Right Nulled GLR Parsers

ELIZABETH SCOTT and ADRIAN JOHNSTONE
Royal Holloway, University of London

The right nulled generalized LR parsing algorithm is a new generalization of LR parsing which provides an elegant correction to, and extension of, Tomita's GLR methods whereby we extend the notion of a *reduction* in a shift-reduce parser to include *right nulled* items. The result is a parsing technique which runs in linear time on LR(1) grammars and whose performance degrades gracefully to a polynomial bound in the presence of nonLR(1) rules. Compared to other GLR-based techniques, our algorithm is simpler and faster.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: General context-free grammars, generalized LR parsing

In this article we advocate the replacement of *ad hoc* extensions to deterministic parser generators with theoretically tractable, yet efficient, generalised parsers. The technical contribution is a new generalization of LR parsing which provides an elegant correction to, and extension of, Tomita's GLR methods whereby we redefine the notion of a *reduction* in a shift-reduce parser to include *right nulled* items. That is, in addition to reductions corresponding to items $A ::= \alpha\cdot$, we also apply reductions corresponding to items of the form $A ::= \alpha \cdot \beta$, where β reduces to the empty string. The result is a parsing technique which runs in linear time on LR(1) grammars and whose performance degrades gracefully to a polynomial bound in the presence of non-LR(1) rules. Compared to other GLR-based techniques, our algorithm is simpler and faster, both in theory and on grammars for real programming languages. We shall show speed-up factors of 10.8 on a highly nonLR(1) grammar for COBOL, 6.8 on a near-LR(1) grammar for ANSI-C, and just over 4 for ISO-7185 Pascal, and we give a grammar for which the searching component of the GLR algorithm is reduced from $O(n^3)$ to $O(n^2)$.

Authors' addresses: E. Scott, A. Johnstone, Department of Computer Science, Royal Holloway, University of London, Egham, Surrey, TW20 0EX, United Kingdom; email: adrian@cs.rhul.ac.uk.

To motivate the work we give an overview of the development of parsing theory and practice since the early 1970's, examining a variety of approaches to extending deterministic parsers and their application to compiler writing projects. We then give a more detailed description of the development of generalized LR (GLR) parsing based on Tomita's graph structured stack approach, before presenting our right nulled GLR parsing algorithm and the associated parse table transformations. We also present a proof of correctness for our algorithm which, to our knowledge, is the first formal proof for a Tomita-style algorithm. This is significant because early versions of GLR parsing were subsequently found to fail on certain classes of grammar rule. Finally, we present a range of experimental results comparing our approach to the traditional GLR algorithm.

1. THE DEATH AND AFTERLIFE OF PARSING RESEARCH

It is well known amongst mainstream computer scientists that parsing is *done*. Early programming languages, of which COBOL and FORTRAN are the main survivors, were designed and implemented before the full development of context-free parsing theory and they remain difficult languages to parse using standard tools. During the Algol-60 project, the correspondence between Backus' form of syntax description and Chomsky's context-free languages was recognised and from that came a new discipline of language design and implementation. Nevertheless, early parsers tended to be general: Irons [1961] describes a backtracking recursive descent parser for Algol-60 and a roughly contemporary Japanese-to-English translator uses a version of what is now called the CYK algorithm [Sakai 1962].

By the mid-1970's, practical parser generator tools employing algorithms such as LALR parsing [DeRemer 1969] and LL(1) predictive parsing [Lewis II and Stearns 1968] had been demonstrated allowing automatic generation of parsers from grammars for real programming languages. Essentially, we had techniques for converting human-comfortable programming notations into intermediate forms in time proportional to the length of the input. This was clearly good enough, and interest in the compiler theory community shifted to other aspects of the translation process since ". . . there is strong evidence that the restricted classes of grammars for which we can construct these efficient [LL(k) and LR(k)] parsers are adequate to specify the syntactic features of programming languages that are normally specified by context-free grammars." [Aho and Ullman 1972, p. 333]

Nevertheless, a plethora of parser generator tools sporting a range of extensions to the standard bottom-up and top-down approaches continued to appear, almost as if nobody had told compiler writers that near-deterministic techniques were sufficient. Of course, on a trivial level, deterministic techniques are *not* sufficient: The conventional grammatical descriptions of the IF-THEN-ELSE construct are ambiguous and no ambiguous grammar can be LR(1) or LL(1). The classical solution is to implement a simple disambiguation rule such as *longest match*, which in shift-reduce parsers corresponds to performing shifts rather than reduces where conflicts arise: Such an approach is immediately suggestive

of other forms of nondeterminism resolution and many authors have described extensions to deterministic algorithms, giving rise to this diversity of parser generator tools.

1.1 Extending Deterministic Parsers

Once we accept the possibility of imposing linear-time behaviour on nondeterministic parsers by forcing the parser to ignore particular paths, it is tempting to extend the approach. YACC, which, with its modern successor Bison, constitutes the most venerable and probably most widely-used parser generator, implements the strategy recommended in Aho et al. [1975] that allows languages to be described using ambiguous grammars. Disambiguation is then achieved using supplementary priority and associativity declarations. Sadly, when used in this way, YACC can surprise the naïve user since disambiguation declarations may be used to 'fix' shift-reduce errors regardless of whether they arise from the judicious use of deliberate ambiguity or whether they represent unintended nondeterminisms in the grammar. The user ends up with a grammar that YACC accepts without errors, but the specific language recognised by the YACC-generated parser may not be what the user intended.

Extended lookahead in the underlying parser automaton can be used to bridge some LR(1) and LL(1) nondeterminisms, but the size of the lookahead sets (and thus the automaton) grows exponentially in worst-case. It is well known that there is an LR(1) grammar for every LR(k) language, so from a theoretician's point of view, the utility of, say, an LR(3) parser generator might seem limited. In practice, users want to use *their* grammar rather than applying the transformations necessary to convert an LR(3) grammar to LR(1). The analogy with programming is strong: We have expended great effort in the design of human-comfortable programming notations that shield the user from the details of the underlying machine, improving comprehensibility and reliability as well as productivity. By the same token, language implementers want to express language syntax using rules that are comfortable for them and which allow the underlying semantics of the language to be specified in a natural way. We shall return to this point.

In practice, LR(k) and LL(k) parser generators are rare. The best example is Terence Parr's [1996] PCCTS/ANTLR family of tools which incorporate his Ph.D. thesis work on a 'linear approximation' to LL(k) parsing. The tools perform well on grammars that are broadly LL(1) but which have some rules that are LL(k) for small k. This is a useful capability because many LL(1) conflicts arise from locally nonleft-factored rules in which a small extension of the lookahead can resolve nondeterminisms.

Rather than extending the lookahead in the underlying automaton, many authors have described parsers which, when encountering a nondeterministic configuration, select one of multiple actions for evaluation, subsequently backtracking if this putative parse fails. Backtracking is easily added to both LR and LL algorithms (see, for instance, BtYACC [Dodd and Maslov 1995] and PRECC [Breuer and Bowen 1995]), but claims of generality should be treated conservatively, especially if the backtracking search is essentially depth-first.

Aho and Ullman [1972, p. 466] described a backtracking formalism called TDPL (*top down parsing language*) and noted that 'it can be quite difficult to determine what language is defined by a TDPL program'. The root of this problem is that the depth-first singleton match strategy adopted by most backtracking parsers corresponds to performing first match on an *ordered* grammar. We can simulate the behaviour of a longest match parser if the rules associated with the longest match are tried first, and on this basis, PRECC, for instance, claims generality. However, it can be difficult (especially for the naïve user) to ensure that the grammar is correctly ordered, and in fact, it may be shown that there are grammars for which no such ordering exists. One computationally expensive solution is to return the set of all outstanding putative matches at each backtrack stage, thus maintaining all backtrack contexts: GRDP [Johnstone and Scott 1998] provides a *prototyping mode* which provides full generality in this way, as well as a production mode for the confident user that only returns the first putative match.

Apart from the nasty surprises associated with incorrectly ordered grammars, backtracking results in exponential time complexity and so the acceptability of the resulting parser depends critically on the probability of deeply nested backtracking being encountered in practice. Sheil [1976] showed how to impose a polynomial time bound on such parsers by populating a *well-formed substring table* which effectively caches intermediate matches, but we know of no production parser generator that uses this technique.

A trend exemplified by PCCTS and its successors has been to allow *semantic* and *syntactic* predicates to select between nondeterministic options. A semantic predicate effectively provides a feedback mechanism from the translator's semantics to the parser, allowing, for instance, the type of an operand to determine which form of an expression to match against. A syntactic predicate instructs the parser to perform a local lookahead operation, only proceeding with the full parse if the predicate succeeds. This brings with it the dangers inherent in backtracking noted already, but in a particularly insidious form: It is quite possible to establish a set of predicates which allows several alternative parses of a string, or indeed, no parses, and it is very difficult in general to formally describe the language matched by a predicated parser. Apart from the ordering of the rules required for longest match parsing, we have the added difficulty of ensuring that the predicates are consistent with one another and that they correctly control the parser so that the required derivation is delivered.

1.2 Generalized Parsing and Compiler Writing

Any parsing technique which delivers performance by ignoring some potential parses should be treated with caution because of the difficulty of establishing exactly what language is matched by the parser. When testing parsers, and by extension, compilers, it is difficult to provide full coverage. By their very nature, the inputs to a compiler are of linguistic complexity rather than the relatively constrained inputs supplied to, say, a menu-driven application. It is not uncommon to find features in a language that are very rarely used and bugs

associated with their semantics (or even their parsing, arising from one of the unpleasant effects previously described) may take a long time to uncover.

The community has a well developed theory of context-free grammars which we are not using because performance concerns have led us to use nongeneral techniques extended with *ad hoc* mechanisms for nondeterminism resolution. Does this matter? Well, we know that conventional engineering disciplines are underpinned by a body of pragmatic mathematical theory which is used to provide 'black box' reliability to designers. Computer science is notoriously underprovided with such techniques and it is generally accepted that the failure rate of large computer-based projects is unacceptably high. It is at the very least a disappointment that parsing (which *did* yield to theoretical analysis) is not supported by completely general and reliable tools, and that in practice, compiler writers cannot use the grammar which seems natural to them, but have to tune their grammars to the underlying parsing technology and hope that nothing gets broken in the process.

The constraints of deterministic parsers have fundamentally affected language design and implementation. This is most clearly demonstrated by the case of C++. Stroustrup [1994, p. 68–69] writes eloquently of the difficulties he faced when constructing Cfront (the embryonic C++ translator) at a time when there was no LALR grammar for ANSI-C: He resorted to an uncomfortable mix of top-down and bottom-up parsing tricks to achieve his desired syntax. Today, we still have no useful LALR(1) C++ grammar: On the contrary, we believe that the difficulty of parsing C++ with conventional tools has stimulated the development of the extensions described in the previous section.

One might expect ANSI and ISO language standards to describe language syntax using grammars that are directly usable with widely distributed tools and, of course, we have the theoretical tools to do just that. Canonical grammars would result, which would provide a formal foundation for commercial compilers. In practice, we cannot even achieve this for C++, which is currently perhaps our most widely-used programming language. With the honourable exception of ANSI-C, standardisation committees seem not to prioritize this need: The ANSI/ISO Pascal standard grammar, for instance, contains many nondeterministic rules even after lexical issues have been resolved. Given that Pascal comes from the Wirth tradition of simple LL(1) style languages, this seems almost perverse.

1.3 Approaches to General Parsing

It is widely believed that general techniques are impractical and that it is necessary to use more limited techniques for 'real' applications, even though some of the earliest high-level language parsers used exponential or high-polynomial order techniques [Irons 1961] and yet were still acceptable even on primitive hardware. Since the early 1970's, when LR and LL techniques became dominant, we have seen instruction execution rates per second rise from around 10^3 to around 10^9. This increased performance has encouraged a resurgence of general techniques challenging this belief: For instance, several compilers

for Eiffel use Earley parsing to allow the rules governing punctuation placement to be relaxed and parser generators using GLR algorithms are being reported [McPeak and Necula 2004]. Microsoft Research has developed a compiler for the C# language that uses a GLR parser to allow the (ambiguous) grammar from the language standard to be used directly, and to allow easy experimentation with language constructs from other languages such as Modula-3 [Hanson and Proebsting 2003]. Significantly, Bison itself has recently acquired a general parsing mode which will allow many users to experiment with more flexible language descriptions [Eggert 2003]. In detail, although Bison's new mode is described as GLR, it is really a stack-splitting algorithm. It does not remerge stacks, so exponential growth can occur: It is not hard to write a small grammar that causes Bison's new mode to run out of memory on strings of just a few tokens. Nevertheless, the permeation of general parsing into such widely-used tools indicates that there is a rôle for practical generalized parsing algorithms.

In this section we provide a course taxonomy of general parsing algorithms so as to motivate our study of Tomita-style GLR parsers. For more detail, Grune and Jacobs [1990] provides an informal survey of such techniques and Graham and Harrison [1976] article provides a particularly useful unified treatment of CYK and Earley parsing. Nederhof and Satta [Nederhof 1994; Nederhof and Satta 1996, 2004] have also analysed CYK, Earley, and GLR parsing in terms of *tabular* parsing.

The simplest general algorithms have uncomfortable worst-case performance. Iron's backtracking parser is closely related to the top-down backtracking parsers described earlier, and has exponential worst-case performance. Another exponential algorithm was described by Unger [1968]: It is straightforward to implement but incorporation of semantic actions requires additional machinery, whereas semantics are easily incorporated into recursive descent parsers.

A variety of authors described an algorithm that constructs, for a string α of length n, an $n \times n$ *recognition matrix* in which location (i, j) contains the set of nonterminals that match a substring of α of length j starting at position i. The most well known presentations are by Cocke [Hays 1967], Younger [1967], and Kasami [1965], hence the CYK algorithm. The basic algorithm requires grammars to be in Chomsky normal form for which unambiguous derivations may be obtained in $O(n^2 \log n)$ time. Valiant [1975] showed how to construct the same recognition matrix as the transitive closure of an initial matrix, yielding an algorithm whose time bound is that of Boolean matrix multiplication. Although the best currently known approach yields an asymptotic bound of $O(n^{2.376})$, the constants of proportionality render Valiant's approach impractical for all but the longest strings.

Earley's [1970] parser, which is used by some Eiffel compilers, may also be described as building a recognition matrix, but in this case the entries are sets of *grammar positions* (or LR(0) items in Knuth's terminology). Graham and Harrison's [1976] article develops parallel presentations of CYK and Earley in terms of recognition matrices, and their analysis allows them to make very useful comments on implementation details. Unambiguous context-free

grammars may be recognised by Earley's algorithm in $O(n^2)$ time and ambiguous grammars in $O(n^3)$ time.

Tomita's *generalized LR* (GLR) parser builds on deterministic shift-reduce parsing by performing breadth-first search, synchronised on token input. Essentially, the string is read left-to-right, and at each string position, all possible reductions are performed before executing the shift operations. To maintain multiple contexts, multiple parse stacks are required. Stacks with the same prefix can, of course, share a single representation of that prefix. What may be less clear is that, as a result of the context-free nature of string matches, stacks with the same state on the top may also be merged. Tomita calls the resulting structure a graph structured stack (GSS) (and although it is really a stack structured graph, the terminology is now standard and will be used here). The GSS is made up of a sequence of $n+1$ *frontiers* (where n is the length of the string) containing up to s nodes, one for each of the s states in the underlying parse table. A node on frontier U_i may have out-edges linking it to any node on frontiers $U_0 - U_i$: hence, the size of the GSS is at most $O(n^2)$. A detailed exposition will be found in Section 3.1 to follow.

1.4 The Development of GLR Parsers

The GLR algorithm was developed by Tomita for applications in natural language parsing. Tomita noted that programming language translators were able to parse thousands of tokens per second, whereas contemporary natural language parsers (typically using CYK-style chart parsers) were realistically limited to phrases containing only tens of words. If a natural language grammar contains a large deterministic core, then a deterministic parser augmented with structures to handle nondeterminism and ambiguity might be competitive.

