International Journal of Foundations of Computer Science

GENERALIZED LR PARSING ALGORITHM FOR BOOLEAN GRAMMARS

ALEXANDER OKHOTIN*
Department of Mathematics, University of Turku, Turku FIN–20014, Finland
alexander.okhotin@utu.fi

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

The generalized LR parsing algorithm for context-free grammars is extended for the case of Boolean grammars, which are a generalization of the context-free grammars with logical connectives added to the formalism of rules. In addition to the standard LR operations, *Shift* and *Reduce*, the new algorithm uses a third operation called *Invalidate*, which reverses a previously made reduction. This operation makes the mathematical justification of the algorithm significantly different from its prototype. On the other hand, the changes in the implementation are not very substantial, and the algorithm still works in time $O(n^4)$.

Keywords: Boolean grammars, language equations, conjunctive grammars, parsing, LR, bottom-up, shift–reduce.

1. Introduction

The *generalized LR* parsing was introduced in 1986 by Tomita [18, 19] as a polynomial-time method of simulating nondeterminism in standard Knuth's LR [7]. Every time a deterministic LR parser is faced with a choice of actions to perform (to *shift* an input symbol or to *reduce* by one or another rule), a generalized LR parser performs both actions at the same time, storing all possible contents of an LR parser's stack in the form of a graph. Although the number of possible computations of a nondeterministic LR parser can exponentially depend on the length of the input, the compact graph-structured representation always contains $O(n)$ vertices and therefore fits in $O(n^2)$ memory. If carefully implemented, the algorithm is applicable to every context-free grammar, and its complexity can be bounded by a polynomial of a degree as low as cubic [6].

Initially, the algorithm was proposed with linguistic applications in view [18, 19], and in the recent years there has been a growing interest in the use of this method for

*Supported by the Academy of Finland under grant 206039.
Present address: Research Group on Mathematical Linguistics, Department of Romance Philology, Rovira i Virgili University, 1 Placa de la Imperial Tàrraco, Tarragona 43005, Spain.

software engineering [2]. In particular, efficient implementation techniques are being researched [3, 8], application-oriented parser generators are being implemented [8], and some extensions to the algorithm motivated by applications are considered [4].

The engineering approach to extending the applicability of LR parsers lies in equipping them with certain kludges to implement some behaviour that cannot be expressed or is inconvenient to express in the formalism of context-free grammars. The control over the added machinery is then given to the user in the form of additional instructions to the parser included together with a context-free grammar. For instance, Salomon and Cormack [15] thus implemented a very particular form of negation in Knuth's LR, while van der Brand et al. [4] extended their technique and obtained a form of negation in the framework of generalized LR parsing. This work makes apparent the insufficiency of a context-free grammar as a mathematical model to the respective engineering tasks.

The goal of the present paper is to apply the ideas of generalized LR parsing to construct a practically useful parsing algorithm for a theoretically defined family of formal grammars: namely, for *Boolean grammars* recently introduced by the author [13]. Boolean grammars are context-free grammars augmented with Boolean operations in the formalism of rules, which, for instance, allows one to specify negation of syntactical conditions in the most natural way. This increase in expressive power does not lead to a complexity blowup: the languages generated by Boolean grammars are contained in $DTIME(n^3) \cap DSPACE(n)$. Boolean grammars can specify many abstract non-context-free languages, such as $\{a^n b^n c^n | n \geqslant 0\}$, $\{ww \,|\, w \in \{a,b\}^*\}$ and $\{a^{2^n} \,|\, n \geqslant 0\}$, the latter being outside of the Boolean closure of the context-free languages. Another evidence of their expressive power is given by a fairly compact grammar for the set of well-formed programs in a simple model programming language [14], which is the first specification of any programming language by a formal grammar from a computationally feasible class. The generalized LR algorithm presented in this paper allows one to convert this particular grammar to a square-time correctness checker. The algorithm has been implemented in an ongoing parser generator project [11], and such a correctness checker has been successfully produced out of that Boolean grammar.

The new algorithm is partially based upon the author's earlier attempt to extend the applicability of the Generalized LR to conjunctive grammars [10]. *Conjunctive grammars* [9] are, to put it simply, context-free grammars with added conjunction, or, in other words, Boolean grammars without negation. Their expressive power is still quite greater than that of the context-free grammars. Generalized LR parsing can be extended to these grammars rather straightforwardly, and its worst-case complexity is still only cubic [10].

Extending the Generalized LR algorithm to Boolean grammars presents new challenges associated with the negation. The most interesting quality of the proposed algorithm is that the negation is implemented by removing arcs from the graph-structured stack. In terms of the theory of parsing schemata [17], this means that atomic items can be not only gained, but also lost, and regained back, etc. This is a clear departure from the paradigm of *parsing as deduction* [16]. Establishing

the correctness of a parsing algorithm that behaves in such an uncommon way is a new task to be solved. We shall see how this can be done.

After a short introduction to Boolean grammars given in Section 2, the LR parsing table for the new algorithm is defined in Section 3. In Section 4, a high-level description of the algorithm is given, which abstracts from some details of its implementation. The description is followed by an example (Section 5), which serves as an introduction to the proof of correctness. A negligible subclass of Boolean grammars, to which the algorithm is not applicable, is characterized in Section 6, and the correctness proof in Section 7 shows that the algorithm works correctly on every grammar not from this subclass. Grammars from the prohibited subclass can be used after some initial transformations. The earlier omitted implementation details are defined in Section 8: it is shown that the most obvious implementation can use exponential time, but a slightly different technique leads to an $O(n^4)$ complexity upper bound. An attempt to relate this unconventional parsing method to the theory of parsing schemata is made in Section 9. In the concluding Section 10, the contribution is summarized and some research directions are proposed.

2. Boolean grammars

Boolean grammars are context-free grammars augmented with propositional connectives. In addition to the implicit disjunction represented by multiple rules for a single nonterminal, which is the only logical operation expressible in context-free grammars, Boolean grammars include explicit conjunction and negation in the formalism of rules.

Definition 1 *A Boolean grammar [13] is a quadruple $G = (\Sigma, N, P, S)$, where Σ and N are disjoint finite nonempty sets of terminal and nonterminal symbols respectively; P is a finite set of rules of the form*

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \quad (m+n \geqslant 1,\ \alpha_i, \beta_i \in (\Sigma \cup N)^*), \qquad (1)$$

while $S \in N$ is the start symbol of the grammar. For each rule (1), the objects $A \to \alpha_i$ and $A \to \neg\beta_j$ (for all i, j) are called conjuncts, positive and negative respectively. A conjunct with unknown sign can be denoted $A \to \pm\gamma$, which means "$A \to \gamma$ or $A \to \neg\gamma$". Let $conjuncts(P)$ be the sets of all conjuncts, let $uconjuncts(P) = \{A \to \gamma \mid A \to \pm\gamma \in conjuncts(P)\}$.

A Boolean grammar is called a conjunctive grammar [9], if negation is never used, i.e., $n = 0$ for every rule (1); it degrades to a standard context-free grammar if neither negation nor conjunction are allowed, i.e., $m = 1$ and $n = 0$ for all rules. Assume, without loss of generality, that there is a positive conjunct in every rule, i.e., $m \geqslant 1$ in every rule (1). Let us adopt a commonly used short notation $A \to \varphi_1 \mid \ldots \mid \varphi_n$ for n rules $A \to \varphi_i$ of the form (1) for a single a nonterminal A.

Intuitively, a rule (1) can be read as *"if a string satisfies the syntactical conditions $\alpha_1, \ldots, \alpha_m$ and does not satisfy any of the syntactical conditions $\beta_1, \ldots, \beta_n$, then this string satisfies the condition represented by the nonterminal A".* This intuitive interpretation is not yet a formal definition, but this understanding is suf-

ficient to construct grammars. Following is an example of a Boolean grammar for a simple non-context-free language.

Example 1 *Let $\Sigma = \{a, b, c\}$ and $N = \{S, A, C, D, E\}$, and consider the following set of nine rules:*

$$S \to AD\&\neg EC$$
$$A \to aA \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$
$$D \to bDc \mid \varepsilon$$
$$E \to aEb \mid \varepsilon$$

This grammar generates the language $L = \{a^m b^n c^n \mid m, n \geqslant 0,\ m \neq n\}$.

Indeed, the nonterminals A, C, D and E have only context-free rules and hence are assumed to generate the languages a^*, c^*, $\{b^i c^i \mid i \geqslant 0\}$ and $\{a^i b^i \mid i \geqslant 0\}$, respectively, and then the rule for S specifies the language

$$a^* \cdot \{b^i c^i \mid i \geqslant 0\} \cap \overline{\{a^i b^i \mid i \geqslant 0\}) \cdot c^*} = \{a^m b^n c^n \mid m, n \geqslant 0,\ m \neq n\}$$

If the negation were omitted in the rule for S, the grammar would be conjunctive [9] and would generate the language $L' = \{a^n b^n c^n \mid n \geqslant 0\}$.

Though this common-sense interpretation is clear for "reasonably written" Boolean grammars, the use of negation can, in general, lead to logical contradictions, and for that reason the task of defining a mathematically sound formal semantics for Boolean grammars is far from being trivial. The existing definition of Boolean grammars [13] is based upon representing a grammar as a system of language equations with concatenation, union, intersection and complementation, similarly to the well-known characterization of the context-free grammars due to Ginsburg and Rice [5], which uses language equations with concatenation and union only.

Definition 2 *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar. The system of language equations associated with G is a system over Σ in variables $N = \{A_1, \ldots, A_n\}$, resolved with respect to these variables and containing the following equations:*

$$A_k = \bigcup_{A_k \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P} \left[\bigcap_{i=1}^{m} \alpha_i \cap \bigcap_{j=1}^{n} \overline{\beta_j} \right] \quad (\textit{for all } A_k \in N) \qquad (2)$$

A vector of languages $L = (L_1, \ldots, L_n)$ is a solution of this system if a substitution of L_k for A_k for all k turns each equation in (2) into an equality.

Example 2 *The grammar from Example 1 has the following associated system:*

$$S = AD \cap \overline{EC}$$
$$A = aA \cup \{\varepsilon\}$$
$$C = cC \cup \{\varepsilon\}$$
$$D = bDc \cup \{\varepsilon\}$$
$$E = aEb \cup \{\varepsilon\}$$

The unique solution of this system is $S = \{a^m b^n c^n \mid m, n \geqslant 0,\ m \neq n\}$, $A = a^*$, $C = c^*$, $D = \{b^i c^i \mid i \geqslant 0\}$, $E = \{a^i b^i \mid i \geqslant 0\}$).

