLR-style algorithms build an efficient representation of all possible parse trees, using subtree sharing and local ambiguity packing. However, the SGLR and RNGLR algorithms construct parse trees in different ways and use slightly different representations. RNGLR essentially follows the approach described by Rekers – the creation and sharing of trees is handled directly by the parsing algorithm – but does not construct the most compact representation possible. The SGLR algorithm uses the ATerm library [27] to construct parse trees thereby taking advantage of the maximal sharing it implements. This approach has several consequences. The parsing algorithm can be simplified significantly by replacing all parse tree creation and manipulation code with calls to the ATerm library. Although the library takes care of all sharing, the creation of ambiguities and cycles requires extra work (see Section 4.1).

As previously mentioned, in addition to the different construction approaches, a slightly different representation of parse forests is used. RNGLR labels interior nodes using non-terminal symbols and uses packing nodes to represent ambiguities [22]. SGLR labels interior nodes with productions and represents ambiguous

[3] α, β are possibly empty lists of terminals and non-terminals, ϵ is the empty string and $\stackrel{*}{\Rightarrow}$ represents a derivation in zero or more steps

trees using ambiguity clusters labeled by non-terminal symbols. The reason that production rules are used to label the interior nodes of the forest is to implement some of the disambiguation filters that are discussed later in this section.

The SGLR algorithm is different from RNGLR mainly due to the filters that are targeted at solving lexical ambiguity. Its filters for priority and preference will be discussed as well. SGLR introduces the following four types of filters: follow restrictions, rejects, preferences and priorities. Each filter type targets a particular kind of ambiguity. Each filter is derived from a corresponding declarative disambiguation construct in the SDF grammar formalism [7]. Formally, each filter is a function that removes certain derivations from parse forests (sets of derivations). Practically, filters are implemented as early in the parsing architecture as possible, i.e. removing reductions from parse tables or terminating parallel stacks in the GSS.

Four filter types. We now briefly define the semantics of the four filter types for later reference. A *follow restriction* is intended to implement longest match and first match behavior of lexical syntax. In the following example, the -/- operator defines a restriction on the non-terminal I. Its parse trees may not be followed immediately by any character in the class [A-Za-z0-9_], which effectively results in longest match behavior for I:

$$I ::= [A\text{-}Za\text{-}z0\text{-}9\_]* \qquad I \text{ -/- } [A\text{-}Za\text{-}z0\text{-}9\_] \tag{3.1}$$

In general, given a follow restriction A -/- α where A is a non-terminal and α is a character class, any parse tree whose root is $A ::= \gamma$ will be filtered if its yield in the input string is immediately followed by any character in α. Multiple character follow restrictions, as in A -/- $\alpha_1.\alpha_2 \ldots \alpha_n$, generalize the concept. If each of the n characters beyond the yield of A, fit in their corresponding class α_i the tree with root A is filtered. Note that the follow restriction incorporates information from beyond the hierarchical context of the derivation for A, i.e. it is not context-free.

The *reject* filter is intended to implement reservation, i.e. keyword reservation. In the following example, the {reject} attribute defines that the keyword `public` is to be reserved from I:

$$I ::= [A\text{-}Za\text{-}z][A\text{-}Za\text{-}z0\text{-}9\_]* \qquad I ::= \text{“}public\text{”}\{\text{reject}\} \tag{3.2}$$

In general, given a production $A ::= \gamma$ and a reject production $A ::= \delta\{\text{reject}\}$, all trees whose roots are labeled $A ::= \delta\{\text{reject}\}$ are filtered and any tree whose root is labeled $A ::= \gamma$ is filtered if its yield is in the language generated by δ. Reject filters give SGLR the ability to parse non-context-free languages such as $a^n b^n c^n$ [29].

The *preference* filter is intended to select one derivation from several alternative overlapping (ambiguous) derivations. The following example uses the {prefer} attribute to define that in case of ambiguity the preferred tree should be the only one that is not filtered. The dual of {prefer} is {avoid}.

$$I ::= [A\text{-}Za\text{-}z][A\text{-}Za\text{-}z0\text{-}9\_]* \qquad I ::= \text{“public”}\ \{\text{prefer}\} \tag{3.3}$$

In general, given n productions $A ::= \gamma_1$ to $A ::= \gamma_n$ and a preferred production $A ::= \delta\{\text{prefer}\}$, any tree whose root is labeled by any of $A ::= \gamma_1$ to $A ::= \gamma_n$ will be filtered if its yield is in the language generated by δ. All trees whose roots are $A ::= \delta\{\text{prefer}\}$ remain. Dually, given an avoided production $A ::= \kappa\{\text{avoid}\}$ any tree whose root is $A ::= \kappa\{\text{avoid}\}$ is filtered when its yield is in one of the languages generated by γ_1 to γ_n. In this case, all trees with roots $A ::= \gamma_1$ to $A ::= \gamma_n$ remain. Consequently, the preference filter can not be used to recognize non-context-free languages.