The GLR landscape is rich in variants, a fact often overlooked by potential users. In Figure 1 we show some milestones in the development of GLR-style algorithms. The box on the lefthand side shows three toolsets that use GLR-style parsing and their relation to the algorithms shown centrally. The dashed box on the right refers to what we call *reduction incorporated* parsing: a technique related to GLR parsing which we shall summarise later.

The foundations of shift-reduce parsing are well known: Knuth's [1965] seminal article contains most of the relevant results, in particular, that the LR(1) languages are the languages of deterministic PDAs. Although Knuth concludes that the technique is impractical due to the size of the LR(1) automaton, De Remer's [1969] thesis describing LALR automata and his article [1971] on SLR parsing show how to produce parsers whose size is that of the LR(0) automaton, but whose parsing power approaches that of the full set of deterministic languages.

Lang [1974] gave a general framework for computing the traversals of nondeterministic push-down automata, and a modification which can be applied to produce derivation trees has been discussed in Billot and Lang [1989]. While Lang's algorithm is not directly applicable to LR(1) tables, the Tomita GSS can be interpreted as a concrete instance of Lang's theoretical structure, although it is not clear that Tomita was aware of Lang's work.

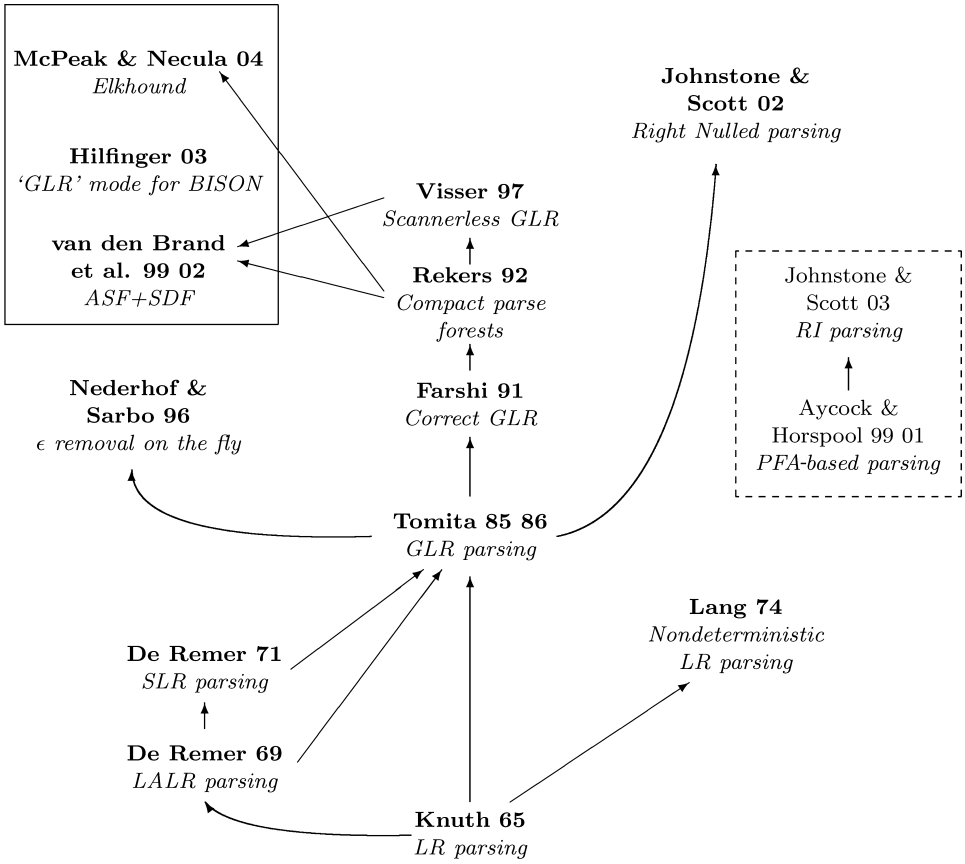


Fig. 1. The development of GLR parsers.

Tomita's original publications describe a family of five algorithms which we summarise in Section 3. His algorithms are correct for ϵ-free grammars, but the machinery that handles ϵ-rules fails to terminate for hidden left recursion. In a later volume [Bunt and Tomita 1991] he gives only the algorithm for ϵ-free grammars. In the same volume, Farshi [Nozohoor-Farshi 1991] describes a modified GLR recogniser that corrects the error in the Tomita recogniser, but his solution involves exhaustive searching of paths from the current GSS frontier, and as a result is much less efficient than Tomita's original algorithm. Most subsequent work uses the Farshi algorithm which we believe correctly handles all grammars, even though Farshi himself states "although no formal proof was provided here it is believed that the modified algorithm is a precompiled equivalent of Earley's algorithm with respect to its coverage."

In his [1992] thesis, Rekers added shared packed parse forest (SPPF) construction to Farshi's algorithm in a way which produces a more compact representation of the derivation trees, work which was incorporated into the ASF+SDF toolset [van den Brand et al. 2002], and which also forms the basis of the Elkhound parser generator [McPeak and Necula 2004] (in the introduction

to his thesis Rekers claims that one contribution of the thesis is a correction of the error in Tomita's algorithm. However, when the algorithm is introduced (see page 17 of Rekers [1992]) Rekers makes it clear that he is using Farshi's recogniser and it is the parse tree construction which he modifies and improves).

Visser [1997] modified Rekers' algorithm further to increase efficiency, particularly in cases where the input grammar's terminals are the individual characters, thus not incorporating a separate scanner; so-called scannerless parsing. However, both Rekers' and Visser's algorithms have Farhsi's underlying recogniser and inherit the increased searching required by his modification. Visser's scannerless parser is incorporated into current versions of ASF+SDF, and into Stratego/XT [Visser 2004].

An algorithm given by Nederhof and Sarbo [1996] addresses the problem with Tomita's algorithm by essentially carrying out ϵ-rule removal 'on the fly,' and the article includes a proof of the correctness of the algorithm. The additional rules generated by the ϵ-removal algorithm are not added to the grammar because this can substantially increase the number of states in the underlying parse table. Instead, the additional rules are applied at parse-time. Although the table size is not increased, this approach can still result in the application of many more rules, and furthermore, it is no longer possible to tell whether the application of a rule is valid before the rule is applied. So, the symbols on the paths being retraced must be matched against the rule being applied.

Aycock and Horspool's [1999] algorithm is a recogniser that reduces stack activity (compared to that of an LR parser) by using a finite automaton to handle all rules apart from those containing embedded recursion. This algorithm cannot be applied to grammars with hidden left recursion, but a fully general algorithm based on this approach has been developed [Johnstone and Scott 2003]. Extending the recogniser to a parser is not straightforward and table sizes can be very large, although it is possible to trade table size for parse-time stack activity. These algorithms have an affinity with the GLR approach because the call stacks associated with the embedded recursion are implemented by a Tomita-style GSS. However, they behave differently as to generalized LR parsers.

1.5 RNGLR Parsing

Our RNGLR algorithm takes as its starting point Tomita's ϵ-free algorithm and uses a modified parse table to effectively short-circuit the reductions for which Tomita needed to create subfrontiers. Our algorithm is simpler and more theoretically attractive than Farshi's algorithm since it involves only a modification of the input parse table rather than extensive additional searching. We have implemented Farshi's algorithm, our algorithm, and Tomita's ϵ-free algorithm, comparing them on grammars for ANSI-C, ISO7085 PASCAL, and IBM VS-Cobol. In these cases we found that Farshi's algorithm involves between four and ten times as much graph searching as our algorithm, and in fact, it is possible to construct grammars for which Farshi's algorithm requires $O(n)$ more searching, as we shall show in Section 7. We now describe the RNGLR algorithm starting with some basic definitions and terminology.

2. TERMINOLOGY

A *context-free grammar* (CFG) consists of a set $\mathbf{N}$ of nonterminal symbols, a set $\mathbf{T}$ of terminal symbols, an element $S \in \mathbf{N}$, called the start symbol, and a set of grammar rules of the form $A ::= \alpha$, where $A \in \mathbf{N}$ and α is a (possibly empty) string of terminals and nonterminals. We write X^* for the set of all strings of elements of a set X, and ϵ for the empty string.

A *derivation step* is an element of the form $\gamma A \delta \Rightarrow \gamma \alpha \delta$, where $\gamma, \delta \in (\mathbf{N} \cup \mathbf{T})^*$ and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence of derivation steps

$$\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$$

and we also write $\sigma \stackrel{*}{\Rightarrow} \tau$, $\sigma \stackrel{n}{\Rightarrow} \tau$, and if $n > 0$, $\sigma \stackrel{+}{\Rightarrow} \tau$.

A *sentential form* is a string α such that $S \stackrel{*}{\Rightarrow} \alpha$ and it is a *sentence* if $\alpha \in \mathbf{T}^*$. For a grammar Γ, the set $L(\Gamma)$ of sentences is the *language* generated by Γ.

A grammar Γ has *hidden right (or left) recursion* if for some nonterminal A, some string α, and some string $\beta \neq \epsilon$ such that $\beta \stackrel{*}{\Rightarrow} \epsilon$, $A \stackrel{*}{\Rightarrow} \alpha A \beta$ (or $A \stackrel{*}{\Rightarrow} \beta A \alpha$). A rule $A ::= \alpha\beta$, where $\beta \neq \epsilon$ but $\beta \stackrel{*}{\Rightarrow} \epsilon$, is said to be *right nullable*.

A *derivation tree* corresponding to a derivation $S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ of a sentential form α_n is defined inductively as follows. If $n = 0$, so $\alpha_n = S$, then the derivation tree is just a single node corresponding to and labelled by S. If $\alpha_j = x_1 \ldots x_{i-1} A x_{i+1} \ldots x_m$ and $\alpha_{j+1} = x_1 \ldots x_{i-1} y_1 \ldots y_p x_{i+1} \ldots x_m$ (so $A ::= y_1 \ldots y_p$), then a derivation tree for α_{j+1} is the tree obtained by adding child nodes corresponding to and labelled with $y_1, \ldots, y_p$ to the leaf node corresponding to this instance of A in a derivation tree for α_j.

Next we define the standard FIRST sets which are used in many well known parsing techniques. For a grammar symbol x and a string γ, we define

$$\begin{aligned}
\textsc{first}_{\mathbf{T}}(x) &= \{t \in \mathbf{T} \mid \text{for some string } \beta, x \stackrel{*}{\Rightarrow} t\beta\}, \\
\textsc{first}(\epsilon) &= \{\epsilon\} \\
\textsc{first}(x\gamma) &= \begin{cases} \textsc{first}_{\mathbf{T}}(x) \cup \textsc{first}(\gamma), & \text{if } x \stackrel{*}{\Rightarrow} \epsilon, \\ \textsc{first}_{\mathbf{T}}(x), & \text{otherwise.} \end{cases}
\end{aligned}$$

We now briefly review the standard LR parsing technique in order to establish the terminology that we use in the rest of the article, and to emphasise the view that we take when discussing our algorithm.

A traditional bottom-up parser essentially starts with an input string $x_1 \ldots x_k$ and reads the symbols from the left until a position i is reached such that there is a grammar rule $A ::= x_j \ldots x_i$, and a string w such that $S \stackrel{*}{\Rightarrow} x_1 \ldots x_{j-1} A w$. The parser then begins again with input string $x_1 \ldots x_{j-1} A x_{i+1} \ldots x_k$. This process is repeated until the input string reduces to the start symbol S, or until no further reductions are possible. Given any CFG, there is a deterministic finite state automaton (DFA), the so-called handle-finding automaton, which accepts precisely the strings $x_1 \ldots x_i$ for which there is a grammar rule $A ::= x_j \ldots x_i$ and a string w such that $S \stackrel{*}{\Rightarrow} x_1 \ldots x_{j-1} A w$. There are several handle-finding automata described in the literature [Aho et al. 1986], including the LR(0), SLR, and LALR automata. In this article we use LR(1) automata, but our results and discussions apply equally well to LR(0), SLR, and LALR DFAs.

An *item* is a pair $(A ::= \beta \cdot \gamma, x)$, where $A ::= \beta\gamma$ is a grammar rule and $x \in (\mathbf{T} \cup \{\$\})$, where \$ is the special end-of-string symbol which does not appear in the grammar itself. We assume that the reader is familiar with the construction of the standard LR(1) DFA (see, for example, Aho et al. [1986]). The DFA is labelled with sets of items, and a DFA state whose label contains an item of the form $(A ::= \gamma\cdot, x)$ is an accepting state. An example is given in Section 3.1. We shall assume that the grammar has an augmented start rule $S' ::= S$. The state whose label includes the item $(S' ::= S\cdot, \$)$ is the *final accepting* state of the DFA.

Given an input string, the parser traverses the DFA using the input symbols. When an accepting state is reached, to avoid reinputting the whole string, the path which was taken to reach that state is retraced and the traversal continues from that point. The states on the path are recorded on a stack and removed when that portion of the path is retraced. It is traditional in descriptions of LR parsers to push both the states and the corresponding input symbols onto the stack. The input symbols are not actually necessary, although they may be used as part of an additional derivation tree construction. We shall use a tree construction which is similar to Rekers [1992] because it produces more compact representations in the case of ambiguous grammars, so we do not push the grammar symbols onto the stack.

It is possible for an LR parser to be presented with a choice of actions either because a DFA accepting state also has a transition labelled with the next input symbol (a shift-reduce conflict) or because an accepting state is labelled with two different reductions of the form $(A ::= \gamma\cdot, b)$ (a reduce-reduce conflict). The obvious behaviour is for the parser to pursue all alternatives, but we cannot continue to use a single stack in this case. We could create copies of the stack and maintain them all, but the space required to store all these stacks can become infeasible. Sharing common parts of different stacks and remerging stacks which have the same top state allows these multiple stacks to be represented in worst-case $O(n^2)$ space. The resulting GSS is at the heart of Tomita's algorithms, which we now discuss.

3. TOMITA'S ALGORITHMS

Tomita's algorithms were designed to efficiently construct a graph structure (the GSS) which records all the possible traversals of a DFA for a given input string. As originally written, Tomita's algorithms include the grammar symbols in the stacks, and correspondingly, GSS nodes labelled with grammar symbols. However, we give an equivalent description in which the grammar symbol nodes are omitted. The theoretical issues that we consider are identical in both cases.

The basic idea is to maintain a list of states which can be reached using the input symbols seen thus far. When an input symbol, say, a, is read, any state which can be reached from a state in the current list along a transition labelled a is added to a new list of current states. When an accepting state h with an item $(A ::= \alpha\cdot, x)$ is reached, the paths labelled α which end at h are retraced and the states which can then be reached along transitions labelled A are added to the list of current states. This process is very similar to the

'subset construction' which is used to construct a DFA from an NFA, see, for example, Aho et al. [1986]. The main difference between the DFA-based general parser and the NFA traverser which forms the basis of the subset construction is that in the latter, the ϵ-closure contains all the states which can be reached via ϵ-transitions, while the 'reduction closure' in the former can only contain states which can be reached by retracing a path taken (we refer to these as input-related reduction closures, see Section 5.3).

In his original description Tomita [1986] gives five versions of his algorithm for constructing parsers, which we shall call Algorithms 0 to 4.

—*Algorithm 0* is an initial description of the algorithm for the case where the underlying parse table is conflict-free.
—*Algorithm 1* requires the underlying grammar to be ϵ-free, but works correctly on all CFGs with this property.
—*Algorithm 2* accepts all CFGs, but fails to terminate on grammars with rules which contain hidden left recursion.
—*Algorithm 3* is an extension of Algorithm 2 that constructs a slightly different GSS, which in turn results in more compact derivation tree representations. This fails to terminate in the same way as Algorithm 2.
—*Algorithm 4* contains additional mechanisms to construct a shared packed parse forest (a representation of the derivation trees of a given input string). This also fails to terminate in the same way as Algorithm 2.