In general, systems of language equations of the form (2) have a high expressive power and the associated undecidability results [12]. The class of languages represented by their unique solutions is exactly the class of recursive languages, and the way these languages are represented does not well correspond to the intuitive semantics of Boolean grammars defined above. However, a certain restriction upon these equations leads to a feasible semantics for Boolean grammars.

Definition 3 *A vector $L = (L_1, \ldots, L_n)$ is called a naturally reachable solution of (2) if for every finite substring-closed $M \subseteq \Sigma^*$ and for every string $u \notin M$ (such that all proper substrings of u are in M) every sequence of vectors of the form*

$$L^{(0)}, L^{(1)}, \ldots, L^{(i)}, \ldots \tag{3}$$

(where $L^{(0)} = (L_1 \cap M, \ldots, L_n \cap M)$ and every next vector $L^{(i+1)} \neq L^{(i)}$ in the sequence is obtained from the previous vector $L^{(i)}$ by substituting some j-th component with $\varphi_j(L^{(i)}) \cap (M \cup \{u\})$) converges to

$$(L_1 \cap (M \cup \{u\}), \ldots, L_n \cap (M \cup \{u\})) \tag{4}$$

in finitely many steps regardless of the choice of components at each step.

The unique solution of the system in Example 2 is naturally reachable. To illustrate Definition 3, let us show how the solution modulo $\{\varepsilon\}$ of that system is uniquely determined according to this definition, for $M = \varnothing$ and $u = \varepsilon$. The initial vector in the sequence (3) is $L^{(0)} = (\varnothing, \varnothing, \varnothing, \varnothing, \varnothing)$. At the first step one can choose any of the last four components, substitute $L^{(0)}$ into it and obtain $\{\varepsilon\}$ in all four cases. One cannot choose S at the first step, because $(AD \cap \overline{EC})(L^{(0)}) = \varnothing = S(L^{(0)})$. These four choices, as well as all possible continuations of the sequence, are shown in Figure 1. The majority of these sequences converge to the unique solution modulo $\{\varepsilon\}$, $L' = (\varnothing, \{\varepsilon\}, \{\varepsilon\}, \{\varepsilon\}, \{\varepsilon\})$, in four steps, and these sequences differ only in the order in which ε is added to A, C, D and E.

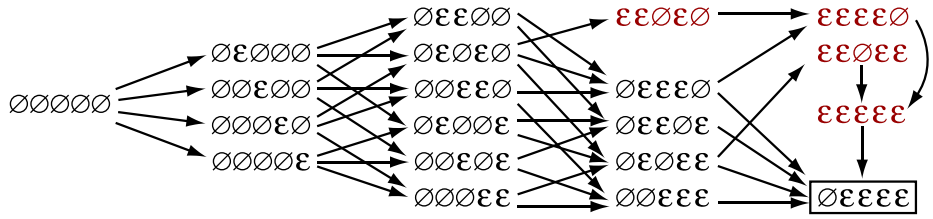

Figure 1: Computing the naturally reachable solution in Example 1, $M = \varnothing$, $u = \varepsilon$.

However, the sequence can proceed differently: if ε is added to A and D before E and C, such as in the vector $L^{(2)} = (\varnothing, \{\varepsilon\}, \varnothing, \{\varepsilon\}, \varnothing)$, then $\varepsilon \in (AD \cap \overline{EC})(L^{(2)})$, and it becomes possible to choose S at this step. The next vector will be $L^{(3)} = (\{\varepsilon\}, \{\varepsilon\}, \varnothing, \{\varepsilon\}, \varnothing)$, but, as one can see from Figure 1, the computation will eventually repair itself and converge to the unique solution modulo $\{\varepsilon\}$.

For all moduli larger than $\{\varepsilon\}$, the sequence (3) for this grammar converges in a much simpler way. Consider the extension from $M = \{\varepsilon\}$ to $M \cup \{a\}$: the

sequence starts from $L^{(0)} = (\varnothing, \{\varepsilon\}, \{\varepsilon\}, \{\varepsilon\}, \{\varepsilon\})$, only A can be chosen at the first step, which gives $L^{(1)} = (\varnothing, \{\varepsilon, a\}, \{\varepsilon\}, \{\varepsilon\}, \{\varepsilon\})$. At the second step, only S can be chosen, and the resulting vector $L^{(2)} = (\{a\}, \{\varepsilon, a\}, \{\varepsilon\}, \{\varepsilon\}, \{\varepsilon\})$ is the solution modulo $\{\varepsilon, a\}$.

Proceeding in this way, the language generated by the grammar is defined.

Definition 4 *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar and assume that the associated system of language equations has a naturally reachable solution; let $(L_1, \ldots, L_n)$ be this solution. The language $L_G(A_i)$ generated by every i-th nonterminal A is defined as L_i, while the language generated by the grammar is $L(G) = L_G(S)$.*

Despite the increased descriptive power, the theoretical upper bound for the parsing complexity for Boolean grammars is still $O(n^3)$ [13], the same as in the context-free case, which is obtained by an extension of the Cocke–Kasami-Younger algorithm. This algorithm uses cubic time for every language generated by a Boolean grammar and on every input, and it requires that the grammar is transformed to the following extension of Chomsky normal form [13]:

Definition 5 *A Boolean grammar $G = (\Sigma, N, P, S)$ is in the binary normal form if every rule in P is of the form*

$$A \to B_1 C_1 \& \ldots \& B_m C_m \& \neg D_1 E_1 \& \ldots \& \neg D_n E_n \& \neg\varepsilon \quad (m \geqslant 1, n \geqslant 0)$$
$$A \to a$$
$$S \to \varepsilon \quad \text{(only if } S \text{ does not appear in right-hand sides of rules)}$$

Though every Boolean grammar can be effectively transformed to this form [13], the transformation is more difficult than in the context-free case, and thus it is particularly important to have an algorithm that in most cases will not require initial transformation of a grammar. Also, for a practical use it is crucial that an algorithm uses less than worst-case time on large classes of "easy" grammars. These requirements are met by the context-free generalized LR [18], and we shall see that Boolean generalized LR algorithm constructed in this paper meets them as well.

3. The parsing table

Let us begin defining the generalized LR parsing algorithm for Boolean grammars from its parsing table. It is generally the same as in the deterministic context-free case [1, 7], but it still requires a new construction.

Each algorithm from the LR family is guided by a parsing table constructed with respect to a grammar. For every state from a finite set of states Q and for each lookahead string from $\Sigma^{\leqslant k}$ (where $\Sigma^{\leqslant k}$ is defined as $\{w \mid w \in \Sigma^*, |w| \leqslant k\}$), the parsing table provides the parser with the action to perform: whether to shift the next input symbol or to reduce by a certain rule. There exists a great variety of different table construction techniques for Knuth's deterministic algorithm, applicable to slightly different classes of grammars and yielding tables of different size. But already in the case of Tomita's nondeterministic algorithm, the difference

between these methods is not very essential. Let us adapt the simplest of them, SLR(1) [1], for the case of Boolean grammars.

The first step is to construct a deterministic finite automaton over the alphabet $\Sigma \cup N$, called the LR(0) automaton, which recognizes the bodies of grammar rules in the stack. In our case this step is the same as in the context-free case. While in the context-free case the states of the LR(0) automaton are sets of dotted rules, dotted unsigned conjuncts are used in the case of Boolean grammars:

Definition 6 *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar. $A \to \alpha \cdot \beta$ is called a dotted conjunct, if the grammar contains a conjunct $A \to \pm\alpha\beta$. Let* $\mathrm{dc}(P)$ *denote the (finite) set of all dotted conjuncts.*

Let the set of states be $Q = 2^{\mathrm{dc}(P)} \cup \{q_{acc}\}$. In order to define the initial state and the transitions between states, the functions *closure* and *goto* are used. They are defined as in the standard context-free LR theory [1], with the only difference that the objects they deal with are called conjuncts rather than rules.

For every set of dotted conjuncts X and for every $s \in \Sigma \cup N$, define

$$goto(X, s) = \{A \to \alpha s \cdot \beta \mid A \to \alpha \cdot s\beta \in X\}$$

Next, $closure(X)$ is defined as the least set of dotted conjuncts that contains X and satisfies the condition that for each $A \to \alpha \cdot B\gamma \in closure(X)$ (where $\alpha, \gamma \in (\Sigma \cup N)^*$, $B \in N$) and for each conjunct $B \to \pm\beta \in conjuncts(P)$ it holds that $B \to \cdot\beta \in closure(X)$.

Define the initial state of the automaton as

$$q_0 = closure\big(\{S \to \cdot\sigma \mid S \to \pm\sigma \in conjuncts(P)\}\big),$$

while the transition from a state $q \subseteq \mathrm{dc}(P)$ by a symbol $s \in \Sigma \cup N$ is defined as follows:

$$\delta(q, s) = closure(goto(q, s)).$$

If $closure(goto(q_0, S)) = \varnothing$, this transition is redefined as $\delta(q_0, S) = q_{acc}$. The state $\varnothing \subset \mathrm{dc}(P)$ is an error state and will be denoted by $-$. Note that, in the terminology of Aho, Sethi and Ullman [1], $\delta(q, a) = q'$ $(a \in \Sigma)$ is expressed as "$\textsc{Action}[q, a] = \text{Shift } q'$", while $\delta(q, A) = q'$ $(A \in N)$ means "$\textsc{Goto}(q, A) = q'$".

The finite automaton constructed so far recognizes the bodies of the conjuncts, providing the pertinent information in the states it computes. The other component of an LR automaton, the *reduction function*, decodes this information from the numbers of the states and reports which rules can be applied. Some additional terminology is needed to define this function.

For every string w, define

$$First_k(w) = \begin{cases} w, & \text{if } |w| \leqslant k \\ \text{first } k \text{ symbols of } w, & \text{if } |w| > k \end{cases}$$

This definition is extended to languages as $First_k(L) = \{First_k(w) \mid w \in L\}$.