The *priority* filter solves operator precedence and associativity. The following example uses priority and associativity:

$$E ::= E \text{ “}\rightarrow\text{” } E\{\text{right}\} \quad > \quad E ::= E \text{ “or” } E\{\text{left}\} \tag{3.4}$$

The $>$ defines that no tree with the “$\rightarrow$” production at its root will have a child tree with the “or” at its root. This effectively gives the “$\rightarrow$” production higher precedence. The {right} attribute defines that no tree with the “$\rightarrow$” production at its root may have a first child with the same production at its root. In general, we index the $>$ operator to identify for which argument a priority holds and map all priority and associativity declarations to sets of indexed priorities. Given an indexed priority declaration $A ::= \alpha B_i \beta >_i B_i ::= \delta$, where B_i is the ith symbol in $\alpha B_i \beta$, then any tree whose root is $A ::= \alpha B_i \beta$ with a subtree that has $B_i ::= \delta$ as its root at index i, is filtered. The priority filter is not known to extend the power of SGLR beyond recognizing context-free languages.

4 SRNGLR

We now discuss the amalgamated algorithm SRNGLR that combines the scannerless behaviour of SGLR with the faster parsing behaviour of RNGLR. The SRNGLR algorithm is mainly different in the implementation of SGLR's filters at parse table generation time. All of SGLR's filters need to be applied to the static construction of RNGLR's ϵ-trees. However, there are also some changes in the other stages, parse-time and post-parse filtering. The reject filter was changed for clarification and for improving the predictability of its behavior. Note however that the latter change was applied to both SGLR and RNGLR before measuring performance differences.

4.1 Construction of ϵ-trees

The basic strategy is to first construct the complete ϵ-trees for each RN reduction in a straightforward way, and then apply filters to them. We collect all the productions for nullable non-terminals from the input grammar, and then for each non-terminal we produce all of its derivations, for the empty string, in a top-down recursive fashion. If there are alternative derivations, they are collected under an ambiguity node.

We use maximally shared ATerms [6] to represent parse trees. ATerms are directed acyclic graphs, which prohibits by definition the construction of cycles.

However, since parse trees are not general graphs we may use the following trick. The second time a production is used while generating a nullable tree, a cycle is detected and, instead of looping, we create a cycle node. This special node stores the length of the cycle. From this representation a (visual) graph can be trivially reconstructed.

Note that this representation of cycles need not be minimal, since a part of the actual cycle may be unrolled and we detect cycles on twice visited productions, not non-terminals. The reason for checking on productions is that the priority filter is specific for productions, such that after filtering, cycles may still exist, but only through the use of specific productions.

4.2 Restrictions

We distinguish single character follow restrictions from multiple lookahead restrictions. The first are implemented completely statically, while the latter have a partial implementation at parse table generation time and a partial implementation during parsing.

Parse table generation. An RN reduction $A ::= \alpha \cdot \beta$ with nullable tree T_β in the parse table can be removed or limited to certain characters on the lookahead. When one of the non-terminals B in T_β has a follow restriction B -/- γ, T_β may have less ambiguity or be filtered completely when a character from γ is on the lookahead for reducing $A ::= \alpha \cdot \beta$. Since there may be multiple non-terminals in T_β, there may be multiple follow restrictions to be considered.

The implementation of follow restrictions starts when adding the RN reduction to the SLR(1) table. For each different kind of lookahead character (token), the nullable tree for T_β is filtered, yielding different instances of T_β for different lookaheads. While filtering we visit the nodes of T_β in a bottom-up fashion. At each node in the tree the given lookahead character is compared to the applicable follow restrictions. These are computed by aggregation. When visiting a node labelled $C ::= DE$, the restriction class for C is the union of the restriction classes of D and E. This means that C is only acceptable when both follow restrictions are satisfied. When visiting an ambiguity node with two children labeled F and G, the follow restrictions for this node are the intersections of the restrictions of F and G. This means that the ambiguity node is acceptable when either one of the follow restrictions is satisfied.

If the lookahead character is in the restricted set, the current node is filtered, if not the current node remains. The computed follow restrictions for the current node are then propagated up the tree. Note that this algorithm may lead to the complete removal of T_β, and the RN reduction for this lookahead will not be added. If T_β is only partially filtered, and no follow restriction applies for the non-terminal A of the RN reduction, the RN reduction is added to the table, including the filtered ϵ-tree.