As we shall discuss later, in order to make the construction efficient, Tomita's algorithms all ensure that edges in the GSS are only created from nodes that currently have no parents. If Algorithm 1 is used in the natural way with grammars containing right nullable rules, then it is possible for GSS edges to be created from nodes with existing parents. For this reason, Tomita's Algorithm 2 contains a complicated method for dealing with ϵ-rules. This modification increases both the size and the conceptual complexity of the construction of the GSS, and in some cases is nonterminating. Farshi's solution [Nozohoor-Farshi 1991] was to abandon the assumption that edges are only created from nodes with no parents and to use a modified version of Tomita's Algorithm 1 that searches for all paths in the GSS from currently 'active' nodes which contain the newly created edge.

In this section we present a version of Tomita's Algorithm 1, which we call Algorithm 1e, that has merely been extended to remove the assumption that all grammar rules have length of at least 1 (this is not a modification in the sense that Farshi's is because the algorithm works identically to the original on non-ϵ rules and behaves in an obviously equivalent way on ϵ-rules). In the next section we analyse Algorithm 1e and show how it can produce incorrect parsers for input grammars which contain rules with hidden right recursion. We shall then show, by extending the definition of what constitutes a reduction to include items of the form $(A ::= \alpha\cdot\beta, b)$, where $\beta \stackrel{*}{\Rightarrow} \epsilon$, that Algorithm 1e can be modified to work correctly for all CFGs. The resulting *right nulled generalized LR (RNGLR) algorithm* has essentially the same property as Tomita's algorithms in the sense that new paths are created in the GSS only by adding edges to the ends of

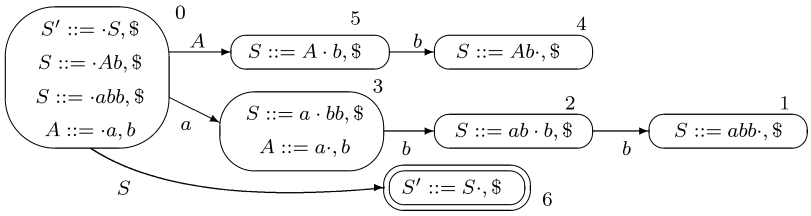
existing paths (in fact, there is a slight subtlety here in that we actually traverse only part of the path and edges are only added to the end of the traversed section). We give a proof that the RNGLR algorithm generates correct parsers for all CFGs which relies only on the correctness of the underlying LR(1) parsing algorithm. Furthermore, it turns out that our algorithm is actually slightly more efficient even than Tomita's for grammars which contain right nullable productions.

Before giving Algorithm 1e, we give an informal description of the algorithm.

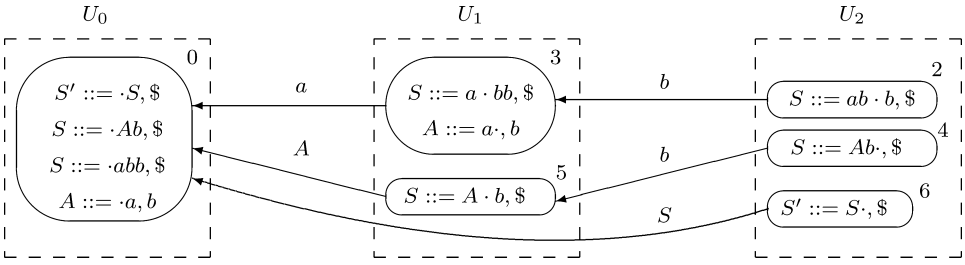
3.1 Graph Structured Stacks

A *graph structured stack* (GSS) is a directed graph associated with an LR(1) DFA and an input string $a_1 \dots a_d$. The GSS nodes are labelled with states of the DFA. Strictly, GSS edges are unlabelled, but we often put labels on the edges in the diagrams of this text to make it easier to see how the GSS edges correspond to transitions (edges) in the DFA. The nodes are grouped together into disjoint sets: an initial set, U_0, and one set, U_i, for each element a_i of the input string. For example, given the following grammar and corresponding LR(1) DFA,

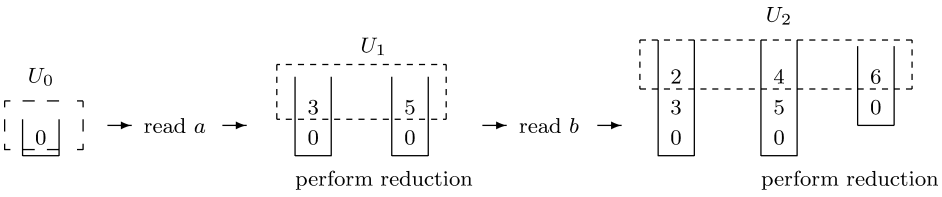
$$S ::= Ab \mid abb \qquad A ::= a \qquad (\Gamma_0)$$



the input ab results in the GSS

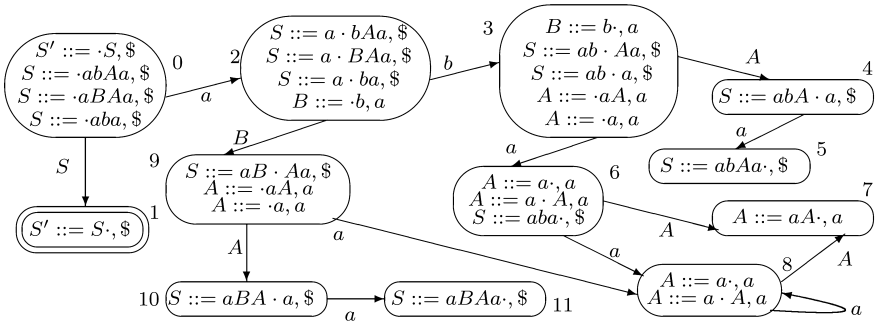


corresponding to the stack activity

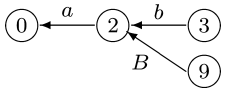


Stacks with the same prefix can share this prefix and stacks with a common top state can be recombined. We illustrate this using the grammar Γ_1, whose LR(1) DFA is shown in the following.

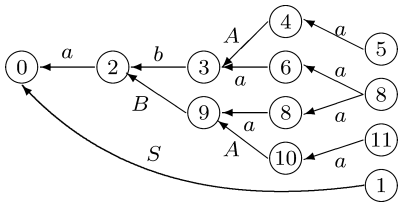
$$S ::= abAa \mid aBAa \mid aba \qquad A ::= a \mid aA \qquad B ::= b \qquad (\Gamma_1)$$



Given input *abaa*, we read the first two input symbols, traverse the DFA, and construct the stack 0, 2, 3. State 3 contains the item $(B ::= b\cdot, a)$, so we trace back to state 2 and then move to state 9, resulting in the stack 0, 2, 9. The common prefix of these two stacks is shared.



We then read the next a and create stacks 0, 2, 3, 6 and 0, 2, 9, 8. Applying the reductions creates the stacks 0, 2, 3, 4 and 0, 2, 9, 10. Reading the final a, we create the stacks 0, 2, 3, 6, 8 and 0, 2, 9, 8, 8, which can be recombined, and two other stacks 0, 2, 3, 4, 5 and 0, 2, 9, 10, 11. Applying the reductions to these latter stacks generates two copies of the stack 0, 1, which are combined.



We say that a node is at *level i* if it is in U_i. If a level i node v is labelled with the DFA state h, and h contains an item of the form $(A ::= \alpha\cdot, a_{i+1})$, then we say that v *has a valid reduction*. Conversely, we say that a reduction via the rule $A ::= \alpha$ is *valid for a node* v which is at level i and has label h, if the DFA state h contains the item $(A ::= \alpha\cdot, a_{i+1})$. In other words, valid reductions are reductions which can be applied when the input $a_1 \ldots a_i$ has been read and the lookahead input symbol is a_{i+1}. We now describe how the levels U_i are constructed in general.

First we assume that we have constructed the GSS for input $a_1 \ldots a_{i-2}$, where $i \geq 2$, so levels $U_0, \ldots, U_{i-2}$ have been constructed, and we assume that the symbol a_{i-1} has been pushed onto the appropriate stacks with

corresponding states (we shall describe how to construct U_0 later). The nodes in the GSS labelled with these states form the basis of the set U_{i-1}.

Each node $u \in U_{i-1}$ is *processed* by applying the valid reductions in the state which labels u. Reductions of length 0 are applied to u and reductions of length $q \geq 1$ are applied down paths of length q from u.

—*Non-ϵ reductions* Reductions $(Y ::= \alpha\cdot, a_i)$ of length $|\alpha| = q \geq 1$ are applied to nodes $u \in U_{i-1}$ by finding each node w which is on a path of length q from u. If the label of w is say, h,we find or create a node $y \in U_{i-1}$ labelled with the target of the transition labelled Y from h in the DFA, then create an edge (y, w) if one does not already exist.

—*ϵ reductions* A reduction $(X ::= \cdot, a_i)$ of length 0 is applied to a node $u \in U_{i-1}$, whose label is say, h, by finding or creating a node $z \in U_{i-1}$ labelled with the target of the transition labelled X from the DFA node h, then creating an edge (z, u).

—*Shifts* We continue selecting nodes and paths and applying reductions until all possible reductions have been carried out. Then we check each node $u \in U_{i-1}$, labelled say, h, and if h has a transition labelled a_i to a state say, l, then a node t labelled l is found or created in U_i, together with an edge (t, u). The nodes constructed in this way form the basis of U_i, to which the input-related reduction closure process described earlier is then applied.

—*Construction of U_0* We begin the whole process by constructing a base node, $v_0 \in U_0$, at level 0 in the graph, labelled with the start state, 0, from $\mathcal{T}$. This corresponds to an LR stack which just contains the start state. We then form the input-related reduction closures and new base sets, as described previously.

—*Termination and Acceptance* If at any point, the base set constructed for some U_i is empty, then the process terminates and reports failure. Otherwise, when the construction of U_d is complete, we check to see if it contains a node whose label includes the item $(S' ::= S\cdot, \$)$. If it does, then the process terminates and reports success, otherwise it terminates and reports failure.

3.2 Constraining the Creation of New Paths in the GSS

When building the U_i in the GSS, reductions in a state which labels a node must be applied down all paths from this node. Because a new node is not created if a node with the required label already exists (since the existing one is reused), it is possible to create new paths from existing nodes by adding new edges, thus reductions of length greater than 0 must be applied down these new paths.

One of the data management issues that an algorithm for GSS construction needs to address is how to ensure that all valid reductions are applied down all paths from a given node in U_i. To do this without retracing all paths each time that a new edge is created, we need to keep a record of which paths have already been traversed, or equivalently, of those valid paths which have not yet been traversed. If edges are only created at the front ends of existing paths, that is, from nodes with no parents, then paths to be retraced can be recorded

by recording their first edge. Algorithm 1e, which we shall now formally define, has this property, provided that the underlying grammar does not contain any right nullable rules.

3.3 Algorithm 1e

We assume that the input DFA is represented as a table, $\mathcal{T}$, whose rows are labelled with DFA state numbers, and whose columns are labelled with the grammar symbols and the end-of-string symbol, \$. The grammar rules are numbered. The entry in row h, column x, is denoted $\mathcal{T}(h,x)$ and is a set of actions. If there is a DFA transition labelled x from h to k, then pk (rather than sk or gk) is in $\mathcal{T}(h,x)$ (we use pk, *push* k rather than the more traditional sk, *shift* k for a terminal and gk, *goto* k for a nonterminal because the action taken in either case is to create an edge in the GSS). If $(A ::= \alpha\cdot, x)$ is an item in h, then rm is in $\mathcal{T}(h,x)$, where m is the number of the rule $A ::= \alpha$. Algorithm 1e uses three sets to manage the selection of nodes during the computation of the sets U_i. When a node in U_i is created, it is inserted in a set $\mathcal{A}$ which contains the nodes still to be processed. The set $\mathcal{R}$ contains the reductions, together with the first edge of the paths down which they are to be applied. The set $\mathcal{Q}$ contains pairs (v,k), where $v \in U_i$ has a label h and there is a DFA transition labelled a_{i+1} from h to k.

Input a CFG whose production rules are uniquely numbered, a DFA constructed from this grammar in the form of a table $\mathcal{T}$, and an input string $a_1 \ldots a_d$.

```
ALG1e {
 create a node v0 labelled with the start state 0 of the DFA
 set U0 = {v0}, A = ∅, R = ∅, Q = ∅, a_{d+1} = $
 for i = 0 to d do PARSE_SYMBOL(i)
 let s be the final accepting state of the DFA
 if U_d contains a node whose label is s report success else report failure }

PARSE_SYMBOL(i) {
 A = U_i,  U_{i+1} = ∅
 while A ≠ ∅ or R ≠ ∅ { if A ≠ ∅ do ACTOR(i) else do REDUCER(i)}
 do SHIFTER(i) }

ACTOR(i) {
 remove v from A, and let h be the label of v
 if pk ∈ T(h, a_{i+1}) add (v, k) to Q
 for each entry rj ∈ T(h, a_{i+1})
      if the length of j is 0 add (v, j) to R
      else add (u, j) to R, for each successor node u of v }

REDUCER(i) {
 remove (u, j) from R
 let m be the length of the righthand side of rule j and let X be
          the symbol on the lefthand side of rule j
 if m = 0 {
   let k be the label of u and let pl ∈ T(k, X)
   if there is no node in U_i labelled l then create a new node, v,
        in the GSS, labelled l and add v to U_i and to A
   let v be the node in U_i labelled l
```

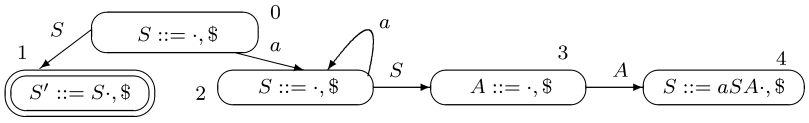
```
    if there does not exist an edge (v, u) in the GSS {
        create an edge (v, u) in the GSS
        if v ∉ A  forall rk ∈ T(l, a_{i+1}), with (length k)≠ 0, add (u, k) to R }}
  else
    for each node w which can be reached
                from u along a path of length m − 1  do {
      let k be the label of w and let pl ∈ T(k, X)
      if there is no node in U_i labelled l then create a new
            node, v, in the GSS labelled l and add v to U_i and to A
      let v be the node in U_i labelled l
      if there does not exist an edge (v, w) in the GSS {
        create an edge (v, w) in the GSS
        if v ∉ A  forall rk ∈ T(l, a_{i+1}), with (length k)≠ 0, add (w, k) to R}}}

SHIFTER(i) {
  while Q ≠ ∅ {remove an element (v, k) from Q
                if there is no node, w, labelled k in U_{i+1} create one
                if there is no edge (w, v) in the GSS create one }}
```

3.4 Right Nullable Rules and Hidden Right Recursion

In this section we shall give an example which demonstrates the problem with creating edges from nodes with existing parents.

Consider the following grammar, Γ_2, and associated LR(1) DFA,

$$1.\ S ::= aSA \qquad 2.\ S ::= \epsilon \qquad 3.\ A ::= \epsilon \qquad (\Gamma_2)$$



We use Algorithm 1e to construct the GSS for the input string aa. Neither of the states 0 or 2 has a valid reduction on the input symbol a, so the GSS constructed at the point where the second a has been read is just a simple stack, as shown in the following.