In the case of the context-free SLR(k), the reduction function is constructed using the sets $\textsc{Follow}_k(A) \subseteq \Sigma^{\leqslant k}$ $(A \in N)$ that specify the possible continuations

of strings generated by a nonterminal A. This is formalized by context-free derivations: $u \in \textsc{Follow}_k(A)$ means that there exists a derivation $S \Longrightarrow^* xAy$, such that $First_k(y) = u$. The corresponding notion for the case of Boolean grammars is, in the absence of derivation, somewhat harder to define:

Definition 7 *Let us say that $u \in \Sigma^*$ follows $B \in N$ if there exists a number $\ell \geqslant 0$ and a sequence of conjuncts $A_0 \to \pm\alpha_1 A_1 \beta_1$, $A_1 \to \pm\alpha_2 A_2 \beta_2$, $\ldots$, $A_{\ell-1} \to \pm\alpha_\ell A_\ell \beta_\ell$, such that $A_0 = S$, $A_\ell = B$ and $u \in L_G(\beta_\ell \ldots \beta_1)$.*

Now, for every nonterminal $A \in N$, define $\textsc{First}_k(A) = First_k(L_G(A))$ and $\textsc{Follow}_k(A) = \{First_k(u) \mid u \text{ follows } A\}$. Already for conjunctive grammars there cannot exist an algorithm to compute the sets $\textsc{First}_k$ and $\textsc{Follow}_k$ precisely [10]: this is an easy consequence of the undecidability of the emptiness problem [9]. However, since the LR algorithm uses the lookahead information solely to eliminate some superfluous reductions, if the sets $\textsc{First}_k(A)$ and $\textsc{Follow}_k(A)$ are replaced by some of their supersets, the resulting LR parser will still work, though it will have to spend extra time doing some computations that will not influence the result. Following are the algorithms for constructing suitable supersets $\textsc{Pfirst}_k(A) \supseteq \textsc{First}_k(A)$ and $\textsc{Pfollow}_k(A) \supseteq \textsc{Follow}_k(A)$.

Algorithm 1 *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar compliant to the semantics of naturally reachable solution. Let $k > 0$. For all $s \in \Sigma \cup N$, compute the set $\textsc{Pfirst}_k(A)$, so that $u \in L_G(s)$ implies $First_k(u) \in \textsc{Pfirst}_k(s)$.*

let $\textsc{Pfirst}_k(A) = \varnothing$ for all $A \in N$;
let $\textsc{Pfirst}_k(a) = \{a\}$ for all $a \in \Sigma$;
while new strings can be added to $\langle \textsc{Pfirst}_k(A) \rangle_{A \in N}$
  for each $A \to s_{11} \ldots s_{1\ell_1} \& \ldots \& s_{m1} \ldots s_{m\ell_m} \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P$
    $\textsc{Pfirst}_k(A) = \textsc{Pfirst}_k(A) \cup$
      $\cup \bigcap_{i=1}^{m} First_k\big(\textsc{Pfirst}_k(s_{i1}) \cdot \ldots \cdot \textsc{Pfirst}_k(s_{i\ell_i})\big)$;

Note that the algorithm completely ignores negative conjuncts, effectively using a conjunctive grammar

$$G_+ = (\Sigma, N, \{A \to \alpha_1 \& \ldots \& \alpha_m \mid A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P\}, S)$$

instead of G. It is easy to see that $L_G(A) \subseteq L_{G_+}(A)$ for every $A \in N$. For the case of conjunctive grammars [10] it is known that if $u \in L_{G_+}(s)$, then $First_k(u) \in \textsc{Pfirst}_k(s)$, which immediately implies the correctness of Algorithm 1.

Algorithm 2 *For a given Boolean grammar G compliant to the semantics of naturally reachable solution and for $k > 0$, compute the sets $\textsc{Pfollow}_k(A)$ for all $A \in N$, so that if u follows A, then $First_k(u) \in \textsc{Pfollow}_k(A)$.*

let $\textsc{Pfollow}_k(S) = \{\varepsilon\}$;
let $\textsc{Pfollow}_k(A) = \varnothing$ for all $A \in N \setminus \{S\}$;
while new strings can be added to $\langle \textsc{Pfollow}_k(A) \rangle_{A \in N}$
  for each $B \to \beta \in uconjuncts(P)$
    for each factorization $\beta = \mu A \nu$, where $\mu, \nu \in V^*$ and $A \in N$
      $\textsc{Pfollow}_k(A) = \textsc{Pfollow}_k(A) \cup$
        $\cup First_k\big(\textsc{Pfirst}_k(\nu) \cdot \textsc{Pfollow}_k(B)\big)$;

Proof of correctness.. One needs to show that whenever u follows A, $First_k(u)$ is added to $\textsc{Pfollow}_k(A)$ at some point of the computation of the algorithm. The proof is by induction on the length ℓ of the sequence of conjuncts from Definition 7.

Basis $\ell = 0$. Then $A = S$ and $u = \varepsilon$, and ε is added to $\textsc{Pfollow}_k(S)$ by the first statement of the algorithm.

Induction step. Suppose there exists a sequence of conjuncts $A_i \to \pm\alpha_{i+1}A_{i+1}\beta_{i+1}$ $(0 \leqslant i < \ell)$, where $A_0 = S$, $A_\ell = B$ and $u \in L_G(\beta_\ell \dots \beta_1)$. Then there exists a factorization $u = xy$, such that $x \in L_G(\beta_\ell)$ and $y \in L_G(\beta_{\ell-1} \dots \beta_1)$. According to Algorithm 1, $First_k(x) \in \textsc{Pfirst}_k(\beta_\ell)$. By the induction hypothesis, $First_k(y)$ is added to $\textsc{Pfollow}_k(B)$ at some point. Then, at this point, $First_k(u) = First_k(First_k(x) \cdot First_k(y)) \in First_k(\textsc{Pfirst}_k(\beta_\ell) \cdot \textsc{Pfollow}_k(B))$, and hence $First_k(u)$ is added to $\textsc{Pfollow}_k(A)$ next time the conjunct $A_{\ell-1} \to \pm\alpha_\ell A_\ell \beta_\ell$ and the factorization $\alpha_\ell A_\ell \beta_\ell = \alpha_\ell \cdot A_\ell \cdot \beta_\ell$ are considered. □

The sets $\textsc{Pfollow}_k(A)$ are then used to define the *reduction function* $R : Q \times \Sigma^{\leqslant k} \to 2^{uconjuncts(P)}$, which tells the conjuncts recognized in a given state if the unread portion of the string starts with a given k-character string. In the SLR(k) table construction method, it is defined as follows:

$$R(q, u) = \{A \to \alpha \mid A \to \alpha\cdot \in q,\ u \in \textsc{Pfollow}_k(A)\}$$

for every $q \in Q$ and $u \in \Sigma^{\leqslant k}$. In the notation of Aho, Sethi and Ullman [1], $A \to \alpha \in R(q, u)$ means "$\textsc{Action}[q, u] = \text{Reduce } A \to \alpha$", assuming $A \to \alpha \in P$. As in the context-free case, the states from $Q \setminus \{-\}$ can be enumerated with consecutive numbers $0, 1, \dots, |Q| - 1$, where 0 refers to the state q_0.

This completes the construction of the SLR(k) table for the case of Boolean grammars, which is a straightforward generalization of the context-free and the conjunctive case. The differences of the parsing algorithm itself from its prototype are much more substantial.

4. The algorithm

The new LR parsing algorithm for Boolean grammars is a generalization of the corresponding algorithm for conjunctive grammars [10], which is in turn based upon a variant of Tomita's algorithm [19] for context-free grammars. This section gives a general description of the new algorithm, emphasizing its difference from its prototypes. The common aspects of the three algorithms will be presented uniformly, while the crucially different part, the way of doing reductions, will be presented in three versions: for the context-free case, for the conjunctive case and for the Boolean case.

To begin with, all three algorithms share a common data structure: the *graph-structured stack*, introduced by Tomita [19] as a compact representation of the contents of the linear stack of Knuth's LR algorithm in all possible branches of a nondeterministic computation. The graph-structured stack is an oriented graph with a designated *source node*. The nodes are labelled with states of an LR automaton (such as the SLR(k) automaton constructed in the previous section), of which the source node is labelled with the initial state. The arcs are labelled with

symbols from $\Sigma \cup N$. There is a designated nonempty collection of nodes, called *the top layer* of the stack. Every arc leaving one of these nodes has to go to another node in the top layer. The labels of these nodes should be pairwise distinct, and hence there can be at most $|Q|$ top layer nodes.

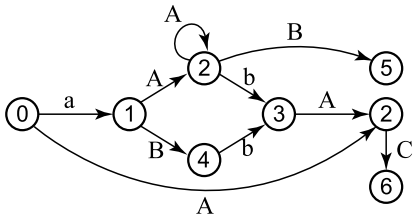

Figure 2: Sample contents of the graph-structured stack.

Consider the graph in Figure 2: the leftmost node labelled 0 is the source node; the three rightmost nodes labelled 5, 2 and 6 are assumed to form the top layer. There is another node labelled 2 (the predecessor of 5), which is not in the top layer.

Initially, the stack contains a single source node, which at the same time forms the top layer. The computation of the algorithm is an alternation of *reduction phases*, in which the arcs going to the top layer are manipulated without consuming the input, and *shift phases*, where a single input symbol is read and consumed, and a new top layer is formed as a successor of the former top layer.

The shift phase is done identically in all three algorithms. Let a be the next input symbol. For each top layer node labelled with a state q, the algorithm looks up the entry $\delta(q, a)$ in the transition table. If $\delta(q, a) = q' \in Q$, then a new node labelled q' is created and q is connected to q' with an arc labelled a; this action is called *Shift* q'. If $\delta(q, a) = -$, no new nodes are created; let us call this condition *a local error*. The nodes created during a shift phase form the new top layer of the graph, while the previous top layer nodes become regular nodes. The branches of the graph-structured stack that do not get extended to the new top layer (due to local errors during the shift phase) are removed; if this is the case for all the nodes, then the entire graph is effectively deleted, and the algorithm terminates, reporting a syntax error.

Consider the example in Figure 3(a). Before the shift phase the top layer contains the nodes 1, 2, 3 and 4. Since $\delta(1, a) = 5$, $\delta(2, a) = -$ and $\delta(3, a) = \delta(4, a) = 6$, a new top layer formed of 5 and 6 is created, while 2 and its predecessors that became disconnected from the new top layer are accordingly removed from the stack.

The reduction phase in each of the cases amounts to doing some uniform transformations of the top layer until the stack comes to a stable condition, that is, no further transformations are applicable. The difference between the three algorithms is in the particular transformations used.