Parser run-time. Multiple character follow restrictions cannot be filtered statically. They are collected and the RN-reductions are added and marked to be conditional as lookahead reductions in the parsetable. Both the testing of the follow restriction as well as the filtering of the ϵ-tree must be done at parse-time.

Before any lookahead RN-reduction is applied by the parsing algorithm, the ϵ-tree is filtered using the follow restrictions and the lookahead information from the input string. If the filtering removes the tree completely, the reduction is not performed. If it is not removed completely, the RN reduction is applied and a tree node is constructed with a partially filtered ϵ-tree.

4.3 Priorities

Parse table generation. The priority filters only require changes to be made to the parse table generation phase; the parser runtime and post parse filtering phases remain the same as SGLR. The priority filtering depends on the chosen representation of the ϵ-trees (see also Section 3); each node holds a production rule and cycles are unfolded once. Take for example $S ::= SS\{\text{left}\}|\epsilon$. The filtered ϵ-tree for this grammar should represent derivations where $S ::= SS$ can be nested on the left, but *not* on the right. The cyclic tree for S must be unfolded once to make one level of nesting explicit. Then the right-most derivations can be filtered. Such representation allows a straightforward filtering of all trees that violate priority constraints. Note that priorities may filter all of the ϵ-tree, resulting in the removal of the corresponding RN reduction.

4.4 Preferences

Parse table generation. The preference filter strongly resembles the priority filter. Preferences are simply applied to the ϵ-trees, resulting in smaller ϵ-trees. However, preferences can never lead to the complete removal of an ϵ-tree.
Post-parse filter. RN reductions labeled with {prefer} or {avoid} are processed in a post-parse filter. This was already present in SGLR and has not needed any changes.

4.5 Rejects

Parse table generation. If any nullable production is labeled with {reject}, then the empty language is not acceptable by that production's non-terminal. If such a production occurs in an ϵ-tree, we can statically filter according to the definition of rejects in Section 3. If no nullable derivation is left after filtering, we can also remove the entire RN reduction.
Parser run-time. Note that we have changed the original algorithm [29] for reject filtering at parser run-time for *both* SGLR and SRNGLR. The completeness and predictability of the filter have been improved. The simplest implementation of reject is to filter redundant trees in a post-parse filter, directly following the definition of its semantics given in Section 3. However, the goal of the implementation is to prohibit further processing on GSS stacks that can be rejected as early as possible. This can result in a large gain in efficiency, since it makes the parsing process more deterministic, i.e. there exist on average less parallel branches of the GSS during parsing.

The semantics of the reject filter is based on syntactic overlap, i.e. ambiguity (Section 3). So, the filter needs to detect ambiguity between a rejected production $A ::= \gamma\{\text{reject}\}$ and a normal production for $A ::= \delta$. The goal is to stop further processing reductions of A. For this to work, the ambiguity must be detected *before* further reductions on A are done. Such ordering of the scheduling of reductions was proposed by Visser [29]. However, the proposed ordering is not complete. There are grammars for which the ordering does not have the desired effect and rejected trees do not get filtered. Especially nested rejects and rejects of nullable productions lead to such issues. Later alternative implementations of Visser's algorithm have worked around these issues at the cost of filtering too many derivations.

Instead we have opted for not trying to order reductions anymore and to implement an efficient method for not using rejected productions in derivations. The details of this reject implementation are:

- Edges created by a reduction of a rejected production are stored separately in GSS nodes. We prevent other reductions traversing the rejected edges, thereby preventing possible further reductions on many stacks.
- In GLR, edges collect ambiguous derivations, and if an edge becomes rejected because one of the alternatives is rejected, it stays rejected.
- Rejected derivations that escape are filtered in a post-parse tree walker. They may escape when an alternative, non-rejected, reduction creates an edge and this edge is traversed by a third reduction before the original edge becomes rejected by a production marked with {reject}.

Like the original, this algorithm filters many parallel stacks at run-time with the added benefit that it is more clearly correct. We argue that: (A) we do not filter trees that should not have been filtered, (B) we do not depend on the completeness of the filtering during parse time, and (C) we do not try to order scheduling of reduce actions, which simplifies the code that implements SRNGLR significantly.

The Post-parse filter of rejects simply follows the definition of its semantics as described in Section 3. For the correct handling of nested rejects, it is imperative to apply the filter in a bottom-up fashion.

5 Related work

The cost of general parsing as opposed to deterministic parsing or parsing with extended lookahead has been studied in many different ways. Our contribution is a continuation of the RNGLR algorithm applied in a different context.