When the node w is processed, $(w, 2)$ will be added to the set $\mathcal{R}$, recording that a reduction by rule 2 must be applied from node w. Then $(w, 2)$ will be removed from $\mathcal{R}$, causing the node v to be created and $(v, 3)$ to be added to $\mathcal{R}$. Next $(v, 3)$ will be removed from $\mathcal{R}$, causing the node u to be created, and $(u, 1)$ to be added to $\mathcal{R}$. When $(u, 1)$ is removed from $\mathcal{R}$ we find that there is already a node, $v \in U_2$, labelled 3, so a new edge is created from this node to t. The resulting GSS is shown on the right in the preceding diagram. Since state 3 does not contain any reductions of length greater than 0, nothing is added to $\mathcal{R}$ when the reduction

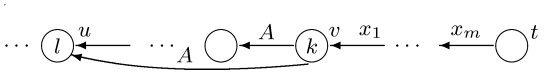
$(u, 1)$ is processed, and so at this point $\mathcal{A}$, $\mathcal{R}$, and $\mathcal{Q}$ are empty and the algorithm terminates. But U_2 does not contain a node labelled 1, so the algorithm reports failure even though $aa \in L(\Gamma_2)$.

It is not difficult to show (see Scott et al. [2000]) that Algorithm 1e can only create an edge from a node that has parents if the underlying grammar contains right nullable rules. However, it is not always the case that Algorithm 1e fails on grammars with right nullable rules. In some cases, it is possible to fail if the reductions are applied in one order, but to succeed if they are applied in a different order. The interested reader can verify that this is the case for the grammar

$$\begin{array}{lcllcl} S & ::= & BDab \mid aDad & A & ::= & aBB \mid \epsilon \\ D & ::= & aAB & B & ::= & \epsilon \end{array}$$

and input string $aaab$.

One situation in which the order of reduction application cannot be chosen is when a new reduction path is created as a result of applying a reduction down the path to which the new edge will be added. That is, it can't be chosen when a reduction is applied from a node t down to a node u and the new path is added from a node, v, between t and u, to u.



In this case, it is not possible to construct the path $t, \ldots, v, u$ before the node t is removed from $\mathcal{A}$ because the path from v to u is only created after the reductions in t have been added to—and then removed from—$\mathcal{R}$. This is the situation in the aforementioned example Γ_2.

It is not hard to show that in the situation described in the previous paragraph, we must have $A ::= \delta A x_1 \ldots x_d$ for some δ, where $d \geq 1$ and $x_1 \ldots x_d \stackrel{*}{\Rightarrow} \epsilon$. So, the grammar contains hidden right recursion. Thus, in this sense, it is hidden right recursion which ultimately breaks Tomita's Algorithm 1 when it is applied to general CFGs.

Tomita addressed the problem with right nullable rules in his Algorithm 2 by subdividing the sets U_i and creating a new subset every time an ϵ-reduction is applied. This algorithm fails to terminate when given a grammar with hidden left recursion.

Farshi [Nozohoor-Farshi 1991] addressed the issue by revisiting all the nodes in U_i and finding all paths which contain the newly added edge, and then applying all reductions down these paths. This method is inefficient in that it has to search for all paths containing a specified edge. We now describe our solution, which involves increasing the set of items which generate reductions to include right nullable rules, and then using Algorithm 1e essentially without modification (in fact, we could use Algorithm 1e without any modification at all, but we introduce a few simplifications which allow us to do slightly less graph traversal in the GSS construction when the grammar has right nullable rules).

4. RIGHT NULLED GLR PARSERS

In this section we describe a general parsing algorithm which we prove is correct on all CFGs and input strings. The algorithm is based on Algorithm 1e, but reductions whose final symbols are nullable are applied when the last nonnullable symbol is seen.

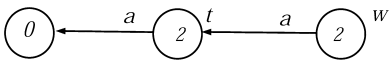
4.1 Right Nulled (RN) Tables

We construct the DFA for a grammar in the normal fashion, however, in the corresponding table, $\mathcal{T}$, we store the reductions, slightly differently. If there is a transition from state h to state k labelled x, then $pk \in \mathcal{T}(h, x)$. If h contains an item of the form $(A ::= x_1 \dots x_m \cdot B_1 \dots B_t,\ b)$, where $A \neq S'$ and $t = 0$ or $B_p \stackrel{*}{\Rightarrow} \epsilon$ for all $1 \leq p \leq t$, then $r(A, m) \in \mathcal{T}(h, b)$. If l contains $(S' ::= S\cdot, \$)$, then acc is added to $\mathcal{T}(l, \$)$. If $S' \stackrel{*}{\Rightarrow} \epsilon$, then acc is added to $\mathcal{T}(0, \$)$.

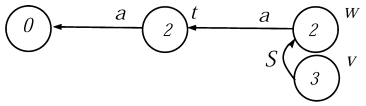
We illustrate this using the example Γ_2 from Section 3.4, which has the following RN-table

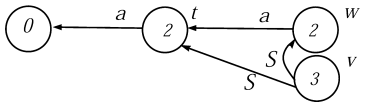
	$	a	A	S
0	r(S,0)/acc	p2		p1
1	acc			
2	r(S,0)/r(S,1)	p2		p3
3	r(S,2)/r(A,0)		p4	
4	r(S,3)			

We run Algorithm 1e on this table with input string aa. Steps $i = 0$ and $i = 1$ are carried out as before, resulting in the GSS



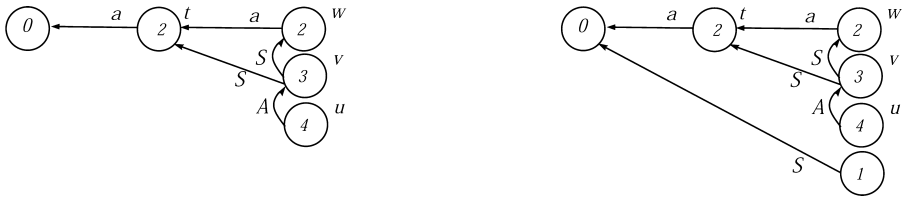
with $U_2 = \{w\} = \mathcal{A}$, $\mathcal{R} = \mathcal{Q} = \emptyset$, and $a_3 = \$$. We then remove w from $\mathcal{A}$ and add the reductions $(w, S, 0)$ and $(t, S, 1)$ to $\mathcal{R}$. Removing the first of these causes a new node, v, labelled 3, to be added to U_2 and to $\mathcal{A}$, resulting in the following below leftmost GSS. Then v is removed from $\mathcal{A}$, causing $(w, S, 2)$ and $(v, A, 0)$ to be added to $\mathcal{R}$. If we remove $(t, S, 1)$ from $\mathcal{R}$ next, then we trace back along a path of length 0 to t, and since the entry in $\mathcal{T}(2, S)$ is $p3$ and we already have a node $v \in U_2$ labelled 3, we create a new edge (v, t) and add $(t, S, 2)$ to $\mathcal{R}$.





Now suppose that we remove $(w, S, 2)$ from $\mathcal{R}$. Tracing back along the path of length 1 from w, we get to t, but there is already a node $v \in U_2$ labelled 3 and an edge (v, t), so no action is taken. Removing the reduction $(v, A, 0)$ from $\mathcal{R}$ results in the construction of a new node, $u \in U_2$, labelled 4 (see the GSS on the left) and processing this node causes $(v, S, 3)$ to be added to $\mathcal{R}$. When this reduction is removed from $\mathcal{R}$, we find that there are two paths of length 2 from

v: one to t which does not result in any additional construction, and one to the base node labelled 0. Since $p1 \in \mathcal{T}(0, S)$, we add a new node labelled 1 to U_2 and to $\mathcal{A}$.



There are no reductions in state 1 and removing $(t, S, 2)$ from $\mathcal{R}$ does not create any new nodes, so the algorithm terminates and correctly reports success.

4.2 Right Nulled GLR Parser (RNGLR)

As we have remarked, we can just run Algorithm 1e with RN-tables to generate correct parsers. However, we give a slightly modified algorithm which is more efficient.

Reductions of length m from v must be applied down all paths from v of length m. If $m = 0$, there can only ever be one such path, the empty path. If $m \geq 1$, then new paths of length m from v are created every time a new successor node is added to v. Thus, when a node, v, is created via a reduction of length greater than 0, we record in $\mathcal{R}$ all reductions of length greater than 0, together with the second node along the path from v, for subsequent execution. If the node has been created as the result of the application of a reduction of length 0 (a reduction $(B ::= \cdot\beta, a_{i+1})$), then any reductions of length greater than 0 will already have been applied from a previous node (this is the role of the new reduction items $(A ::= \alpha \cdot B, b)$ in the DFA table), so no reductions are added to $\mathcal{R}$ in this case.

If a new path from an existing node v is created, then this must be as a result of applying a reduction. If the reduction is of length greater than 0, then the valid reductions of length greater than 0 in v and this new path are recorded in $\mathcal{R}$ for subsequent processing.

Our algorithm is in the same style as Algorithm 1e, except we do not have a separate ACTOR. In our algorithm when a node v is created, any shift which is possible on the next input symbol is immediately recorded in the set $\mathcal{Q}$ for execution once the input-related reduction closure has been completed. Since nodes are processed as soon as they are generated, the SHIFTER processes nodes and a temporary $\mathcal{Q}'$ is needed to hold the pending shifts for the next level. Also, rather than storing the reduction number in $\mathcal{R}$, we store triples (u, X, m), where u is the start node, m is the length of the reduction to be applied, and X is the lefthand side of the reduction rule.

The RNGLR Algorithm

Input: an RN-table $\mathcal{T}$, an input string $a_1 \ldots a_d$

PARSER {

```
 if d = 0 { if acc ∈ T(0, $) report success  else report failure }
 else {
  create a node v0 labelled with the start state 0 of the DFA.
  set U0 = {v0}, R = ∅, Q = ∅, a_{d+1} = $, U1 = ∅, ..., Ud = ∅
  if pk ∈ T(0, a1) add (v0, k) to Q
  forall r(X, 0) ∈ T(0, a1) add (v0, X, 0) to R
  for i = 0 to d  while Ui ≠ ∅ { while R ≠ ∅ do REDUCER(i)
                                 do SHIFTER(i) }
  if the DFA final accepting state is in Ud report success  else report failure }}

REDUCER(i) {
 remove (v, X, m) from R
 find the set χ of nodes which can be reached from v along a path of
     length (m − 1), or length 0 if m = 0
 for each node u ∈ χ  do {
  let k be the label of u and let pl ∈ T(k, X)
  if there is a node w ∈ Ui with label l  {
    if there is not an edge from w to u  {
     create an edge from w to u
     if m ≠ 0 { forall r(B, t) ∈ T(l, a_{i+1}) where t ≠ 0, add (u, B, t) to R }}}
  else {
    create a node w ∈ Ui labelled l and an edge (w, u)
    if ph ∈ T(l, a_{i+1}) add (w, h) to Q
    forall r(B, 0) ∈ T(l, a_{i+1}) add (w, B, 0) to R
    if m ≠ 0 { forall r(B, t) ∈ T(l, a_{i+1}) where t ≠ 0, add (u, B, t) to R }}}}

SHIFTER(i) {
 if i ≠ d {
  Q' = ∅
  while Q ≠ ∅ {
   remove an element (v, k) from Q
   if there is w ∈ U_{i+1} with label k  {
      create an edge from w to v
      forall r(B, t) ∈ T(k, a_{i+2}) where t ≠ 0 add (v, B, t) to R}
   else { create a node, w ∈ U_{i+1} labelled k and an edge (w, v)
        if ph ∈ T(k, a_{i+2}) add (w, h) to Q'
        forall r(B, t) ∈ T(k, a_{i+2}) where t ≠ 0 add (v, B, t) to R
        forall r(B, 0) ∈ T(k, a_{i+2}) add (w, B, 0) to R}}
  copy Q' into Q }}
```

5. CORRECTNESS OF THE RNGLR ALGORITHM

A language recognition algorithm is correct for a given CFG if, for any given any input string, the algorithm terminates and reports success if the input string is in the language generated by the grammar, and terminates and reports failure otherwise.

Our proof of the correctness of our algorithm depends on the correctness of the standard stack- and table-based parsing technique. We shall give a formal definition of what we mean by the language accepted by a (possibly nondeterministic) table-based parser, and we shall prove that for all CFGs, our algorithm (deterministically) accepts exactly the language accepted by an RN parser, and furthermore that this is the language accepted by the LR(1) parser.

We begin with a general definition of a parse table. We then define the operation of a stack-based machine with such a table on a given input string, and next we define the language accepted by this machine. These definitions are just standard definitions for LR(1) tables extended to include tables which may have conflicts and reductions applied on partially recognised handles.

5.1 Table-Based Parsers

A *parse table* for a grammar Γ is a table whose rows are labelled with distinct integers starting from 0, and whose columns are labelled with the terminals and nonterminals of the grammar together with the end-of-string symbol, \$. The entries in the table are sets of *actions* of the form pk, $r(A, m)$, and acc, where k is a row number, A is a nonterminal, and m is an integer. Entries in columns labelled with nonterminals can contain, at most, one element, which must be of the form pk. Entries in columns labelled with terminals or \$ can contain up to one action of the form pk, and arbitrarily many actions of the form $r(A, m)$. In the column labelled \$, the entries can also contain the action acc.

We call the parse table constructed from a grammar Γ using LR(1) items in the standard way the *LR(1) table* for Γ. For nonLR(1) grammars, the LR(1) table will contain some entries with more than one action. The *right nulled LR(1) table* (or the *RN table*) is described in Section 4.1.

We define a *table-based parser*, in the usual way, to be a stack-based machine with an associated parse table, $\mathcal{T}$, which takes as input a string of symbols. Initially, the stack contains 0, the label of the first row of the table. An *execution step* consists of looking at the current symbol, say, a, in the input string and the integer, say, h, on the top of the stack and (nondeterministically) selecting and executing an action from the set $\mathcal{T}(h, a)$. An *execution path* of a table-based parser for a given input string is a sequence of execution steps which start with 0 on the stack and the input pointer at the beginning of the string. A string u is *accepted* by a table-based parser if, on input $u\$$, there is some execution path of the parser whose last action is acc.

An LR(1) parser for a grammar Γ is a table-based parser whose associated table is the LR(1) table for Γ. An RN parser for Γ is a table-based parser whose associated table is the RN table for Γ. We shall now show that the LR(1) and RN parsers for a given CFG are equivalent.

5.2 Equivalence of LR(1) and RN Parsers

The proofs in this and the following section use the fact that if there is an execution path of an LR(1) or RN parser which results in a stack of the form $0, \ldots, k_1, \ldots, k_{q+1}, \ldots, h$, then there is a path in the DFA from k_1 to k_{q+1}, which is labelled $x_1, \ldots, x_q$, say. Furthermore, if $(A ::= \alpha \cdot \beta, a) \in k_{q+1}$, where $|\alpha| \geq q$, then $\alpha = \alpha' x_1 \ldots x_q$, $(A ::= \alpha' \cdot x_1 \ldots x_q \beta, a) \in k_1$, and if x_1 is a nonterminal, then $(x_1 ::= \cdot\delta, g) \in k_1$ for all rules $x_1 ::= \delta$ and $g \in \text{FIRST}(\beta y)$.

We begin with the following lemma which addresses the behaviour of the LR(1) parser on right nullable rules.

LEMMA 1. *Suppose that there is an execution path Θ of an LR(1) parser whose table is $\mathcal{T}$, which results in a stack of the form $0, h_1, h_2, \ldots, h_q$ and the input pointer pointing at symbol a. Suppose also that $(A ::= \alpha \cdot \beta, a) \in h_q$, where $\beta \stackrel{*}{\Rightarrow} \epsilon$. Then there is a continuation of Θ in which the input pointer is not moved and the stack takes the form $0, h_1, \ldots, h_i, h$, where $|\alpha| = (q-i)$ and $ph \in \mathcal{T}(h_i, A)$.*

PROOF. We prove by induction on the length of the derivation $\beta \stackrel{n}{\Rightarrow} \epsilon$ that there is a continuation Θ' of Θ in which the input pointer is not moved and the stack takes the form

$$0, h_1, \ldots, h_q, k_1, \ldots, k_p$$

where $|\beta| = p$ and $(A ::= \alpha\beta \cdot, a) \in k_p$. Then $r(A, q - i + p) \in \mathcal{T}(k_p, a)$ and this action can be used to extend Θ', resulting in the required stack.