In the **context-free** case [19], the only operation is *reduction*. Let $u \in \Sigma^{\leqslant k}$ be the current lookahead string and suppose there exists a top layer node q, a node q' and a rule $A \to \alpha$, such that $A \to \alpha \in R(q, u)$ and q' is connected to q by a path α. Then the algorithm can perform the operation "Reduce $A \to \alpha$" at q', adding a

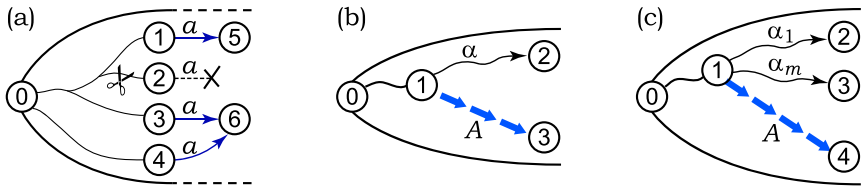

Figure 3: (a) Shifting; (b) Reductions in context-free and (c) conjunctive cases.

new arc labelled by A, which goes from q' to a top layer node labelled $q'' = \delta(q', A)$. If there is no node q'' in the top layer, it is created. This is shown in Figure 3(b).

In the **conjunctive** case [10], *reduction* is still the only operation. However, now rules may consist of multiple conjuncts, and, accordingly, the condition of performing a reduction can now have multiple premises. Let $A \to \alpha_1 \& \ldots \& \alpha_m$ be a rule, let q be a node and let $q_1, \ldots, q_m$ be top layer nodes, such that, for all j, $A \to \alpha_j \in R(q_j, u)$ and q is connected to each q_j by a path α_j. Then the operation "Reduce $A \to \alpha_1 \& \ldots \& \alpha_m$" can be done, adding a new nonterminal arc A from q to a top layer node $\delta(q, A)$, as illustrated in Figure 3(c).

The case of **Boolean** grammars is more complicated. There are two operations: *reduction*, which is the same as in the previous cases, but with yet more complicated conditions, which can now have negative premises, and *invalidation*, which means removing an arc placed by an earlier reduction. In order to reduce by a rule $A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n$ from a node q, this q should be connected to the top layer by each of the paths $\alpha_1, \ldots, \alpha_m$ and by none of the paths $\beta_1, \ldots, \beta_n$; nonexistence of paths is shown in Figure 4(left) by dotted lines ending with crosses. Then the algorithm performs "Reduce $A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n$", adding an arc labelled A from q to $\delta(q, A)$ in the top layer.

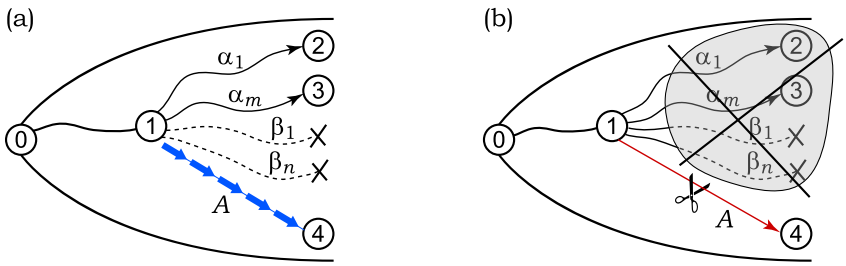

Figure 4: Reduction phase for Boolean grammars: (a) *Reduce* and (b) *Invalidate*.

Invalidation is the opposite of reduction. Suppose there exists a node q and an arc labelled by A from q to a node in the top layer, such that the conditions for making a reduction by any rule for A from the node q are not met, i.e., for every rule $A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n$ for A either for some i there is no path from q to the top layer labelled α_i, or for some j there exists a path from q to the top layer with the labels forming β_j. Then the earlier reduction (which added this A arc to the graph) has to be invalidated, by removing the arc from q to the top layer

node $\delta(q, A)$. Note that an invalidation of an arc can make the graph disconnected.

Let us note that in the case of context-free and conjunctive grammars, in the absence of negation, arcs can only be added, and the conditions for invalidation would never hold. On the other hand, if there is a negation, then a reduction by a rule $A \to \alpha\&\neg\beta$ at a node q can be made at the time when there is a path α from q to the top layer, but there is no path β; however, subsequent reductions may cause this path β to appear, rendering the earlier reduction invalid. Also, subsequent invalidations may cause the path α to disappear, which would also qualify the arc A for invalidation. This is something that does not have an analogue in LR parsing for grammars without negation.

The reduction phase as a whole is defined by the following nondeterministic procedure:

> **while** any reductions or invalidations can be done
> arbitrarily choose a nonempty set of reductions/invalidations to do
> add/remove these arcs simultaneously

This nondeterministic behaviour leaves open a dangerous possibility of nondeterministic results, i.e., that different computations might construct different graphs. Also, if two possible actions are being done at once, one of them can change the graph-structured stack so that the conditions for making the other no longer hold. The termination of the procedure can also be doubted. Though this definition arouses natural suspicions regarding its validity, in Section 7 below the correctness of the algorithm will be proved for any possible choice of reductions and invalidations at every step, under certain very weak assumptions on the grammar.

One possible implementation of the reduction phase is to do just one action at once; this is the obvious approach, yet, as we shall see, it can theoretically lead to an exponential time complexity. The implementation described in Section 8 does all valid reductions and invalidations at every step, which allows one to prove a polynomial complexity upper bound.

Except for these significant differences in the reduction phase, the three algorithms are the same in all other respects. Following is the general schedule of generalized LR parsing:

> *Input:* a string $w = a_1 \dots a_n$.
> Let the stack contain a single node x labelled q_0, let the top layer be $\{x\}$.
> do the *Reduction phase* using lookahead $First_k(w)$
> **for** $i = 1$ **to** n
> do the *Shift phase* using a_i
> **if** the top layer is empty, **then** Reject
> do the *Reduction phase* using lookahead $First_k(a_{i+1} \dots a_n)$
> remove the nodes unreachable from the source node
> **if** there is an arc S from the source node to $\delta(q_0, S)$ in the top layer, **then**
> Accept
> **else**
> Reject

States as elements of $2^{\mathrm{dc}(P)} \cup \{q_{acc}\}$		δ			R	
		a	S	A	ε	a
$\{S \to \cdot A,\ S \to \cdot aS,\ A \to \cdot aA,\ A \to \cdot\}$	0	1	5	2	$A \to \varepsilon$	
$\{S \to a \cdot S,\ A \to a \cdot A,$ $S \to \cdot A,\ S \to \cdot aS,\ A \to \cdot aA,\ A \to \cdot\}$	1	1	3	4	$A \to \varepsilon$	
$\{S \to A\cdot\}$	2				$S \to A$	
$\{S \to aS\cdot\}$	3				$S \to \neg aS$	
$\{S \to A\cdot,\ A \to aA\cdot\}$	4				$S \to A$ $A \to aA$	
q_{acc}	5					

Table 1: The LR table for the grammar from Example 3.

5. Example

Many different grammars could be taken to illustrate the algorithm. If one aims to give a convincing example of the algorithm's clarity and feasibility, one can take any reasonably written grammar for some known non-context-free language, such as the grammar in Example 1, and trace its execution on a short input string. The resulting operation will not be much different from that in the context-free case, which would support the claim that the algorithm is well-suited for a practical use. On the other hand, this kind of demonstration would not give any idea of the mathematical difficulties that appear in the new algorithm for Boolean grammars. and would not much help understanding the proof of its correctness.

Because of that, the easy task of trying the algorithm on reasonable grammars is left to the reader. A short artificial example given in this section is aimed to attract the reader's attention to the new aspects of the algorithm associated with negation, and thus to serve as an introduction to the proof of correctness. This goal is achieved by using the following strange grammar for a simple regular language.

Example 3 *Consider the Boolean grammar*

$$S \to A\&\neg aS$$
$$A \to aA \mid \varepsilon$$

over the unary alphabet, which generates the language $(aa)^*$. *Fix* $k = 1$ *and use the method of Section 3 to construct the sets* $\textsc{Pfirst}_1(S) = \textsc{Pfirst}_1(A) = \{\varepsilon, a\}$ *and* $\textsc{Pfollow}_1(S) = \textsc{Pfollow}_1(A) = \{\varepsilon\}$. *The resulting set of six states and the functions* δ *and* R *are given in Table 1; transitions to the seventh state "–" are denoted by empty squares.*

Consider the computation on the string aa. The parser starts from a single source node, as in Figure 5(a). No reductions are made until the entire input is consumed (consider that $R(q, a) = \varnothing$ for all q), and the contents of the stack before the final reduction phase is presented in Figure 5(b).

Now, according to the transition table, one S-arc and one A-arc from each of the three nodes can possibly be added by reduction: these arcs are shown in Figure 5(c). The arc $A_{[2]}$ can be set by doing a reduction by the rule $A \to \varepsilon$, and it does not

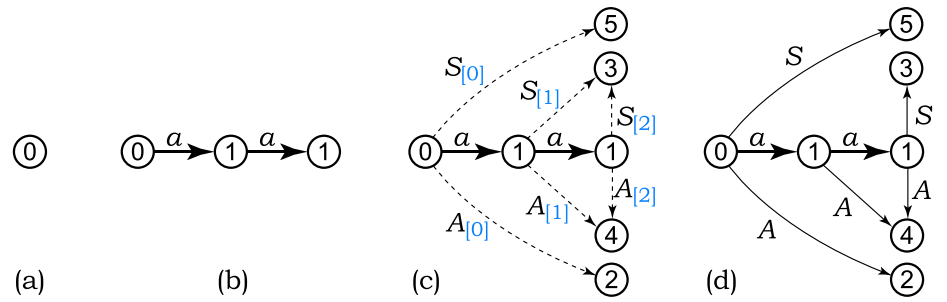

Figure 5: Contents of the stack: (a) in the beginning; (b) before the last reduction phase; (c) possible arcs in the last reduction phase; (d) final.

depend upon the rest of the arcs. The arc $A_{[1]}$ is set by a reduction by $A \to aA$, which can only be done if the arc $A_{[2]}$ is present, and hence there is the required path aA. The arc $A_{[0]}$ is set by the same rule and requires the presence of $A_{[1]}$. Turning to the S-arcs, each of them can be set only by $S \to A\&\neg aS$, since this is the sole rule for S. The arc $S_{[2]}$ can be set, provided that the arc $A_{[2]}$ is in the graph; here the negative conjunct has no impact, since the path aS cannot start from the top layer node 1. On the other hand, the arc $S_{[1]}$ requires both the presence of $A_{[2]}$ and the absence of $S_{[2]}$; if the arc $S_{[2]}$ were in the graph, then there would be a path aS violating the negative conjunct. Similarly, the arc $S_{[0]}$ can be set if and only if $A_{[0]}$ is present and $S_{[1]}$ is absent. The arc $S_{[0]}$ is responsible for accepting the input.