Despite the fact that general context-free parsing is a mature field in Computer Science, its worst case complexity is still unknown. The algorithm with the best asymptotic time complexity to date is presented by Valiant [25]. However, because of the high constant overheads this approach is unlikely to be used in practice. There have been several attempts at speeding the run time of LR parsers that have focused on achieving speed ups by implementing the handle

finding automaton (DFA) in low-level code, [4, 13, 17]. A different approach to improving efficiency is presented in [2, 3], the basic ethos of which is to reduce the reliance on the stack. Although this algorithm fails to terminate in certain cases, the RIGLR algorithm presented in [14] has been proven correct for all context-free grammars.

Two other general parsing algorithms that have been used in practice are the CYK [10, 15, 30] and Earley [11] algorithms. Both display cubic worst case complexity, although the CYK algorithm requires grammars to be transformed to Chomsky Normal Form before parsing. The BRNGLR [23] algorithm achieves cubic worst case complexity without needing to transform the grammar.

Note however that the SGLR and the SRNGLR algorithm described in this paper is more than a parsing algorithm. Filtering is a major factor too, which makes SRNGLR incomparable to other parsing algorithms.

6 Conclusions

We improved the speed of parsing and filtering for scannerless grammars significantly by applying the ideas of RNGLR to SGLR. The disambiguation filters that complement the parsing algorithm at all levels needed to be adapted and extended. Together the implementation of the filters and the RN tables make scannerless GLR parsing quite a bit faster. The application areas in software renovation and embedded language design are directly serviced by this. It allows experimentation with more ambiguous grammars, e.g. interesting embeddings of scripting languages, domain specific languages and legacy languages.

Acknowledgements. We are grateful to Arnold Lankamp for helping to implement the GSS garbage collection scheme for SRNGLR. The first author was partially supported by EPSRC grant EP/F052669/1.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. John Aycock and R. Nigel Horspool. Faster generalised LR parsing. In *Proceedings 8th International Compiler conference*, volume 1575 of *LNCS*, pages 32–46, Amsterdam, March 1999. Springer-Verlag.
3. John Aycock, R. Nigel Horspool, Jan Janousek, and Borivoj Melichar. Even faster generalised LR parsing. *Acta Inform.*, 37(9):633–651, 2001.
4. Achyutram Bhamidipaty and Todd A. Proebsting. Very fast YACC-compatible parsers (for very little effort). *Softw., Pract. Exper.*, 28(2):181–190, February 1998.
5. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.
6. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Softw., Pract. Exper.*, 30(3):259–291, 2000.

7. M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In R. Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.
8. Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for AspectJ. *SIGPLAN Not.*, 41(10):209–228, 2006.
9. Keneth Church and Ramesh Patil. Coping with syntactic ambiguity or how to put the block in the box on the table. *American Journal of Computational Linguistics*, 8(3–4):139–149, July–December 1982.
10. John Cocke and Jacob T. Schwartz. Programming languages and their compilers. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.
11. Jay Earley. An efficient context-free algorithm. *Comm. ACM*, 13(2):94–102, Feb 1970.
12. Giorgios Robert Economopoulos. *Generalised LR parsing algorithms*. PhD thesis, Royal Holloway, University of London, August 2006.
13. R. Nigel Horspool and Michael Whitney. Even faster LR parsing. *Softw., Pract. Exper.*, 20(6):515–535, June 1990.
14. Adrian Johnstone and Elizabeth Scott. Automatic recursion engineering of reduction incorporated parsers. *Sci. Comp. Programming*, 68(2):95–110, 2007.
15. T. Kasami and K. Torii. A syntax analysis procedure for unambiguous context-free grammars. *J. ACM*, 16(3):423–431, 1969.
16. Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR Parsing*, chapter 5, pages 61–75. Kluwer Academic Publishers, Netherlands, 1991.
17. Thomas J. Pennello. Very fast LR parsing. In *SIGPLAN Symposium on Compiler Construction*, pages 145–151. ACM Press, 1986.
18. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
19. D.J. Salomon and G.V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Not.*, 24(7):170–178, 1989.
20. D.J. Salomon and G.V. Cormack. The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Dept. of Computer Science, University of Manitoba, 1995.
21. Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, 2006.
22. Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. BRNGLR: a cubic Tomita-style GLR parsing algorithm. *Acta Inform.*, 44(6):427–461, 2007.
23. M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
24. L. G. Valiant. General context-free recognition in less than cubic time. *J. Comput. System Sci.*, 10:308–315, 1975.
25. M.G.J. van den Brand. *Pregmatic, a generator for incremental programming environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.
26. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Softw., Pract. Exper.*, 30(3):259–291, 2000.
27. Guido van Rossum. Python reference manual. `http://docs.python.org/ref/`.
28. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
29. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Inform. and control*, 10(2):189–208, 1967.