If $n = 0$, then $\beta = \epsilon$, $p = 0$ and the new stack is the same as the original one.

Now suppose that $n \geq 1$ and that the result is true for items of the form $(X ::= \gamma \cdot \delta, a)$, where $\delta \stackrel{d}{\Rightarrow} \epsilon$ and $d < n$. Since $n \geq 1$, we must have $\beta = y_1\beta'$, where $y_1 \neq \epsilon$ and $y_1 \Rightarrow \tau \stackrel{d}{\Rightarrow} \epsilon$, so $d < n$. Then $(y_1 ::= \cdot\tau, a) \in h_q$, and so by induction we can extend the Θ so that the stack is of the form

$$0, h_1, \ldots, h_q, k'_1, \ldots, k'_s$$

where $|\tau| = s$ and $(y_1 ::= \tau \cdot, a) \in k'_s$. Then $r(y_1, s) \in \mathcal{T}(k'_s, a)$, so there is an execution step in which the input pointer is not moved and the stack becomes

$$0, h_1, \ldots, h_q, k_1$$

where $pk_1 \in \mathcal{T}(h_q, y_1)$ and $(A ::= \alpha y_1 \cdot \beta', a) \in k_1$. Now we have $\beta' \stackrel{f}{\Rightarrow} \epsilon$, where $1 + d + f = n$ and $|\beta'| = p - 1$, so $f < n$, and by induction, without moving the input pointer we can extend the execution path so that the stack is of the form

$$0, h_1, \ldots, h_q, k_1, \ldots, k_p$$

where $(A ::= \alpha y_1 \beta' \cdot, a) \in k_p$, as required. □

THEOREM 1. *For any CFG Γ, the LR(1) and RN parsers for Γ are equivalent in the sense that they accept the same set of strings.*

PROOF. Since the LR(1) table, $\mathcal{T}$, for Γ is a subset of the RN table, $\mathcal{T}'$, for Γ, for any execution path through the LR(1) parser, there is an identical execution path through the RN parser. Thus, any string accepted by the LR(1) parser for Γ will also be accepted by the RN parser for Γ.

If the empty string is accepted by the RN parser, then we must have $acc \in \mathcal{T}'(0, \$)$, and hence $S \stackrel{*}{\Rightarrow} \epsilon$. So the LR(1) parser will also accept ϵ.

To show that any nonempty string which is accepted by the RN parser for Γ is also accepted by the LR(1) parser, we show that if on input $a_1 \ldots a_{i+1}$ there is an execution path through the RN parser which results in the stack $0, h_1, \ldots, h_p$ and input pointer pointing at a_{i+1}, then there is an execution path through the LR(1) parser which results in the same stack and pointer position.

If the execution path is empty, then both parsers have stacks containing just the state 0 and the input pointer pointing at the first input symbol, so the result is true.

Now suppose that the execution path in question consists of $q \geq 1$ steps and that the first $(q-1)$ execution steps resulted in the stack $0, k_1, \ldots, k_t$ with the input pointer pointing at b (where $b = a_i$ or $b = a_{i+1}$). Suppose also that at the qth execution step, the action $act \in \mathcal{T}'$ was selected.

By induction we can assume that there is an execution path on input $a_1 \ldots a_n\$$ in the LR(1) parser which results in the stack $0, k_1, \ldots, k_t$ and the input pointer pointing at b. We let $a_{n+1} = \$$.

If act was acc, then we must have $b = \$ = a_{i+1}$, and since the string is nonempty, $k_t \neq 0$. So, k_t is the final accepting state of the DFA, and hence, $acc \in \mathcal{T}(k_t, \$)$. So the required execution path exists in the LR(1) parser.

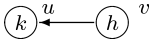
If act was pk, then to arrive at the given stack $0, h_1, \ldots, h_p$ with the pointer at a_{i+1}, we must have $b = a_i$, $k = h_p$, and $k_l = h_l$ for $1 \leq l \leq t = p-1$. Thus, the previous step would have been carried out on the stack $0, h_1, \ldots, h_{p-1}$ with the input pointer pointing at a_i. So, $pk \in \mathcal{T}(h_{p-1}, a_i)$ and the LR(1) parser could also extend its execution path using this action.

Finally, suppose that act was $r(A, m)$ so that at the previous step in the execution path the stack had the form $0, h_1, \ldots, h_{p-1}, g_1, \ldots, g_m$ and the input pointer was pointing at $a_{i+1} = b$. By induction, there is an execution of the LR(1) parser on the same input which results in the same stack and input pointer position. Since $r(A, m) \in \mathcal{T}'(g_m, a_{i+1})$, we must have $(A ::= \alpha \cdot \beta, a_{i+1}) \in g_m$, where $|\alpha| = m$ and $\beta \stackrel{*}{\Rightarrow} \epsilon$. Since the next step in the execution path of the RN parser results in the stack $0, h_1, \ldots, h_p$, we must have that $ph_p \in \mathcal{T}'(h_{p-1}, x_p)$, and hence that $ph_p \in \mathcal{T}(h_{p-1}, x_p)$. Then, by Lemma 1, there is a continuation of the LR(1) parser's execution path with results in the stack $0, h_1, \ldots, h_p$ and input pointer remaining at a_{i+1}, as required. □

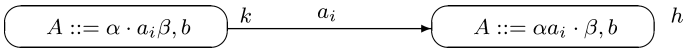
5.3 Notation and Properties of the GSS

We now want to show that the RNGLR algorithm correctly computes all the possible executions of an RN parser on a given input string. In order for us to do this, we need to note some general properties of a GSS.

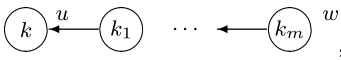
Suppose that the GSS constructed from input $a_1 \ldots a_d$ contains an edge



where $v \in U_i$, $u \in U_j$, and say, x labels the DFA transitions to h. If x is a terminal, then $j = i-1$ and $x = a_i$ (this corresponds to shifting a_i in the parser).

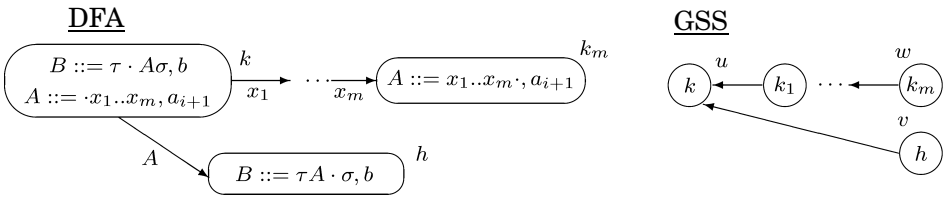


If x is a nonterminal, say, A, then there is a subgraph in the GSS of the form



,

where if x_j labels the DFA transitions to k_j, $1 \leq j \leq m$, $(A ::= x_1 \ldots x_m \cdot, a_{i+1}) \in k_m$, and there is a DFA transition from k to h labelled A.



We say that the GSS node v is *reduction-related* to the node w via a path of length m and a nonterminal A.

Formally, a set of nodes, U, in the GSS is *input-related reduction closed* if for each node, say, w, with label k_m which is at level i and for each valid reduction in w, that is, for each item $(A ::= x_1 \ldots x_m \cdot, a_{i+1})$ in k_m, if k is the label of a node which is reachable in the GSS from w along a path of length m, then there is a node in U which is at level i and has label h, where h is the state in the DFA which can be reached from k along the transition labelled A.

5.4 Correctness of the RNGLR Algorithm

We have already shown that an RN parser accepts the same language as the LR(1) parser for the same grammar, and we assume that the latter accepts precisely the language generated by the underlying grammar. The problem is that both of these machines are, in general, nondeterministic. We define a *determining algorithm* for a table-based parser to be an algorithm which determines whether or not, given an input string, there is an execution path through the table-based parser which results in *acc*.

We define a determining algorithm to be *correct* for a table-based parser if, given any input string, it terminates and reports success if there is an execution path of the table-based parser which reports *acc*, and terminates and reports failure otherwise. It is believed that Tomita's Algorithm 2 is correct for LR(1) parsers whose tables have been generated from CFGs without hidden left recursion, and that Algorithm 1e is correct for LR(1) parsers whose tables have been generated from CFGs without right nullable rules. We shall prove that the RNGLR algorithm given in Section 4.2 is a correct determining algorithm for RN parsers whose associated tables have been generated from any CFG.

LEMMA 2. *For all CFGs, the RNGLR algorithm given in Section 4.2 terminates for all input strings.*

PROOF. We suppose that the algorithm is using an RN table with N rows (states), constructed from a CFG Γ, and that the input string is $a_1 \ldots a_n$.

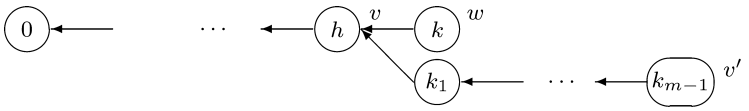
Each set U_i of level i nodes has only one node labelled with each state number, thus the GSS contains, at most, $(n+1)N$ nodes. An edge (u, v) is only created if one does not already exist between u and v, thus the GSS has, at most, $O(n^2)$ edges.

The **for** loops in REDUCER(i) iterate over finite sets which are not modified during the execution of the loop, thus this function will always terminate. The **for** loops in SHIFTER(i) iterate over table entries, and these are fixed. The **while** loop in SHIFTER(i) removes an element from $\mathcal{Q}$ at each iteration, and does not add any elements to $\mathcal{Q}$, thus SHIFTER(i) always terminates. So, to show that the algorithm always terminates we need to show that the inner **while** loop in the function PARSER terminates for each value of i. Each time REDUCER(i) executes, it removes an element from the set $\mathcal{R}$. Looking at the structure of REDUCER(i), we see that it only adds elements to $\mathcal{R}$ when a new edge is created in the GSS. We have already seen that there can only be a finite number of edges in the GSS, so REDUCER(i) must eventually stop adding elements to $\mathcal{R}$, but continue removing one each time it is executed. Thus, eventually, we will have $\mathcal{R} = \emptyset$ and the **while** loop will terminate, as required. □

In order to show that the RN table determining algorithm given in Section 4 accepts precisely the language accepted by the RN parser, we need the following lemma which addresses the impact of nullable nonterminals on the GSS and on the RN table.

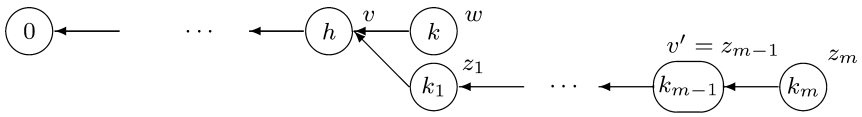
LEMMA 3. *Suppose that v, w are nodes labelled h, k, respectively, in the GSS constructed by the RNGLR algorithm from an RN table, $\mathcal{T}$, and input $a_1 \ldots a_n$. Suppose also that w and v lie in the same U_i, that there is an edge from w to v, and that X is the label of the DFA transitions to k. Then $X \stackrel{*}{\Rightarrow} \epsilon$, $pk \in \mathcal{T}(h, X)$, $r(X, 0) \in \mathcal{T}(h, a_{i+1})$, and hence the DFA state h contains an item $(X ::= \cdot\gamma, a_{i+1})$, where $\gamma \stackrel{*}{\Rightarrow} \epsilon$.*

PROOF. Since $w, v \in U_i$, the edge (w, v) must have been constructed as a result of processing a reduction (v', X, m), where there is a path of length $(m-1)$ (or of length 0 if $m = 0$) from v' to v, and we must have $pk \in \mathcal{T}(h, X)$.



If $m = 0$, then $v' = v$ and for $(v, X, 0)$ to be in $\mathcal{R}$ during the construction of U_i we must have $r(X, 0) \in \mathcal{T}(h, a_{i+1})$, and $X \stackrel{*}{\Rightarrow} \epsilon$ as required.

If $m > 0$, for (v', X, m) to be in $\mathcal{R}$ during the construction of U_i there must be a node $z_m \in U_i$ with label k_m such that there is an edge from z_m to v' and $r(X, m) \in \mathcal{T}(k_m, a_{i+1})$, so $(X ::= \alpha \cdot \beta, a_{i+1}) \in k_m$, where $|\alpha| = m$ and $\beta \stackrel{*}{\Rightarrow} \epsilon$.



Each of the nodes z_d must lie in U_j for some $j \leq i$. Since $v \in U_i$ and there is a path from each of these nodes to v, in fact they must all lie in U_i.

We now prove the result by induction on the order in which the edges in the GSS were created.

If (w, v) is the first edge created, then $v = v_0$ and $w \in U_0$. So, we must have $m = 0$ and the result is true in this case.

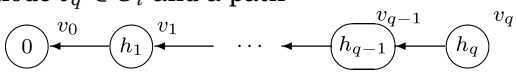
All of the edges (z_d, z_{d-1}), where $1 \leq d \leq m$ and $z_0 = v$, were created before the edge (w, v) and so, by induction, $y_d \stackrel{*}{\Rightarrow} \epsilon$ for $1 \leq d \leq m$, where y_d labels the DFA transitions to k_d. Since $(X ::= \alpha \cdot \beta, a_{i+1}) \in k_m$, we have $\alpha = y_1 \dots y_m$, so $\alpha \stackrel{*}{\Rightarrow} \epsilon$ and hence $X \stackrel{*}{\Rightarrow} \epsilon$. Furthermore, $(X ::= \cdot\alpha\beta, a_{i+1}) \in h$ and so $r(X, 0) \in \mathcal{T}(h, a_{i+1})$. □

THEOREM 2. *Given any CFG* Γ *and any input string* u*, the RNGLR algorithm given in Section 4.2 terminates and reports success if there is an execution of the LR(1) parser for* Γ *which results in acc, and terminates and reports failure otherwise.*

PROOF. If $u = \epsilon$, then the RNGLR algorithm reports success if and only if $S \stackrel{*}{\Rightarrow} \epsilon$, so the result follows from the correctness of the LR(1) parser. From Theorem 1 it is sufficient to show that the RNGLR algorithm terminates and reports success if there is an execution of the RN parser with results in *acc*, and terminates and reports failure otherwise. We have already shown that the algorithm always terminates, so we need to show that it reports success on input u if and only if there is an execution path through the RN parser which results in *acc*.

Let G be the GSS constructed from the RN table $\mathcal{T}$ for Γ and input $a_1 \dots a_n$, let $a_{n+1} = \$$, and let v_0 be the base node of G, the first node constructed.

($\Rightarrow$): We suppose that the RNGLR algorithm reports success. We shall show that if there is a node $v_q \in U_i$ and a path



in G, then there is an execution path through the RN parser on input $a_1 \dots a_n$ which results in the stack $0, h_1, \dots, h_q$ and input pointer pointing at a_{i+1}. Then, since the algorithm reports success, there is a node in U_n whose label, l, is the final accept state of the DFA. Thus, there is an execution path through the RN parser which results in l on the top of the stack and the input pointer pointing at $a_{n+1} = \$$. Since $acc \in \mathcal{T}(l, \$)$, this gives the result.

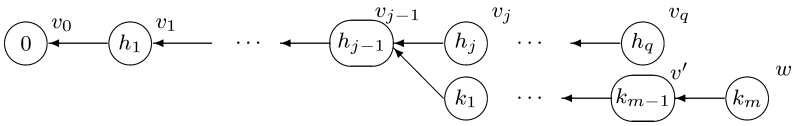
The proof is by induction on the order in which the edges in G are created.

If the path has no edges, then we must have $v_q = v_0$ and $i = 0$. The start configuration of an RN table-based parser ensures that there is an (empty) execution path which results in 0 on the stack and the input pointer pointing to a_1, so the result is trivially true. Now suppose that the edge (v_j, v_{j-1}) was the last of the edges in the path to be created.