Consider how can the reduction phase proceed. Initially, none of the dotted arcs is in the graph. The first action to be done is the reduction by $A \to \varepsilon$, since it is the only reduction that has no prerequisites; this sets the arc $A_{[2]}$. Next, one can set either the arc $A_{[1]}$ or the arc $S_{[2]}$. If it is the arc $S_{[2]}$ that is set at this point, then it is easy to see that the computation proceeds by setting the arcs $A_{[1]}$, $A_{[0]}$ and $S_{[0]}$ is this exact order, which concludes the reduction phase. Note that the arc $S_{[1]}$ cannot be set, since there is a path aS containing the arc $S_{[2]}$ that prohibits that. The final contents of the graph is given in Figure 5(d).

On the other hand, if the reduction phase proceeds by setting $A_{[2]}$ and $A_{[1]}$, then it becomes possible to add the arc $S_{[1]}$ instead of $S_{[2]}$. This can lead to a configuration of the arcs $A_{[2]}$, $A_{[1]}$, $A_{[0]}$ and $S_{[1]}$. The only thing the algorithm can do at this point is to set the arc $S_{[2]}$. Then the condition of existence of the arc $S_{[1]}$ is violated, and the only possible next action is the invalidation of $S_{[1]}$. After that the algorithm can do the final reduction, setting the arc $S_{[0]}$ and arriving at the same graph shown in Figure 5(d).

These are not all possible cases. Since, according to the definition of the reduction phase, several reductions and invalidations can be done simultaneously, following is another valid sequence of operations: reduce $A_{[2]}$; reduce $A_{[1]}$; reduce $A_{[0]}$; reduce $S_{[1]}$ and reduce $S_{[0]}$ in parallel; invalidate $S_{[0]}$ and reduce $S_{[2]}$ in parallel; invalidate $S_{[1]}$; reduce $S_{[2]}$. The resulting graph is again the same as in Figure 5(d).

Having had a look at this example, one is likely to develop doubts about the

algorithm's soundness. Why should this procedure always terminate? Are the results consistent for different choices of reductions and invalidations? How does this computation correspond to the original Boolean grammar, or, in other words, to a system of language equations? In the next section we shall see that, except for an insignificant class of prohibited grammars, the algorithm always works consistently and determines the membership of strings in the language correctly.

6. Domain of applicability

One class of Boolean grammars on which the algorithm works correctly are the grammars in the binary normal form. This, in particular, implies that every language generated by a Boolean grammar can be parsed using this algorithm. Another case where the algorithm surely works is the case of grammars without negation, the *conjunctive grammars* [9]: here the parsing will proceed without using invalidations, and the addition of arcs will correspond to derivations in a grammar [10].

However, when chain dependencies represented by unit conjuncts are combined with negation, the algorithm can go astray and either report inconsistent results for different computations, or enter an infinite loop in one of the computations. Before formulating a sufficient condition of the algorithm's applicability, let us consider the representative problematic cases.

Example 4 (Inconsistent computations) *Consider a Boolean grammar*

$$S \to S \mid a\&\neg aE$$
$$E \to \varepsilon$$

which generates the language $\varnothing$*. The corresponding LR parser has both an accepting and a rejecting computation on the input* a*.*

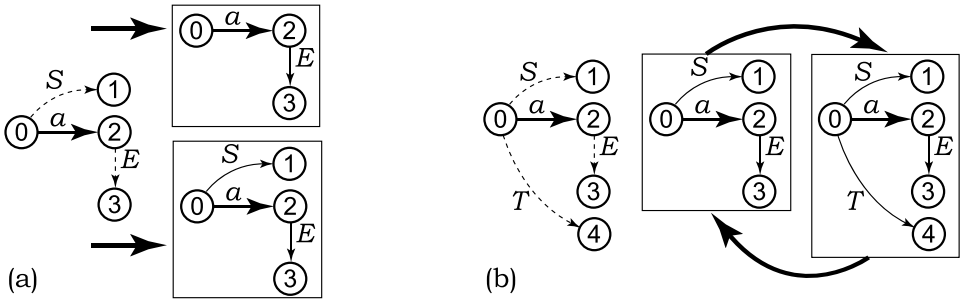

Figure 6: (a) Inconsistency in Example 4; (b) Nontermination in Example 5.

The case is simple enough to omit the parsing table. The form of the graph-structured stack during the second and the last reduction phase is shown in Figure 6(a). Two reductions are possible in the beginning: by $E \to \varepsilon$ or by $S \to a\&\neg aE$; the outcome of the computation depends upon the first reduction done. If it is by the rule $E \to \varepsilon$, then the reduction by $S \to a\&\neg aE$ becomes impossible, the reduction phase terminates and the parser rejects the input. However,

if the reduction by the rule $S \to a\&\neg aE$ takes place before the reduction by $E \to \varepsilon$, then the arc labelled S cannot be invalidated, because its presence is now justified by the rule $S \to S$.

Note that the grammar given in Example 4 belongs to the subclass of *stratified Boolean grammars* introduced by Wrona [20], in which negation cannot be iterated. The next example uses iterated negation to express a contradiction, which can lead the parser into a hopeless infinite search for a solution.

Example 5 (Nonterminating computation) *The Boolean grammar*

$$\begin{aligned} &T \to \neg T\&S \\ &S \to S \mid a\&\neg aE \\ &E \to \varepsilon \end{aligned}$$

with the start symbol T generates the language $\varnothing$. The corresponding LR parser can go into an infinite loop on the input a.

The nonterminals S and E are as in the previous example, T is the new start symbol. If the first reduction is by $E \to \varepsilon$, then the reduction by $S \to a\&\neg aE$ will be blocked. In the absence of the arc S, the contradiction expressed in the rule for T will not take effect. However, if the first reduction in by $S \to a\&\neg aE$, then the reduction by $T \to \neg T\&S$ becomes possible. Once the arc T is added to the stack, it violates its own condition of existence, and the algorithm proceeds with invalidating this arc. The computation proceeds infinitely by alternating reductions and invalidations of the arc T.

In both cases the essential cause of incorrect behaviour is a circular dependence of a nonterminal upon itself (represented by the rule $S \to S$) combined with a negative dependence of this nonterminal on something. Let us formalize this condition.

Definition 8 *Let G be a Boolean grammar, and let $G_+ = (\Sigma, N, P_+, S)$ be a conjunctive grammar with the set of rules*

$$P_+ = \{A \to \alpha_1\& \ldots \&\alpha_m \mid A \to \alpha_1\& \ldots \&\alpha_m\&\neg\beta_1\& \ldots \&\neg\beta_n \in P\}$$

Let $Nullable(G_+) = \{A \mid \varepsilon \in L_{G_+}(A)\}$.

A sequence of conjuncts $A_1 \to \pm\eta_1 A_2\theta_1$, $A_2 \to \pm\eta_2 A_3\theta_2$, $\ldots A_\ell \to \pm\eta_\ell A_{\ell+1}\theta_\ell$ such that $\ell \geqslant 1$ and $\eta_i, \theta_i \in (Nullable(G_+))^*$ is called a chain from A_1 to $A_{\ell+1}$. A cycle is a chain from a nonterminal to itself. If the condition $\eta_i \in (Nullable(G_+))^*$, is lifted, while the rest of the conditions remain (including the requirement that $\theta_i \in (Nullable(G_+))^*$ for all i), the sequence is called a right-chain from A_1 to $A_{\ell+1}$.

Definition 9 *Let G be a Boolean grammar and let a nonterminal A have a chain to itself. This cycle is said to be negatively fed by a right-chain, if there exists a right-chain from A to a nonterminal B, such that some rule for B contains a negative conjunct. In the following, a cycle negatively fed by a right-chain will be referred as just a negatively fed cycle.*

The grammars from Examples 4 and 5 each have a cycle from S to S negatively fed by the conjunct $S \to \neg A$.

The use of negatively fed cycles does not give any extra expressive power to a Boolean grammar. They can be effectively removed from a grammar, for instance, by transforming the grammar to the binary normal form. From the practical point of view, a negatively fed cycle is something unnatural, which one would not typically write, unless aiming to produce a counterexample for this algorithm, and avoiding such cycles can be regarded as a matter of writing a grammar properly. So there is hardly any loss of generality in the assumption that a grammar does not have such cycles.

In addition, this assumption guarantees that the semantics of a grammar is well-defined.

Theorem 1 *Every Boolean grammar without negatively fed cycles complies to the semantics of naturally reachable solution. Furthermore, for every M and w, as in Definition 3, there exists a* ***monotone*** *sequence (3), in which the string w is only added to the components of the vector and is never removed.*

The proof is by reduction to a system of Boolean equations. Let us first define the corresponding restriction on Boolean equations.

Definition 10 *Let $f(x_1,\ldots,x_n)$ be a Boolean function of n variables (i.e., $f:\mathbb{B}^n\to\mathbb{B}$). A variable x_i is said to be an essential variable of f, if $f(c_1,\ldots,c_{i-1},0,c_{i+1},\ldots,c_n)\neq f(c_1,\ldots,c_{i-1},1,c_{i+1},\ldots,c_n)$ for some $c_1,\ldots,c_{i-1},c_{i+1},\ldots,c_n\in\mathbb{B}$.*

Definition 11 *Let $x_i=f_i(x_1,\ldots,x_n)$ $(1\leqslant i\leqslant n)$ be a system of Boolean equations. A sequence of variables $x_{i_1},\ldots,x_{i_{\ell+1}}$ $(\ell\geqslant 1)$, such that $x_{i_{j+1}}$ is an essential variable in f_{i_j} for all j $(1\leqslant j\leqslant \ell)$, is called a chain from x_{i_1} to $x_{i_{\ell+1}}$. A cycle is a chain from a variable to itself. It is said to be negatively fed, if there exists a chain from some x_{i_j} to a variable x_k, such that f_k is not a monotone function.*

The following lemma on Boolean equations contains the main idea of the proof of Theorem 1. This lemma will also be used later in the proof of the algorithm's correctness.