If the edge (v_j, v_{j-1}) was created by the SHIFTER, then since it was the last edge in the path to be created, we must have $j = q$, $v_{q-1} \in U_{i-1}$ and $ph_q \in \mathcal{T}(h_{q-1}, a_i)$. Then, since the edges on the path from v_{q-1} to v_0 were created before the edge (v_q, v_{q-1}), by induction there is an execution path through the RN parser on input $a_1 \dots a_n$ which results in the stack $0, h_1, \dots, h_{q-1}$ with the input pointer at a_i. Since $ph_q \in \mathcal{T}(h_{q-1}, a_i)$, this execution path can then be extended to give the stack $0, h_1, \dots, h_{q-1}, h_q$ leaving the input pointer at a_{i+1}.

Now suppose that the edge (v_j, v_{j-1}) was created by the REDUCER. Since this is assumed to be the last edge created and since $v_q \in U_i$, this edge must have been created by REDUCER(i) while processing an element (v', X_j, m), where $ph_j \in \mathcal{T}(h_{j-1}, X_j)$ and there is a path from v' to v_{j-1} in G. Looking at REDUCER(i) and SHIFTER(i), we see that for (v', X_j, m) to be in $\mathcal{R}$ at this point we must have either

(1) $m = 0$, $v' = v_{j-1}$ and $r(X_j, 0) \in \mathcal{T}(h_{j-1}, a_{i+1})$, or
(2) $m \geq 1$ and there is a node $w \in U_i$ with label k_m such that there is an edge (w, v') and $r(X_j, m) \in \mathcal{T}(k_m, a_{i+1})$.



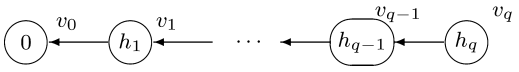
By induction, there is an execution path through the RN parser which results in the stack $0, h_1, \ldots, h_{j-1}$ with the input pointer at a_{i+1}. In case 1, since $r(X_j, 0) \in \mathcal{T}(h_{j-1}, a_{i+1})$, there is a continuation of the execution path which results in the stack $0, h_1, \ldots, h_{j-1}, h_j$ as required. In case 2, since all of the path from w to v_{j-1} was created before the edge (v_j, v_{j-1}), by induction there is an execution path Θ through the RN parser which results in the stack $0, h_1, \ldots, h_{j-1}, k_1, \ldots, k_m$ with the input pointer at a_{i+1}. Since $r(X_j, m) \in \mathcal{T}(k_m, a_{i+1})$ and $ph_j \in \mathcal{T}(h_{j-1}, X_j)$, there is a continuation Θ which again results in the stack $0, h_1, \ldots, h_{j-1}, h_j$.

From the operations in REDUCER(i), we see that we must have $v_j \in U_i$, and hence $v_d \in U_i$ for $j < d \leq q$. Then, by Lemma 3, $ph_d \in \mathcal{T}(h_{d-1}, X_d)$ and $r(X_d, 0) \in \mathcal{T}(h_{d-1}, a_{i+1})$, where X_d labels the transitions to h_d, $j < d \leq q$. Thus, we can continue Θ to generate stacks

$$\begin{gathered} 0, h_1, \ldots, h_{j-1}, h_j, h_{j+1} \\ 0, h_1, \ldots, h_{j+1}, h_{j+2} \\ \vdots \\ 0, h_1, \ldots, h_q \end{gathered}$$

without moving the input pointer, as required.

($\Leftarrow$): Now we suppose that there is an execution path through the RN parser which results in acc. We shall show that if there is an execution path through the RN parser on input $a_1 \ldots a_n$ which results in the stack $0, h_1, \ldots, h_q$ and input pointer pointing at a_{i+1}, then there is a node $v_q \in U_i$ and a path



in G. Then, since the RN parser results in acc, there is an execution path through the RN parser which results in l, the DFA final accepting state, being on the top of the stack and the input pointer pointing at $a_{n+1} = \$$. So, there is a node in U_n whose label is l, and the RNGLR algorithm will report success. We prove the result by induction on the number of steps in the execution path.

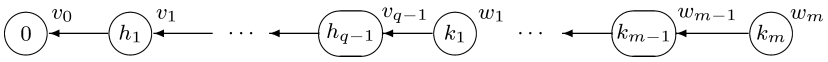
When the execution begins, the input pointer points at a_1 and the stack just contains state 0 and there is an (empty) path from $v_0 \in U_0$ to itself.

Now suppose that it takes M execution steps to result in the stack

$$0, h_1, \ldots, h_{q-1}, h_q$$

and input pointer pointing to a_{i+1}. If the last execution step (the one which resulted in the given stack and position) was a shift action, then it must have been $ph_q \in \mathcal{T}(h_{q_1}, a_i)$ and the stack must have been $0, h_1, \ldots, h_{q-1}$. By induction, there is a path in the GSS from a node $v_{q-1} \in U_i$ to the node v_0. If $q = 1$, then ph_q is added to $\mathcal{Q}$ at the start of the algorithm. Otherwise, ph_q is added to $\mathcal{Q}$ when the node v_{q-1} is created. Then, when SHIFTER(i) is executed, the node v_q and the edge from v_q to v_{q-1} will be created, as required.

Now suppose that the last execution step was a reduction, so that the stack was of the form $0, h_1, \ldots, h_{q-1}, k_1, \ldots, k_m$ with the input pointer pointing at a_{i+1}, and suppose that the DFA transitions to k_d are labelled y_d, $1 \leq d \leq m$. We have $r(X_q, m) \in \mathcal{T}(k_m, a_{i+1})$ with $ph_q \in \mathcal{T}(h_{q-1}, a_{i+1})$ and $(X_q ::= \alpha \cdot \delta, a_{i+1}) \in k_m$, where $|\alpha| = m$ and $\delta \stackrel{*}{\Rightarrow} \epsilon$. By induction, there is a node $w_m \in U_i$ and a path



If $v_{q-1} \in U_i$, then all of the nodes from w_m to w_1 must also be in U_i. Hence, by Lemma 3, we must have $y_d \stackrel{*}{\Rightarrow} \epsilon$, for $1 \leq d \leq m$. Then $(X_q ::= \cdot\alpha\delta, a_{i+1}) \in k_m$ and $\alpha = y_1 \ldots y_m$, so $r(X_q, 0) \in \mathcal{T}(h_{q-1}, a_{i+1})$ and when the node v_{q-1} was created, the pending action $(v_{q-1}, X_q, 0)$ would have been added to $\mathcal{R}$. Since $v_{q-1} \in U_i$, this action would be removed from $\mathcal{R}$ during the ith step of the GSS construction and at this point, a node $v_q \in U_i$, labelled h_q, and the edge (v_q, v_{q-1}) would have been added to the GSS if they were not already there.

Now suppose that $w_{p-1} \notin U_i$ and that $w_d \in U_i$ for $d \geq p$ (here, $w_0 = v_{q-1}$). Then $y_d \stackrel{*}{\Rightarrow} \epsilon$ for $p + 1 \leq d \leq m$, $(X_q ::= y_1 \ldots y_p \cdot y_{p+1} \ldots y_m\delta, a_{i+1}) \in k_p$ and so $r(X_q, p) \in \mathcal{T}(k_p, a_{i+1})$. If the edge (w_p, w_{p-1}) in the GSS was created by the SHIFTER, then since $w_p \in U_i$, it must have been created by SHIFTER($i - 1$). Since $r(X_q, p) \in \mathcal{T}(k_p, a_{i+1})$ and $p \geq 1$, (w_{p-1}, X_q, p) would have been added to $\mathcal{R}$ by SHIFTER($i - 1$). If (w_p, w_{p-1}) was created by the REDUCER, it must have been REDUCER(i) because $w_p \in U_i$. For REDUCER(i) to create the edge (w_p, w_{p-1}) it must be processing a reduction of the form (v', y_p, m') and there must be a path of length $(m' - 1)$, or length 0 if $m' = 0$, from v' to w_{p-1}. If $m' = 0$, then we must have $v' \in U_i$ and $w_{p-1} = v'$, which is contrary to the assumption that $w_{p-1} \notin U_i$. Thus, we must have $m' \neq 0$ and (w_{p-1}, X_q, p) would have been to added $\mathcal{R}$ by REDUCER(i). Thus, in either case $(w_{p-1}, X_q, p) \in \mathcal{R}$ and when this reduction is processed, a node $v_q \in U_i$, labelled h_q, and the edge (v_q, v_{q-1}) would have been added to the GSS if they were not already there. □

6. DERIVATION TREE CONSTRUCTION

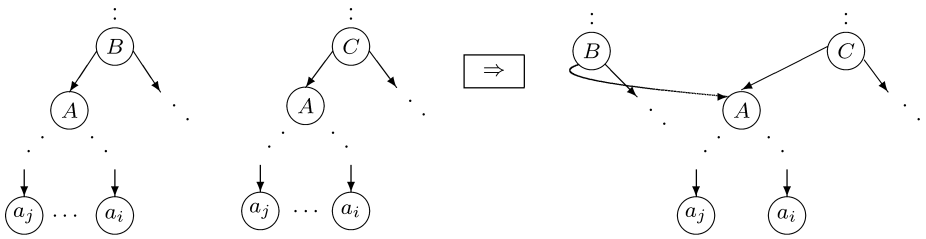
The algorithms that we have considered so far are 'recognisers' rather than parsers in the sense that they do not output a derivation of the string. If the

semantics of the source language are defined with respect to the grammar used to specify the syntax, then it is likely that the derivation of a sentence will be helpful in the semantic analysis stage. The algorithms that we have discussed have been designed with derivation tree construction in mind, so it is relatively straightforward to add features to generate derivation trees. However, there are issues of efficiency of representation which need to be considered in the case where there is ambiguity in the underlying grammar.

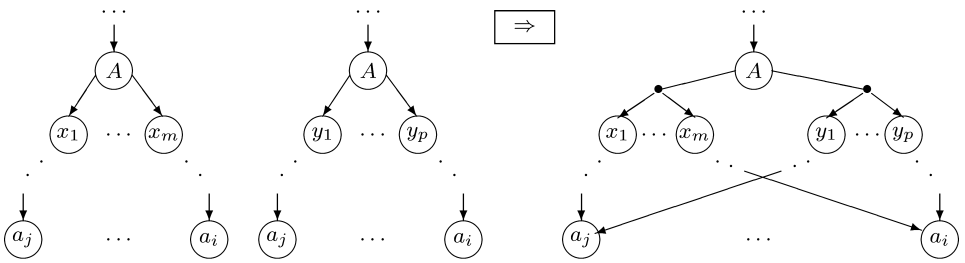
There are several aspects of efficiency of representation which are in tension with each other: (i) the compactness of the final structure, (ii) the efficiency of the algorithm which constructs the structure, and (iii) the efficiency with which the structure can be used in subsequent processes. As the main focus of this article is our improvement of Tomita's original algorithm, we shall not embark on a detailed study of the advantages and disadvantages of various derivation tree representations. We shall take as our basis the SPPFs constructed by Tomita and Rekers and compare these in terms of aspects (i) and (ii).

6.1 Shared Packed Parse Forests (SPPF)

When a grammar contains ambiguity, there can be more than one derivation tree corresponding to a given sentence. We can reduce the amount of space required to represent multiple trees by merging them, sharing nodes which have the same tree below them, and combining nodes which correspond to different derivations of the same substring. In general, given a forest of derivation trees for a string $a_1 \ldots a_n$, if two trees contain the same subtree for a substring $a_j \ldots a_i$, say, then this subtree can be shared.



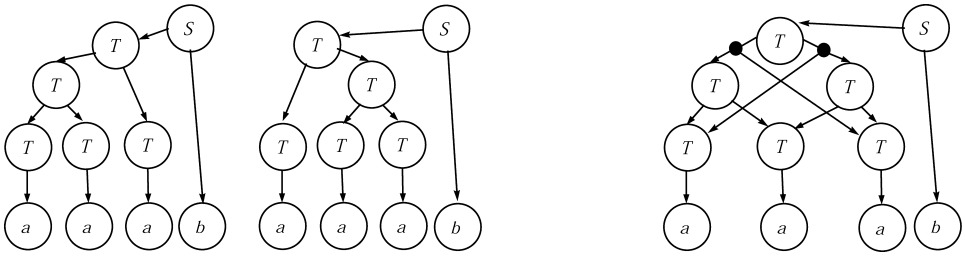
If two nodes labelled say, A, have different subtrees which derive the same substring, say, $a_j \ldots a_i$, then the two nodes labelled A can be packed together and the subtrees can be added as alternates under the packed node. The effect of this is that the contexts around the packed nodes A are shared.



For example, there are two derivation trees (on the below left) for the string $aaab$ in the grammar

$$S \quad ::= \quad Tb \qquad\qquad T \quad ::= \quad TT \mid a.$$

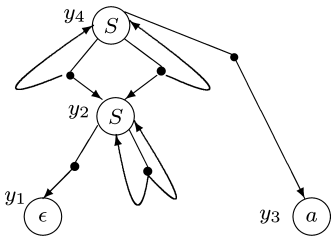
Packing these trees give the SPPF on the right.



Grammars which contain cycles generate sentences which can have infinitely many derivation trees. We represent these in a finite packed shared parse forest by introducing cycles to the graph. For example, all the derivations of the sentence a generated by the grammar Γ_3

$$S \quad ::= \quad SS \mid a \mid \epsilon \qquad\qquad (\Gamma_3)$$

can be represented by the graph



.

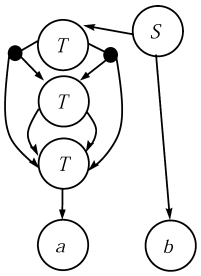
We shall refer to a directed graph obtained by merging and packing the nodes in the fashion just described as a shared packed parse forest (SPPF). Derivation trees can be recovered from SPPFs by unwinding the cycles the required number of times, selecting an alternate from below each packed node, and deleting the other alternates.

It is a property of our SPPFs that if one child of a node is a packed node, then all the children of that node are packed nodes, and that packed nodes do not themselves have children that are packed nodes. We think of nodes as having families of children, each family corresponding to a different derivation of the same portion of the input string. Formally, we define the *family* of a node to be a set of sequences of nodes. If the node does not have packed nodes as children, then its family consists of a single sequence of nodes, its children in left-to-right order. If the node has children that are packed nodes, then its family is the set of sequences which belong to the families of its (packed node) children. Thus,

y_4 in the preceding SPPF has family

$$\{(y_4, y_2), (y_2, y_4), (y_3)\}.$$

It is possible to obtain more compact representations of the derivation trees. For example, leaf nodes which correspond to the same terminal in different positions of the input string could be merged. So, we could use the following structure to represent the derivation trees of *aaab*, previously described.

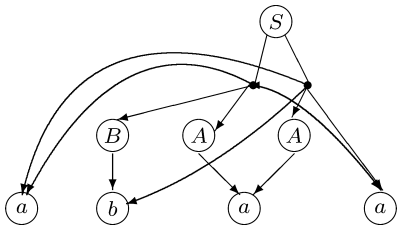


Two nodes u, v can be packed if they have the same label and the yields (string of leaf labels read from left-to-right) of the subtrees of u and v are the same. Ultimately, if we are to pack all possible nodes, then we need to either store or search and construct the yield of every subtree. This gives high importance to aspect (i), the compactness of the tree, at the great expense of aspect (ii), the efficiency of the construction. Tomita's SPPF construction process heavily compromises (i) for the sake of (ii): His process involves almost no tree searching but fails to pack many of the potentially packable nodes. Rekers' algorithm produces more compact SPPFs at the expense of the efficiency of the algorithm. His algorithm does not produce SPPFs in which terminal nodes are shared, so the SPPF is considerably less compact than it could be, but the algorithm is much more efficient than one that creates the more compact SPPFs. Both algorithms have the property that if the grammar is not ambiguous, then the SPPF produced is the standard derivation tree.

6.2 Rekers' Algorithm

A basic property of the graph structured stacks constructed by Tomita's algorithm is that there is a one-to-one correspondence between the edges in the GSS constructed by the REDUCER and the interior nodes of the SPPF. Briefly, SHIFTER(i) creates a node, u, in the SPPF labelled a_i and each of the edges constructed in the GSS by SHIFTER(i) is labelled u. When REDUCER(i) processes a reduction, it records the labels on the edges down which the reduction is applied. Then when REDUCER(i) creates an edge in the GSS, it also creates a new node in the SPPF labelled with the lefthand side of the reduction and with children the nodes of which label the traversed edges. If a new edge is not created, then an additional set of children is added to the label of the existing edge. Using Tomita's SPPF building

algorithm with grammar Γ_1 from Section 3.1 and string $abaa$ results in the GSS



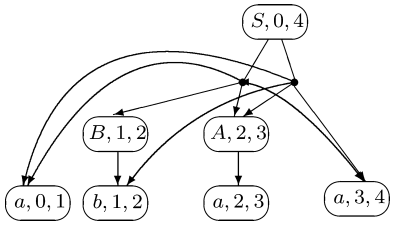
.

However, the two nodes labelled A could have been merged, resulting in a more compact SPPF.

The use of cycles in the GSS to deal with infinitely many derivations was introduced by Farshi [Nozohoor-Farshi 1991] in his correction to Tomita's algorithm. Rekers used Farshi's underlying recognition algorithm and introduced the actions to turn Farshi's algorithm into a parser, generating the SPPFs with cycles described in the previous section. In Rekers' algorithm it is also the case that each edge in the GSS is labelled with a node in the SPPF which is being concurrently constructed, but two different edges can be labelled with the same SPPF node. The nodes in the SPPF are each labelled with a triple (x, j, i), where x is a grammar symbol or ϵ and $0 \leq j \leq i$ are integers. If $x = \epsilon$, then $i = j$, otherwise, if $a_1 \ldots a_d$ is the input string, then $x \stackrel{*}{\Rightarrow} a_j \ldots a_i$.

In detail, SHIFTER(i) constructs a new SPPF leaf node u labelled $(a_{i+1}, i, i+1)$ and then the node u is used to label each edge created by SHIFTER(i). The other nodes and edges are added to the SPPF by the REDUCER. Suppose that REDUCER(i) is processing a GSS node v whose label is k and $(A ::= \alpha\cdot, a_{i+1}) \in k$. For each path of length $m = |\alpha|$ from v to w whose label is, say, l, we record the labels $u_1, \ldots, u_m$ of the edges along this path. If $m = 0$, then our target triple is (A, i, i) otherwise our target triple is (A, j, i), where node u_1 is labelled with a triple (x_1, j, q). Then, if ph is the entry in row l, column A of the input parse table, we find or create a node z in U_{i+1} labelled h. If there is no edge from z to w, then one is created and we search the SPPF constructed so far for a node, u, labelled with our target triple. If no such node exists, then one is created. We then check to see whether u has the nodes $u_1, \ldots, u_m$ as a set of children. If not, then this sequence is added as a set of children of v. The new edge from z to w is then labelled u.

If we use Rekers' SPPF building algorithm with Γ_1 and the string $abaa$, we get the following SPPF.



6.3 SPPF Construction with the RNGLR Algorithm

Although there are additional construction costs involved in using Rekers' rather than Tomita's approach to SPPF generation, we use what is essentially Rekers' approach because it produces more compact graphs. We have to modify the approach to deal with right nullable reductions and we maintain a set $\mathcal{N}$ of SPPF nodes constructed during the current step i of the main for loop. The latter modification reduces the set of nodes which need to be searched for an existing node and means that we only need to label SPPF nodes with the grammar symbol and one integer.

For a grammar with right nullable rules, the full SPPF will not always be directly constructed because the nullable righthand ends of reductions are short-circuited. Our approach is to have preconstructed SPPFs for the nullable righthand ends of rules and to add these in the appropriate places once the GSS construction has been completed. For a nullable string γ we call the SPPF of all the derivations $\gamma \stackrel{*}{\Rightarrow} \epsilon$ the ϵ-SPPF for γ. We might think that we should provide ϵ-SPPFs for all nullable strings γ such that there is a grammar rule of the form $A ::= \alpha\gamma$. It turns out that in the case where γ is the whole righthand side, that is, $\alpha = \epsilon$, this leads to SPPFs which are not as compact as they should be. So we provide ϵ-SPPFs for all the nullable nonterminals and for nullable strings γ such that $|\gamma| > 1$ and there is a grammar rule of the form $A ::= \alpha\gamma$, where $\alpha \neq \epsilon$. We call these strings γ the *required nullable parts*. For rules of the form $A ::= \gamma$ we use the ϵ-SPPF for A.

Given a grammar, we index, starting at 1, the nonterminals X such that $X \stackrel{*}{\Rightarrow} \epsilon$ and the required nullable parts γ. We let I be this indexing function and we define $I(\epsilon) = 0$. We label as $x_{I(\omega)}$ the root node of the ϵ-SPPF for ω, and x_0 is a node labelled ϵ. In the RN table a reduction $(A ::= \alpha \cdot \gamma, a)$ is written $r(A, m, f)$, where $|\alpha| = m$ and $f = I(\gamma)$ if $m \neq 0$ and $f = I(A)$ if $m = 0$.

In the following algorithm, the set $\mathcal{R}$ of 'pending reductions' is a set of quintuples (v, X, m, f, y), where v is the GSS node from which the reduction is to be applied, X is the lefthand side of the reduction rule, m is the length of the righthand side of the reduction rule, f is the index of the required nullable part at the righthand end of the reduction ($f = 0$ if the reduction is not right nullable), and y is the SPPF node labels the first edge of the path down which the reduction is applied (if $m = 0$, then $y = \epsilon$). As for the RNGLR recogniser, $\mathcal{Q}$ consists of pairs (v, k), where v is a GSS node to which a shift is to be applied and k is the label of the GSS node which must created as the parent of v.

The RNGLR Parser

Input: an RN table $\mathcal{T}$, an input string $a_1 \ldots a_d$, the ϵ-SPPFs for each nullable nonterminal and required nullable part ω, and the root nodes $x_{I(\omega)}$ of these ϵ-SPPFs.

```
PARSER {
 if d = 0 { if acc ∈ T(0, $) {
                report success and output the SPPF whose root node is x_I(S) }
            else report failure }
 else {
  create a node v_0 labelled with the start state 0 of the DFA
  set U_0 = {v_0}, R = ∅, Q = ∅, a_{d+1} = $, U_1 = ∅, ..., U_d = ∅
```

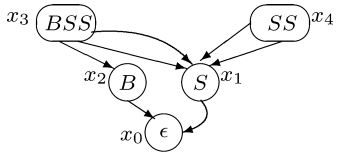
```
  if pk ∈ T(0, a_1) add (v_0, k) to Q
  forall r(X, 0, f) ∈ T(0, a_1) add (v_0, X, 0, f, ε) to R
  for i = 0 to d  while U_i ≠ ∅ {
   N = ∅ (contains the SPPF nodes constructed at this step)
   while R ≠ ∅ do REDUCER(i)
   do SHIFTER(i)  }
  if the DFA accepting state l labels an element t ∈ U_d {
   let root-node be the SPPF node that labels the edge (t, v_0) in the GSS
   remove nodes in the SPPF not reachable from root-node and report success
  else report failure } } }

REDUCER(i) {
 remove (v, X, m, f, y) from R
 find the set χ of paths of length (m − 1) (or length 0 if m = 0) from v
 if m ≠ 0 let y_m = y
 for each path in χ  do {
  let y_{m−1}, ..., y_1 be the edge labels on the path and let u be
                the final node on the path (if m = 1 then y = y_1)
  let k be the label of u and let pl ∈ T(k, X)
  if m = 0 let z = x_f
  else { suppose that u ∈ U_c
         if there is no node z in N labelled (X, c)  create one and add z to N }
  if there is an element w ∈ U_i with label l  {
    if there is not an edge from (w, u)  {
    create an edge (w, u) labelled z
    if m ≠ 0 { forall r(B, t, f) ∈ T(l, a_{i+1}) where t ≠ 0,
                                add (u, B, t, f, z) to R } } }
  else { create a new GSS node w labelled l and an edge (w, u) labelled z
         if ph ∈ T(l, a_{i+1}) add (w, h) to Q
         forall reductions r(B, 0, f) ∈ T(l, a_{i+1}) add (w, B, 0, f, ε) to R
         if m ≠ 0 {  forall r(B, t, f) ∈ T(l, a_{i+1}) where t ≠ 0
                                add (u, B, t, f, z) to R } }
  if m ≠ 0 ADD_CHILDREN(z, y_1, ..., y_m, f) } }

SHIFTER(i) {
 if i ≠ d {
  Q' = ∅  (a temporary set to hold new shifts)
  create a new SPPF node z labelled (a_{i+1}, i)
  while Q ≠ ∅  do {
   remove an element (v, k) from Q
   if there is w ∈ U_{i+1} with label k  {
    create an edge (w, v) labelled z
    forall r(B, t, f) ∈ T(k, a_{i+2}) where t ≠ 0 add (v, B, t, f, z) to R }
   else  {
         create a new node, w ∈ U_{i+1},  labelled k and an edge (w, v) labelled z
         if ph ∈ T(k, a_{i+2}) add (w, h) to Q'
         forall r(B, t, f) ∈ T(k, a_{i+2}) where t ≠ 0 add (v, B, t, f, z) to R
         forall r(B, 0, f) ∈ T(k, a_{i+2}) add (w, B, 0, f, ε) to R } }
  copy Q' into Q } }

ADD_CHILDREN(y, y_1, ..., y_m, f) {
 if f = 0 let Λ = (y_1, ..., y_m)  else let Λ = (y_1, ..., y_m, x_f)
 if Λ does not already belong to the family of y {
   if y has no children then add edges from y to each node in Λ
   else
     if the children of y are not packed nodes {
       create a new packed node z and a tree edge from y to z
```

add tree edges from z to all the other children of y
remove all tree edges from y apart from the one to z }
create a new packed node t and a new tree edge from y to t
create new edges from t to each node in Λ } }

We now give an example using the preceding algorithm with the grammar Γ_4

$$S ::= bBSS \mid a \mid \epsilon \qquad B ::= \epsilon \qquad (\Gamma_4)$$

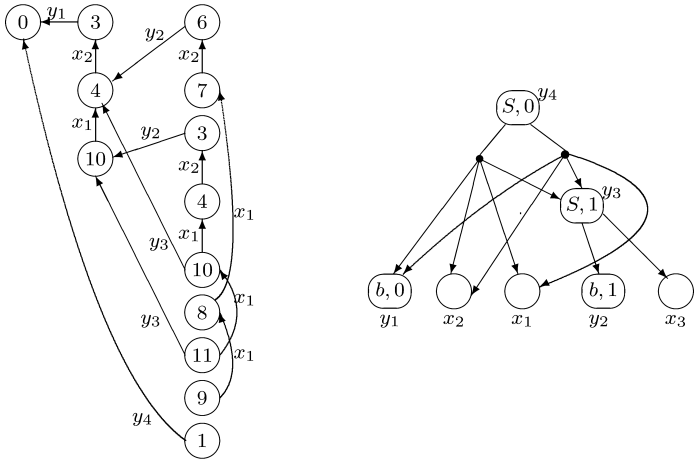
and input string bb. The nullable nonterminals are S and B, and the required nullable parts are BSS and SS. We define $I(S) = 1$, $I(B) = 2$, $I(BSS) = 3$, and $I(SS) = 4$. The ϵ-SPPFs are



and the RN table for Γ_4 is

	$	a	b	S	B
0	r(S,0,1)/acc	p2	p3	p1	
1	acc				
2	r(S,1,0)				
3	r(S,1,3)/r(B,0,2)	r(B,0,2)	r(B,0,2)		p4
4	r(S,2,4)/r(S,0,1)	p5/r(S,0,1)	p6/r(S,0,1)	p10	
5	r(S,1,0)	r(S,1,0)	r(S,1,0)		
6	r(S,1,3)/r(B,0,2)	r(S,1,3)/r(B,0,2)	r(S,1,3)/r(B,0,2)		p7
7	r(S,2,4)/r(S,0,1)	p5/r(S,2,4)/r(S,0,1)	p6/r(S,2,4)/r(S,0,1)	p8	
8	r(S,3,1)/r(S,0,1)	p5/r(S,3,1)/r(S,0,1)	p6/r(S,3,1)/r(S,0,1)	p9	
9	r(S,4,0)	r(S,4,0)	r(S,4,0)		
10	r(S,3,1)/r(S,0,1)	p2	p3	p11	
11	r(S,4,0)				

The RNGLR parser generates the following GSS and SPPF



7. THE EFFICIENCY OF THE RNGLR ALGORITHM

We conclude this article with some remarks and experimental results illustrating the performance of the RNGLR algorithm.

On first inspection we might expect the RNGLR algorithm to be less efficient than Tomita's because, in general, there will be significantly more conflicts in the underlying parse table and hence, perhaps more potential execution paths to be investigated. However, it is essentially a consequence of our proof of the correctness of the algorithm that the GSS constructed using our method is identical to that constructed using Algorithm 1e and an LR(1) table in the cases where the latter algorithm works correctly. Although there are more conflicts in the RN table, these conflicts result only from moving the point at which a reduction is applied. These reductions would have been applied anyway after some 'shifting' of ϵ-matching nonterminals onto the stack, thus, they do not cause any additional edges to be added to the GSS. Furthermore, the GSS construction process is more efficient in our case in the sense that the amount of graph traversal needed is less when the grammar contains right nullable rules. This is because reductions via rules of the form $A ::= \alpha\beta$, where $\beta \stackrel{*}{\Rightarrow} \epsilon$ are applied down paths of length $|\alpha|$ rather than paths of length $|\alpha\beta|$.

The RNGLR parser has been implemented alongside Algorithm 1e and Farshi's algorithm so as to evaluate the strength of our claims concerning greater efficiency. We have compared the algorithms using grammars for ANSI-C, ISO7085 Pascal, IBM VS-Cobol, and the right recursive grammar Γ_5

$$S ::= Ta \qquad T ::= aT \mid \epsilon. \tag{Γ_5}$$

The last grammar has been included because it triggers $O(n^3)$ searching costs in Farshi's algorithm compared to the $O(n^2)$ costs of Algorithm 1e and the RNGLR algorithm.

We measure the statistics of GSS searching for the algorithms in terms of *edge visits*, that is, the total number of GSS edges retraced while performing reductions during the parse. Table I shows these costs along with the size (in terms of the number of edges) of the GSS and SPPF constructed: Building these structures is part of the runtime cost which is not otherwise shown in the edge visit counts. The original results comparing Algorithm 1e and the RNLGR algorithm have been reported in Johnstone and Scott [2002], and later results including comparisons with Farshi's algorithm have been reported in Johnstone et al. [2004a], where the algorithms are also compared with Earley's algorithm. The results shown in Table I for Farshi's algorithm are for LR(1) tables, except for the Cobol grammar where an SLR(1) table was used. The results for Algorithm 1e are for RN tables.

The relatively high numbers of edge visits performed by Algorithm 1e arise because the ACTOR carries out additional visits when collecting the reductions when a node is initially processed. This shows that although Algorithm 1e can be correctly used with an RN table, the modifications introduced in the RNGLR result in a significant improvement.