Lemma 1 *Let $x_i=f_i(x_1,\ldots,x_n)$ $(1\leqslant i\leqslant n)$ be a system of Boolean equations without negatively fed cycles. Consider any sequence of Boolean vectors*

$$(b_1^{(0)},\ldots,b_n^{(0)}),(b_1^{(1)},\ldots,b_n^{(1)}),\ldots,(b_1^{(j)},\ldots,b_n^{(j)}),\ldots, \tag{5}$$

such that $(b_1^{(0)},\ldots,b_n^{(0)})=(0,\ldots,0)$, and for every $j\geqslant 0$ it holds that:

- *There exists a nonempty set $\{t_1,\ldots,t_p\}\subseteq\{1,\ldots,n\}$ of positions in the vector, such that $b_{t_i}^{(j+1)}=f_{t_i}(b_1^{(j)},\ldots,b_{t_i-1}^{(j)})$ (for $1\leqslant t_i\leqslant n$) and $b_i^{(j+1)}=b_i^{(j)}$ for all positions i not in the designated set, and*
- *$(b_1^{(j+1)},\ldots,b_n^{(j+1)})\neq(b_1^{(j)},\ldots,b_n^{(j)})$,*

Then, regardless of the choice of the set of positions at each step, this sequence converges to a solution of the system in at most 2^n steps, and all these sequences converge to the same solution. If $\{1,\ldots,n\}$ is chosen as a set of positions at every step, the sequence converges in at most n steps. Also, there exists a particular choice of sets of positions, such that the sequence (5) increases and converges in at most n steps.

Proof. The absence of negatively fed cycles implies that the variables can be arranged into two groups, $(x_1, \ldots, x_\ell)$ and $(x_{\ell+1}, \ldots, x_n)$, and the latter group can be sorted, so that the system of equations can be written as

$$\begin{cases} x_1 = p_1 \quad (x_1, \ldots, x_\ell) \\ \quad\vdots \\ x_\ell = p_\ell \quad (x_1, \ldots, x_\ell) \\ x_{\ell+1} = f_{\ell+1} \quad (x_1, \ldots, x_\ell) \\ x_{\ell+2} = f_{\ell+2} \quad (x_1, \ldots, x_\ell, x_{\ell+1}) \\ x_{\ell+3} = f_{\ell+3} \quad (x_1, \ldots, x_\ell, x_{\ell+1}, x_{\ell+2}) \\ \quad\vdots \\ x_n = f_n \quad (x_1, \ldots, x_\ell, x_{\ell+1}, x_{\ell+2}, \ldots, x_{n-1}) \end{cases} \tag{6}$$

where $p_1, \ldots, p_\ell$ are monotone Boolean functions (that is, representable by a composition of conjunction, disjunction, 0, 1 and the variables), while $f_{\ell+1}, \ldots f_n$ are arbitrary Boolean functions.

The target solution $c = (c_1, \ldots, c_\ell, c_{\ell+1}, \ldots, c_n)$ is defined as follows: $(c_1, \ldots, c_\ell)$ is the least solution of the first ℓ equations in (6), given by ℓ applications of a vector function $(p_1, \ldots, p_\ell)$ to a zero vector, while each of the subsequent c_i $(\ell < i \leqslant n)$ is $f_i(c_1, \ldots, c_\ell, c_{\ell+1}, \ldots, c_{i-1})$. It remains to prove that every sequence (5) converges to this vector in finitely many steps.

Each of the first ℓ components of every vector $b^{(j)}$ is less or equal to the corresponding component of $(c_1, \ldots, c_\ell)$, which can be proved by a straightforward induction on j. Define the following measure of divergence of a vector from the target solution: this is a mapping $\tau : \mathbb{B}^n \to \mathbb{B}^n$, such that for every Boolean vector $(x_1, \ldots, x_n)$, $\tau(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$, where

$$\begin{aligned} y_1 &= x_1 \oplus c_1, \\ &\vdots \\ y_\ell &= x_\ell \oplus c_\ell, \\ y_{\ell+1} &= x_{\ell+1} \oplus f_{\ell+1}(x_1, \ldots, x_\ell), \\ y_{\ell+2} &= x_{\ell+2} \oplus f_{\ell+2}(x_1, \ldots, x_\ell, x_{\ell+1}), \\ y_{\ell+3} &= x_{\ell+3} \oplus f_{\ell+3}(x_1, \ldots, x_\ell, x_{\ell+1}, x_{\ell+2}) \\ &\vdots \\ y_n &= x_{\ell+1} \oplus f_n(x_1, \ldots, x_\ell, x_{\ell+1}, x_{\ell+2}, \ldots, x_{n-1}) \end{aligned}$$

It is claimed that for every sequence (5) the corresponding sequence $\tau(b_1^{(0)}, \ldots, b_n^{(0)}), \ldots, \tau(b_1^{(k)}, \ldots, b_n^{(k)})$ decreases with respect to the lexicographical order.

Let $(x_1, \ldots, x_n)$ be a term of the sequence (5), let $(x'_1, \ldots, x'_n)$ be the next term. Denote $(y_1, \ldots, y_n) = \tau(x_1, \ldots, x_n)$ and $(y'_1, \ldots, y'_n) = \tau(x'_1, \ldots, x'_n)$. Let i be the least number, such that $x_i \neq x'_i$. This implies that i is the least number, such that $y_i \neq y'_i$.

If $1 \leqslant i \leqslant \ell$, then $x_i = 0$ and $x'_i = c_i = 1$ by the monotonicity of the first ℓ components and by the construction of c_i; hence $y_i = 1$ and $y'_i = 0$. On the other

hand, if $\ell < i \leqslant n$, then $x'_i = f_i(x_1, \ldots, x_{i-1})$ by the definition of the sequence (5), and therefore, since $x_i \neq x'_i$, $y_i = x_i \oplus f_i(x_1, \ldots, x_{i-1}) = x_i \oplus x'_i = 1$, while $y'_i = x'_i \oplus f_i(x'_1, \ldots, x'_{i-1}) = x'_i \oplus f_i(x_1, \ldots, x_{i-1}) = x'_i \oplus x'_i = 0$. In each case $\tau(x'_1, \ldots, x'_n)$ is lexicographically less than $\tau(x_1, \ldots, x_n)$.

So the sequence $\{\tau(b^{(i)})\}$ is decreasing, and hence it converges. Its limit $\tau(x_1, \ldots, x_n)$ is the zero vector: assuming the contrary, it is easy to show that the corresponding vector $(x_1, \ldots, x_n)$ would not be a solution of the system, and hence further transformations would be applicable to it. If $\tau(x_1, \ldots, x_n) = (0, \ldots, 0)$, then $(x_1, \ldots, x_n) = (c_1, \ldots, c_n)$, which is the limit of the sequence regardless of the order of transformations. So the sequence (5) converges. It converges in at most 2^n steps, because there are 2^n distinct vectors of n bits.

It remains to consider two special cases. Suppose the set $\{1, \ldots, \ell\}$ is chosen until the first ℓ components reach $(c_1, \ldots, c_\ell)$ (this takes at most ℓ steps), and then sequence proceeds with the following sets of positions: $\{\ell + 1\}$ (if $f_{\ell+1}(c_1, \ldots, c_\ell) = 1$), $\{\ell + 2\}$ (if $f_{\ell+2}(c_1, \ldots, c_\ell, c_{\ell+1}) = 1$), and so on until $\{n\}$ (if $f_n(c_1, \ldots, c_\ell, c_{\ell+1}, \ldots, c_{n-1}) = 1$). Thus the true bits among $(c_{\ell+1}, \ldots, c_n)$ are gradually set to 1, forming the promised monotone sequence.

If the set of positions chosen at every step is $\{1, \ldots, n\}$, then the first ℓ positive components converge in at most ℓ steps as in the previous case, but the remaining $n - \ell$ acyclic components can assume various wrong values in the process. However, the subsequent iterations (there are at most $n - \ell$ of them) gradually correct these values, and the sequence finally converges to $(c_1, \ldots, c_n)$. □

Proof of Theorem 1. For all M, w and $L \pmod M$ as in the definition of naturally reachable solution, it has to be shown that the computation of a naturally reachable solution modulo $M \cup \{w\}$ always terminates and converges to the same vector modulo $M \cup \{w\}$. The proof is an induction on the cardinality of M.

Basis. $M = \varnothing$, $w = \varepsilon$.

$$x_A = \bigvee_{A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P} \left(\bigwedge_{i=1}^{m} \mathit{conjunct}(\alpha_i) \ \wedge \ \bigwedge_{i=1}^{m} \neg \mathit{conjunct}(\beta_i) \right) \quad (7a)$$

$$\mathit{conjunct}(s_1 \ldots s_k) = \begin{cases} x_{s_1} \wedge \ldots \wedge x_{s_k}, & \text{if } s_i \in \mathit{Nullable}(G_+) \text{ for all } i \\ 0 & \text{otherwise} \end{cases} \quad (7b)$$

Let the equation for a variable x_A refer to a variable x_B in its right hand side; this implies $\varepsilon \in L_{G_+}(s_i)$ for all i, and hence a conjunct $A \to \pm\eta B\theta$, such that $\varepsilon \in L_{G_+}(\eta)$ and $\varepsilon \in L_{G_+}(\theta)$. Hence, every chain in the dependencies (7) implies a chain in the grammar in the sense of Definition 8, and every negatively fed cycle in (7) implies a cycle negatively fed by a chain in the grammar. By assumption, the original grammar contains no such cycles, which implies that the Boolean system (7) has no negatively fed cycles. Therefore, Lemma 1 holds for (7), and the condition from the definition of naturally reachable solution is satisfied.

Induction step. Let L_M be the naturally reachable solution modulo M. Con-

struct a system of Boolean equations as follows:

$$x_A = \bigvee_{A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P} \left(\bigwedge_{i=1}^{m} conjunct(\alpha_i) \ \wedge \ \bigwedge_{i=1}^{m} \neg conjunct(\beta_i) \right) \quad (8a)$$

$$conjunct(s_1 \ldots s_k) = \begin{cases} 1, & \text{if } w \in s_1 \ldots s_k(L_M) \\ \bigvee_{i:\, \varepsilon \in L_M(s_j) \text{ for all } j \neq i} x_{s_k} & \text{otherwise} \end{cases} \quad (8b)$$

Again, if the equation for x_A refers to x_B, this implies a conjunct $A \to \pm\eta B\theta$, such that $\varepsilon \in L_{G_+}(\eta)$ and $\varepsilon \in L_{G_+}(\theta)$. The rest of the argument is as in the basis case. □

7. Proof of correctness

Let G be a Boolean grammar without negative fed cycles. The task is to establish the correctness of the LR algorithm described in Sections 3 and 4. The proof is comprised of two parts. First, it has to be shown that the algorithm reports consistent results for every input string w, that is, the same graph is constructed for every choice of actions during the reduction phases. Second, it should be argued how does the constructed graph correspond to the original grammar. It will follow then that w is accepted if and only if $w \in L(G)$.