The RNGLR algorithm does not work with LR tables, and as we have seen, Algorithm 1e inherently requires more edge visits. Thus, to compare the RNGLR

Table I. Runtime Costs of Farshi, RNGLR, and Tomita 1e Algorithms

Grammar	Input Length	Edge visits Farshi	Edge visits RNGLR	Edge visits Algorithm 1e	GSS Edges	SPPF Edges
ISO Pascal	4,425	22,459	5,572	23,842	21,135	18,147
ANSI-C	4,291	30,484	4,450	28,461	28,477	28,761
COBOL	2,198	38,984	3,581	13,872	13,167	12,204
Γ_5	100	176,454	4,852	4,960	5,251	9,803

Table II. Algorithm 1e Edge Visits Using LR Tables

Grammar	Input Length	RNGLR RN Table	Mod Alg1e LR Table	Alg1e LR Table
ISO Pascal	4,425	5,572	8,087	21,049
ANSI-C	4,291	4,450	4,450	28,461
COBOL	2,198	3,581	4,442	13,039
Γ_5	100	4,852	4,951	5,052

algorithm with Tomita's algorithm on LR tables we have implemented a modified version of Algorithm 1e in which the ACTOR is removed (this algorithm is essentially the RNGLR algorithm with the tests for $m \neq 0$ removed from the REDUCER). The modified Algorithm 1e will usually run correctly on LR tables for grammars which do not contain hidden right recursion, and indeed, it runs correctly on our examples. Table II shows the cost of running the modified Algorithm 1e using LR tables and compares the results with the RNGLR algorithm. The ANSI-C grammar includes no nullable rules, so we observe no difference in performance. In all other cases, as expected, the RNGLR performs fewer, not more, edge visits.

7.1 The RNGLR Algorithm on LR(1) Grammars

In this section, we briefly discuss the behaviour of the RNGLR algorithm on LR(1) grammars. A full discussion of the special properties of RN tables for LR(1) grammars can be found in Scott and Johnstone [2004].

It is possible for the RN table of an LR(1) grammar to contain multiple entries in the form of reduce-reduce conflicts (shift-reduce conflicts cannot occur if the grammar is LR(1), see Scott and Johnstone [2004].) Thus, it is not immediately clear that the RNGLR algorithm behaves efficiently on LR(1) grammars. The conflicts arise because an LR(1) DFA state which contains an item $(A ::= \alpha \cdot \beta, a)$, where $\beta \neq \epsilon$ and $\beta \stackrel{*}{\Rightarrow} \epsilon$ will also contain an item of the form $(C ::= \cdot, a)$. In this case the RNGLR algorithm will perform the reductions associated with both $A ::= \alpha \cdot \beta$ and $C ::= \cdot$. However, the special treatment by the REDUCER in the RNGLR algorithm of reductions of length 0 (the case $m = 0$ in the algorithm) ensures that reductions are not applied down paths whose first edge was created by a length 0 reduction. Thus, in the aforementioned case the reduction $A ::= \alpha\beta\cdot$, which is applied by the LR(1) parser, will not be applied by the RNGLR algorithm.

If we were to get the RNGLR algorithm and the standard LR(1) parser to print out a list of the reductions they apply, we would see that the LR(1) parser applies at least as many reductions as the RNGLR algorithm in the case where

the underlying grammar is LR(1). (In the case of hidden right recursion certain zero length reductions which are applied several times by the LR(1) parser are only applied once by the RNGLR algorithm because the GSS keeps a 'history' of stack activity and nodes can be reused.) For example, consider the grammar Γ_2

$$S ::= aSA \mid \epsilon \qquad A ::= \epsilon$$

whose LR(1) DFA is given in Section 3.4 and whose RN table is given in Section 4.1. On input aa the standard deterministic LR parser will perform the reductions

$$S ::= \cdot, \quad A ::= \cdot, \quad S ::= aSA\cdot, \quad A ::= \cdot, \quad S ::= aSA\cdot$$

while the RNGLR algorithm applies the reductions

$$S ::= \cdot, \quad S ::= a \cdot SA, \quad A ::= \cdot, \quad S ::= aS \cdot A.$$

(the reduction $A ::= \cdot$ is only applied once by the RNGLR algorithm because the corresponding GSS node is reused).

Each RN reduction $A ::= \alpha \cdot \beta$ is applied only down a shorter portion of the path down which the LR(1) parser applies the reduction $A ::= \alpha\beta\cdot$. Thus, we have that the RNGLR algorithm is linear on LR(1) grammars. In fact, the difference between the RNGLR algorithm and Tomita's algorithm on an LR(1) grammar is that the RNGLR algorithm performs marginally less searching because the reduction $A ::= \alpha \cdot \beta$ traces back down a shorter path than the reduction $A ::= \alpha\beta\cdot$.

Even though the RNGLR algorithm is linear on LR(1) grammars, the presence of conflicts in the RN table for an LR(1) grammar is uncomfortable because the standard deterministic LR(1) parsing algorithm cannot be used directly with such tables. Returning to the preceding example we notice that, although the RNGLR algorithm applies the reduction $A ::= \cdot$ and creates a corresponding node and edge in the GSS, this node is never used, so no reductions are applied down it. This leads us to consider whether the reduction needs to be applied at all or whether it can be removed from the RN table. The answer is that both reductions $A ::= \cdot$ and $S ::= \cdot$ can be removed from the RN table without affecting its correctness. There are more complicated examples than this, where it is less clear which reductions can safely be removed from an RN table, but it turns out that for an LR(1) grammar it is always possible to safely remove all the conflicts from the RN table.

In Scott and Johnstone [2004] we analyse RN tables and derive precise conditions under which reduce-reduce conflicts can be removed without compromising the correctness of a parser which uses the table. This results in what we call a *resolved RN table*. It is shown in Scott and Johnstone [2004] that if the grammar is LR(1), the resolved RN table contains no conflicts and can thus be used with the standard LR(1) parsing algorithm.

8. CONCLUSIONS

The development of Tomita-style algorithms is an interesting case study in the development of ideas. To summarise some of the points noted in Section 1.4:

Tomita originally presented in tutorial manner a family of five algorithms which we call Algorithms 0 through 4: Algorithms 0 is a rendition of the deterministic LR parser using a GSS instead of a stack and Algorithm 1 is a general parser for ϵ-free grammars. Tomita developed a rather baroque scheme for handling ϵ-reductions through the use of subfrontiers: As well as being complex, this algorithm does not terminate on grammars with hidden left recursion. Farshi subsequently presented an alternative algorithm which handles ϵ-rules simply by retrying all possible reductions on the current frontier. Although easier to understand than Tomita's ϵ-grammar algorithm, Farshi's algorithm is much less efficient. Nevertheless, it has been the GLR algorithm of choice to date.

This article descibes a new algorithm which takes as its starting point Tomita's ϵ-free algorithm (Algorithm 1) and which uses a new formulation of Knuth's reductions: In our automata and tables a reduction may be a right nulled (RN) reduction.

In detail, our modifications to the Tomita algorithm are twofold: (i) the table is changed to include right nulled reductions, and (ii) the parsing algorithm is modified to save some searching. The speedup arises from the modification to the parsing algorithm which can be made because an RN table is used. These changes are not applicable to Farshi and Tomita's algorithms because they do not use RN tables.

We have given a formal proof of correctness of the RNGLR algorithm and shown that it is more efficient than other GLR variants both on simple grammars designed to display the effects and on real programming language grammars used with real programs. We note in passing that McPeak's paper on Elkhound [McPeak and Necula 2004] displays parse times that are only around 10% greater than Bison when parsing LALR grammars, which shows that even the Farshi algorithm used there is competetive with deterministic parsers when used with deterministic grammars.

This article reports on work which has been carried out over a period of time. Some work-in-progress results have been presented in conference articles. In particular, an early version of the RNGLR algorithm was presented in Johnstone and Scott [2002] (where it was called reduction modified LR parsing). The performance of the Farshi and RNGLR algorithms is compared to Aycock and Horspool reduction incorporated (RI) parsers in Johnstone et al. [2004a] and Johnstone et al. [2004b] (the RI algorithm is introduced in Johnstone and Scott [2003]). In Scott and Johnstone [2004] we explore the possibilities for completely removing some of the reductions from an RN table. It turns out that the conditions under which this may be done without affecting the correctness of the parser are subtle.

ACKNOWLEDGMENTS

We are grateful to Ralf Lämmel for publishing his reverse engineered IBM VS-COBOL grammar (at `http://www.cs.vu.nl/grammars/vs-cobol-ii/`); and also to the anonymous referees for their helpful comments and suggestions.

REFERENCES

AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. 1975. Deterministic parsing of ambiguous grammars. *Commun. ACM 18*, 8 (Aug.).

AHO, A. V., SETHI, R., AND ULLMAN., J. D. 1986. *Compilers: Principles Techniques and Tools*. Addison-Wesley, Reading, Mass.

AHO, A. V. AND ULLMAN, J. D. 1972. *The Theory of Parsing, Translation and Compiling*. Series in Automatic Computation, vol. 1—Parsing. Prentice-Hall, Upper Saddle River, N. J.

AYCOCK, J. AND HORSPOOL, N. 1999. Faster generalised LR parsing. In *Proceedings of the Compiler Construction, 8th International Conference CC'99*. LNCS, vol. 1575. Springer, New York 32–46.

BILLOT, S. AND LANG, B. 1989. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Conference on Association for Computational Linguistics* 143–151.

BREUER, P. T. AND BOWEN, J. P. 1995. A PREttier compiler-compiler: Generating higher-order parsers in C. *Softw. Pract. Exper. 25*, 11 (Nov.), 1263–1297.

BUNT, H. AND TOMITA, M., eds. 1991. *Recent Advances in Parsing Technology*. Text speech and language technology, vol. 1. Kluwer Academic, Hingham, Mass.

DEREMER, F. L. 1969. Practical translators for LR(k) languages. Ph.D. thesis, Massachussetts Institute of Technology.

DEREMER, F. L. 1971. Simple LR(k) grammars. *Commun. ACM 14*, 7 (July), 453–460.

DODD, C. AND MASLOV, V. 1995. `http://compilers.iecc.com/comparch/article/95-03-044`.

EARLEY, J. 1970. An efficient context-free parsing algorithm. *Commun. ACM 13*, 2 (Feb.), 94–102.

EGGERT, P. 2003. `http://compilers.iecc.com/comparch/article/03-01-005`.

GRAHAM, S. L. AND HARRISON, M. A. 1976. Parsing of general context-free languages. *Adv. Comput. 14*, 77–185.

GRUNE, D. AND JACOBS, C. 1990. *Parsing Techniques: A Practial Guide*. Ellis Horwood, Chichester, England. `http://www.cs.vu.nl/~dick/PTAPG.html`.

HANSON, D. R. AND PROEBSTING, T. A. 2003. A research C# compiler. Tech. Rep. MSR-TR-2003-32, Microsoft Research, Redmond, WA.

HAYS, D. 1967. *Introductions to Computational Linguistics*. Elsevier, New York.

IRONS, E. 1961. A syntax directed compiler for Algol 60. *Commun. ACM 4*, 1, 51–55.

JOHNSTONE, A. AND SCOTT, E. 1998. Generalized recursive descent parsing and follow determinism. In *Proceedings of the 7th International Conference on Compiler Construction (CC'98),* LNCS, vol.1383. K. Koskimies, ed. Springer, Berlin, 16–30.

JOHNSTONE, A. AND SCOTT, E. 2002. Generalized reduction modified LR parsing for domain specific language prototyping. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS02)*.

JOHNSTONE, A. AND SCOTT, E. 2003. Generalized regular parsers. In *Proceedings of the Compiler Construction, 12th International Conference, CC'03*, G. Hedin, ed. LNCS, vol. 2622. Springer Berlin, 232–246.

JOHNSTONE, A., SCOTT, E., AND ECONOMOPOULOS, G. 2004a. Generalized parsing: Some costs. In *Proceedings of the Compiler Construction, 13th International Conference, CC'04*, E. Duesterwald, ed. LNCS, vol. 2985. Springer Berlin, 89–103.

JOHNSTONE, A., SCOTT, E., AND ECONOMOPOULOS, G. 2004b. The grammar tool box: A case study comparing GLR parsing algorithms. In *Proceedings of the 4th Workshop on Language Descriptions, Tools and Applications LDTA2004*, G. Hedin and E. V. Wick, eds. Also in *Electronic Notes in Theoretical Computer Science*. Elsevier, New York.

KASAMI, T. 1965. An efficient recognition and syntax analysis algorithm for context-free languages. Tech. Rep. AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass.

KNUTH, D. E. 1965. On the translation of languages from left to right. *Inf. Control 8*, 6, 607–639.

LANG, B. 1974. Deterministic techniques for efficient nondeterministic parsers. In *Proceedings of the Automata, Lanugages and Programming 2nd Colloquium*. LNCS, vol. 14. Springer, Berlin, 255–269.

LEWIS II, P. M. AND STEARNS, R. E. 1968. Syntax directed transduction. *J. ACM 15*, 3, 465–488.

MCPEAK, S. AND NECULA, G. 2004. Elkhound: A fast, practical GLR parser generator. In *Proceedings of the Compiler Construction, 13th Internationl. Conference CC'04*, E. Duesterwald, ed. LNCS. Springer, Berlin.

NEDERHOF, M.-J. 1994. An optimal tabular parsing algorithm. In *Proceedings of the 32nd Meeting of the Association for Computational Linguistics*. 117–124.

NEDERHOF, M.-J. AND SARBO, J. J. 1996. Increasing the applicability of LR parsing. In *Recent Advances in Parsing Technology*, H.Bunt and M. Tomita, eds. Kluwer Academic, Amsterdam, the Netherlands, 35–57.

NEDERHOF, M.-J. AND SATTA, G. 1996. Efficient tabular LR parsing. In *Proceedings of the 34th Meeting of the Association for Computational Linguistics*. 239–246.

NEDERHOF, M.-J. AND SATTA, G. 2004. Tabular parsing. In *Formal Languages and Applications, Studies in Fuzziness and Soft Computing 148*, G. C.Martin-Vide, V.Mitrana, eds. Springer, New York, 529–549.

NOZOHOOR-FARSHI, R. 1991. GLR parsing for ϵ-grammars. In *Generalized LR Parsing*, M. Tomita, ed. Kluwer Academic, Amsterdam, the Netherlands, 60–75.

PARR, T. J. 1996. *Language Translation Using PCCTS and C++*. Automata Publishing.

REKERS, J. G. 1992. Parser generation for interactive environments. Ph.D. thesis, Universty of Amsterdam.

SAKAI, I. 1962. Syntax in universal translation. In *Proceedings of the 1961 International Conference on Machine Translation of Languages and Applied Language Analysis*. 593–608.

SCOTT, E. AND JOHNSTONE, A. 2004. Reducing nondeterminism in right nulled GLR parsers. *Acta Informatica 40*, 459–489.

SCOTT, E., JOHNSTONE, A., AND HUSSAIN, S. S. 2000. Tomita-Style generalized LR parsers. Updated version. Tech. Rep. TR-00-12, Royal Holloway, University of London Dec.

SHEIL, B. 1976. Observations on context-free parsing. In *Statistical Methods in Linguistics*. 71–109.

STROUSTRUP, B. 1994. *The Design and Evolution of C++*. Addison-Wesley, Reading, Mass.

TOMITA, M. 1986. *Efficient Parsing for Natural Language*. Kluwer Academic, Boston.

UNGER, S. 1968. A global parser for context-free phrase structure grammars. *Comm. ACM 11*, 4, 240–246.

VALIANT, L. 1975. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci. 10*, 308–315.

VAN DEN BRAND, M., HEERING, J., KLINT, P., AND OLIVIER, P. 2002. Compiling language definitions: The ASF+SDF compiler. *ACM Trans. Program. Languages Syst. 24*, 4, 334–368.

VISSER, E. 1997. Syntax definition for langauge prototyping. Ph.D. thesis, Universty of Amsterdam.

VISSER, E. 2004. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, C. et al, ed. LNCS, vol. 3016. Springer, Berlin, 216–238.

YOUNGER, D. H. 1967. Recognition of context-free languages in time n^3. *Inform. Control 10*, 2 (Feb.), 189–208.

Received March 2004; revised November 2004; accepted November 2004