In order to analyze the LR parsing algorithm, it is convenient to redefine the graph-structured stack by augmenting its nodes with the information on when they were added: a layer number corresponding to a position in the input. There is no need to store this extended information in an implementation; simply the properties of the algorithm become much clearer in these terms.

Definition 12 *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar, let (Q, q_0, δ, R) be the SLR(k) automaton, let $w = a_1 \ldots a_{|w|}$ be the input string.*

The graph-structured stack is an acyclic graph with a set of vertices $V \subseteq Q \times \{0, 1, \ldots, |w|\}$ and with the arcs labelled with symbols from $\Sigma \cup N$, such that the following condition holds: for every arc from (q', p') to (q'', p'') labelled with $s \in \Sigma \cup N$, $p' \leqslant p''$ and $\delta(q', s) = q''$.

For each p ($0 \leqslant p \leqslant n$), the set of all vertices of the form (q, p), where $q \in Q$, is called the p-th layer of the graph. A nonempty layer with the greatest number is called the top layer.

Consider the reduction phase in a layer p ($0 \leqslant p \leqslant |w|$): the symbols $a_1 \ldots a_p$ have already been read, while $First_k(a_{p+1} \ldots a_{|w|})$ is the lookahead string. For the sake of the argument, let us first suppose that every state q is added to the top layer in the beginning of the reduction phase, that is, $(q, p) \in V$ for all $q \in Q$. The reduction phase thus modified will be called the *simplified reduction phase*. The correctness of the simplified version of the algorithm will be proved first; the correctness of the algorithm as it is will be subsequently established as a corollary of the simplified algorithm's correctness.

This arrangement of the proof initially allows us to abstract from the matter of creating nodes in the top layer, and to represent the reduction phase as a process

of adding arcs to the graph and removing arcs from the graph. Let us use the term *permanent arcs* for the arcs present in the stack immediately before the reduction phase, such as those in Figure 5(b). It is easy to see from the algorithm that all permanent arcs going to the top layer are labelled with terminals.

The arcs that, according to the transition table, can theoretically be added during this reduction phase will be called *possible arcs*, such as those shown with dotted lines in Figure 5(c). Every possible arc is a nonterminal arc going to the top layer, and the reduction phase is a manipulation of these arcs. Initially, there are none of them. A reduction adds an arc. An invalidation removes an arc. This process can be viewed as negating the bits in the bit vector of possible arcs, starting from a zero vector.

A possible arc from (q, p_0) to $(\delta(q, A), p)$ and labelled with $A \in N$, where $q \in Q$, $0 \leqslant p_0 \leqslant p$, will be formally denoted by a triple (q, p_0, A), It is required that the transition $\delta(q, A)$ is defined and there is either a permanent arc entering the vertex (q, p_0), or an arc already known to be a possible arc. There are at most $p \cdot |Q|^2$ possible arcs.

Each possible arc (q, p_0, A) has its *condition of existence*, which causes the reduction by some rule for A at the vertex (q, p_0), or prevents an invalidation of this arc. Let $\langle x_{q,p_0,A} \rangle$ be a vector of Boolean variables corresponding to the possible arcs. The condition of existence of every possible arc $x_{q,p_0,A}$ can be written as the following Boolean formula over these variables:

$$x_{q,p_0,A} = \bigvee_{\substack{A \to \alpha_1 \& \ldots \& \alpha_m \& \\ \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P}} \left(\bigwedge_{i=1}^{m} conj(q, p_0, A \to \alpha_i) \ \wedge \ \bigwedge_{i=1}^{m} \neg conj(q, p_0, A \to \neg\beta_i) \right)$$

$$conj(q, p_0, A \to \pm\gamma) = \begin{cases} path(q, p_0, \gamma), & \\ & \text{if } A \to \pm\gamma \in R(\delta(q, \gamma), First_k(a_{p+1} \ldots a_{|w|})) \\ 0 & \text{otherwise} \end{cases}$$

$$path(q, p_0, \alpha) = \bigvee_{p'=p_0}^{p-1} \bigvee_{\substack{\alpha = \alpha' C D_1 \ldots D_\ell: \\ \exists \text{ path } \alpha' \\ \text{from } (q, p_0) \text{ to layer } p'; \\ D_i \in Nullable(G_+)}} \left(x_{\delta(q,\alpha' C), p', C} \wedge \bigwedge_{i=1}^{\ell} x_{\delta(q,\alpha' C D_1 \ldots D_{i-1}), p, D_i} \right) \vee$$

$$\vee \bigvee_{\substack{\alpha = \alpha' a D_1 \ldots D_\ell: \\ \exists \text{ path } \alpha' a \text{ from } (q, p_0) \text{ to} \\ \text{layer } p;\ D_i \in Nullable(G_+)}} \bigwedge_{i=1}^{\ell} x_{\delta(q,\alpha' a D_1 \ldots D_{i-1}), p, D_i}, \qquad \text{if } p_0 < p$$

$$path(q, p, D_1 \ldots D_\ell) = \begin{cases} \bigwedge_{i=1}^{\ell} x_{\delta(q, D_1 \ldots D_{i-1}), p, D_i}, & \text{if } p_0 = p,\ D_i \in Nullable(G_+) \\ 0 & \text{otherwise} \end{cases}$$

The expression $path(q, p_0, \alpha)$ states the existence of a path from a vertex (q, p_0) to the top layer p, with the labels forming the string $\alpha \in (\Sigma \cup N)^*$. There are three cases, which are illustrated in Figure 7. If $p_0 < p$, then the path should contain an arc from some layer p', where $p_0 \leqslant p' < p$, to the layer p. This arc can be labelled

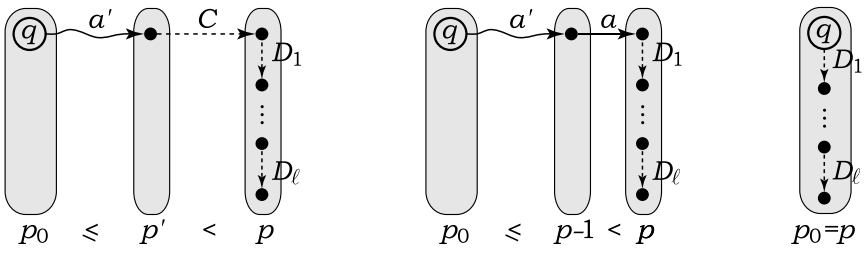

Figure 7: $path(q, p_0, \alpha)$: three cases.

by a nonterminal C (the first case) or by a terminal a (the second case). In the third case, $p_0 = p$, all arcs are nonterminal arcs within the top layer.

The reduction phase can now be viewed as a search for a solution of this system of Boolean equations, which resembles the definition of the semantics of a Boolean grammar (Definition 3).

Lemma 2 *If the equation for a variable $x_{q_1,p_1,A}$ refers to a variable $x_{q_2,p_2,B}$, then:*

1. $p_1 \leqslant p_2$.
2. *There exists a conjunct $A \to \pm\eta B\theta$, such that $\theta \overset{G_+}{\Longrightarrow}{}^* \varepsilon$ and, in the case $p_1 = p_2$, $\eta \overset{G_+}{\Longrightarrow}{}^* \varepsilon$.*

Proof. If there is an equation $x_{q_1,p_1,A} = \ldots x_{q_2,p_2,B} \ldots$, then, according to the above construction, there exists a conjunct $A \to \pm\eta B\theta$, such that the variable $x_{q_2,p_2,B}$ appears in $path(q_1, p_1, \alpha)$, There are four places where it can occur. In each of these places, p_2 has to be greater or equal to p_1, and all the symbols to the right of B (i.e., in θ) have to be nonterminals from $Nullable(G_+)$ by the construction. In the case $p_1 = p_2$, all symbols in $\eta B\theta$ must be in $Nullable(G_+)$, and hence $\eta \overset{G_+}{\Longrightarrow}{}^* \varepsilon$ as well. □

Corollary 1 *If the equation for a variable $x_{q_1,p_1,A}$ refers to a variable $x_{q_2,p_2,B}$, then, using the terminology of Definition 8, there exists a one-step right-chain from A to B; if $p_1 = p_2$, then this is a one-step chain from A to B.*

Lemma 3 (Termination and confluence of simplified reduction phase) *Let G be a Boolean grammar without negatively fed cycles, let (Q, q_0, δ, R) be any LR automaton, let $w \in \Sigma^*$ be an input string. Then, for every layer p ($0 \leqslant p \leqslant |w|$) and for every initial contents of the graph-structured stack (in the form of the permanent arcs), the simplified reduction phase in the layer p terminates, and the resulting contents of the graph-structured stack do not depend upon the order of reductions and invalidations.*

Proof. The main claim is that the resolved system of Boolean equations in variables $\langle x_{q,p_0,A} \rangle$ constructed above does not have negatively fed cycles in the sense of Lemma 1. Suppose there is such a cycle, i.e., a sequence of variables $\langle x_{q_i,p_i,A_i} \rangle_{i=1}^{\ell}$,

such that the variables depend upon each other as follows:

$$\begin{array}{lcl}
x_{q_1,p_1,A_1} & = & \ldots x_{q_2,p_2,A_2} \ldots \\
x_{q_2,p_2,A_2} & = & \ldots x_{q_3,p_3,A_3} \ldots \\
 & \vdots & \\
x_{q_{n-1},p_{n-1},A_{n-1}} & = & \ldots x_{q_n,p_n,A_n} \ldots \\
x_{q_n,p_n,A_n} & = & \ldots x_{q_{n+1},p_{n+1},A_{n+1}} \ldots x_{q_1,p_1,A_1} \ldots \\
x_{q_{n+1},p_{n+1},A_{n+1}} & = & \ldots x_{q_{n+2},p_{n+2},A_{n+2}} \ldots \\
 & \vdots & \\
x_{q_{\ell-1},p_{\ell-1},A_{\ell-1}} & = & \ldots x_{q_\ell,p_\ell,A_\ell} \ldots \\
x_{q_\ell,p_\ell,A_\ell} & = & \ldots \neg(\ldots) \ldots
\end{array} \tag{10}$$

and the right hand side of the equation for x_{q_ℓ,p_ℓ,A_ℓ} employs negation.

According to the first part of Lemma 2, the first n equations in (10) imply $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_{n-1} \leqslant p_n \leqslant p_1$, which obviously means that

$$p_1 = p_2 = \ldots = p_n \tag{11}$$

Using (11) with Corollary 1 for the first n equations in (10), the existence of a chain from A_n to A_n is deduced. Applied for equations for $x_{q_n,p_n,A_n}, \ldots, x_{q_{\ell-1},p_{\ell-1},A_{\ell-1}}$, Corollary 1 gives a right-chain from A_n to A_ℓ. The negation in the equation for x_{q_ℓ,p_ℓ,A_ℓ} implies the existence of a negative conjunct for one of the rules for A_ℓ. Hence a negatively fed cycle in the grammar is obtained, which contradicts the assumption.

This proves that the system of Boolean equations constructed above has no negatively fed cycles. Therefore, as established in Lemma 1, the search for a solution of this Boolean system is successful, and it ends with the same solution regardless of the choice of components at every step. Since this transformation of Boolean vectors directly corresponds to the transformation of the possible arcs in the graph-structured stack in course of the simplified reduction phase, this completes the proof of the lemma. □

Lemma 4 (Termination and confluence of the reduction phase) *Let G be a Boolean grammar without negatively fed cycles, let (Q, q_0, δ, R) be any LR automaton, let $w \in \Sigma^*$ be an input string. Then, for every layer p ($0 \leqslant p \leqslant |w|$) and for every initial contents of the stack, the reduction phase in the layer p terminates, and, provided that the parts of the graph unreachable from the source node are discarded afterwards, the resulting contents of the stack do not depend upon the order of reductions and invalidations.*

Proof. It is easy to see that every possible computation of the standard reduction phase is at the same time a possible computation of the simplified reduction phase.

This observation immediately implies the termination of the standard reduction phase under any choice of actions. Supposing that it could continue infinitely for some order of reductions and invalidations, the same infinite sequence of actions would be possible in the simplified reduction phase, which would contradict Lemma 3.

It remains to prove that the unique result of the simplified reduction phase is the same (modulo reachability) as the graph constructed by the standard reduction phase in each of its computations. Consider an arbitrary computation of the standard reduction phase, which ends at some stable graph Γ_0. The simplified reduction phase might continue further, producing the graphs $\Gamma_1, \Gamma_2, \ldots, \Gamma_\ell, \ldots$, until the unique resulting graph is reached in one or another way. It is claimed that the reachable portion of each of the graphs Γ_ℓ in each of these continuations is equal to the reachable portion of Γ_0.

The proof is an induction on ℓ. Basis, Γ_0 equals Γ_0 modulo reachability: true. For the induction step, suppose Γ_ℓ equals Γ_0 modulo reachability, and consider the next graph $\Gamma_{\ell+1}$. Let (q, p) be the node, from which the action at the $(\ell+1)$-th step of this computation (whether a reduction or an invalidation) does originate.

Let us see that (q, p) is not reachable in Γ_0. Supposing that it is, the set of paths from (q, p) to the top layer should be the same in Γ_0 and Γ_ℓ by the induction hypothesis, because these paths become reachable in this case. Hence, the action done at this step would be applicable to Γ_0 with respect to a standard reduction phase, which would contradict the assumption that Γ_0 is stable.

Since (q, p) is not reachable in Γ_0, it is not reachable in Γ_ℓ. Hence the modifications at the $(\ell+1)$-th step refer to the unreachable portion of Γ_ℓ, and therefore the reachable parts of Γ_0 and $\Gamma_{\ell+1}$ are equal. □

Lemma 4 implies termination and confluence of the whole algorithm. If G is a Boolean grammar without negatively fed cycles and (Q, q_0, δ, R) is an LR automaton, then, for every input string $w \in \Sigma^*$, either the algorithm terminates and accepts regardless of the order of actions, or the algorithm terminates and rejects regardless of the order of actions. So, despite its ambiguously defined computation, the algorithm constructs some definite graph and computes some definite predicate on Σ^*. It is left to show that this predicate is exactly the membership in $L(G)$, while the contents of the stack after every phase have a certain meaning in terms of the grammar. This meaning is stated in the following definition:

Definition 13 *An arc from a node (q, p) to a node $(\delta(q, s), p')$ labelled s ($s \in \Sigma \cup N$) is called **correct** if and only if $a_{p+1} \ldots a_{p'} \in L_G(s)$ and, in the case $s = A \in N$, $First_k(a_{p+1} \ldots a_n) \in \textsc{Follow}_k(A)$.*

*A correct arc is called **reachable** if and only if either it originates from the source node, or it originates from a node to which there comes a correct arc known to be reachable.*

*Let $0 \leqslant i \leqslant |w|$. A reachable correct arc is called i-**reaching** if and only if either it goes to the layer i, or it goes to a node, from which there originates a reachable correct arc known to be i-reaching.*

The goal of the whole algorithm is to construct the graph comprised of all reachable and $|w|$-reaching correct arcs. In order to prove that the algorithm actually constructs such a graph, it is sufficient to present *a single computation* of the algorithm and establish the correctness of this computation. This, by Lemma 4, will imply that the rest of possible computations converge to the same correct result.

Lemma 5 (On a model computation in the simplified reduction phase) *Let G be a Boolean grammar without negatively fed cycles, let (Q, q_0, δ, R) be an SLR(k) automaton, let $w \in \Sigma^*$ be an input string, let $p \geqslant 0$. Suppose the graph-structured stack contains exactly the following arcs:*

- *all reachable and p-reaching correct arcs going to the layers $0, 1, \ldots, p-1$,*
- *all reachable and p-reaching correct terminal arcs coming to the layer p.*

Then there exists a computation of the simplified reduction phase that employs reductions only and adds exactly the following arcs:

- *all reachable and p-reaching correct nonterminal arcs from the layers $0, 1, \ldots, p-1$ to the layer p,*
- *all correct nonterminal arcs within the layer p.*

The resulting configuration of arcs is stable.

The proof of Lemma 5 is by a construction of such a computation. The construction is based upon the following correspondence between the definition of a naturally reachable solution and the computation of the simplified reduction phase.

Lemma 6 (Simulating naturally reachable solution in model computation) *Let G have no negatively fed cycles, let (Q, q_0, δ, R) be an SLR(k) automaton, let $w \in \Sigma^*$ be a string. Let $0 \leqslant p_0 \leqslant p$, denote $u = a_{p_0+1}a_{p_0+2}\ldots a_p$ and $v = a_{p+1}a_{p+2}\ldots a_{|w|}$. For M equal to the set of proper substrings of u, and for this string u, let $L^{(0)}, \ldots, L^{(t)}$ be a monotone sequence of vectors of languages modulo $M \cup \{u\}$ corresponding to the definition of a naturally reachable solution, which exists according to Theorem 1. Let $0 \leqslant \ell \leqslant t$.*

Suppose the stack during the simplified reduction phase in the layer p contains exactly the following arcs:

- *all reachable and p-reaching correct arcs going to the layers $0, 1, \ldots, p-1$,*
- *all reachable and p-reaching correct terminal arcs coming to the layer p,*
- *all reachable correct nonterminal arcs from the layers $p_0+1, p_0+2, \ldots, p-1$ to the layer p,*
- *if $p_0 < p$, all correct nonterminal arcs within the layer p,*
- *all correct arcs (all reachable correct arcs, if $p_0 < p$) from the layer p_0 to the layer p labelled with nonterminals B, such that $u \in B(L^{(\ell)})$.*

Then: (i) for every possible arc (q, p_0, A), such that $First_k(v) \in \textsc{Follow}_k(A)$, and for every conjunct $A \to \pm\gamma$, the graph contains a path γ from (q, p_0) to the top layer if and only if $u \in \gamma(L^{(\ell)})$; (ii) the condition of existence of a possible arc (q, p_0, A) is satisfied if and only if $u \in (\alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n)(L^{(\ell)})$ for some rule

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P, \tag{12}$$

and $First_k(v) \in \textsc{Follow}_k(A)$.

Proof. Let us start from the first part of the lemma.

$\Leftarrow$ Let $\gamma = s_1 \dots s_t$ and let $p_1 \leqslant p_2 \leqslant \dots \leqslant p_t = p$ be the numbers of the layers of the nodes forming the path γ from (q, p_0) to the top layer. These numbers give a factorization $u = u_1 \dots u_t$, such that $|u_j| = p_j - p_{j-1}$.

Consider every j-th arc in the path, labelled s_j and spanning over u_j. By the assumption of the arcs' correctness, $u_j \in s_j(L^{(\ell)})$. Concatenating this for all j, one obtains $u \in \gamma(L^{(\ell)})$.

$\Rightarrow$ If $u \in s_1 \dots s_t(L^{(\ell)})$, there exists a factorization $u = u_1 \dots u_t$, such that

$$u_j \in s_j(L^{(\ell)}) \tag{13}$$

First it has to be proved that each arc s_j going from $(\delta(q, s_1 \dots s_{j-1}), p_0 + |u_1 \dots u_{j-1}|)$ to the layer $p_0 + |u_1 \dots u_{j-1} u_j|$ is correct. If $s_j = a \in \Sigma$, then (13) is sufficient for correctness. If $s_j = B \in N$, it has to be additionally demonstrated that $First_k(u_{j+1} \dots u_t v) \in \textsc{Follow}_k(B)$. By (13), $u_{j+1} \dots u_t \in s_{j+1} \dots s_t(L^{(\ell)})$. Hence, $First_k(u_{j+1} \dots u_t) \in \textsc{First}_k(s_{j+1} \dots s_r)$ by the construction of the sets $\textsc{First}_k$ (see Algorithm 1 on p. 8 and the statement of its correctness). Since $First_k(v) \in \textsc{Follow}_k(A)$ by assumption, the string $First_k(u_{j+1} \dots u_t v)$ was added to $\textsc{Follow}_k(B)$ by Algorithm 2.

If $p_0 = p$, then the correctness of all arcs in this path, together with the condition (13), implies that each of them is in the graph.

If $p_0 < p$, it is easy to see that all arcs in the path are reachable: the first of them is reachable, because (q, p_0, A) is a possible arc and hence the node (q, p_0) must be in the graph, while the rest are reachable because each of them follows a reachable arc. Similarly, the last arc of the path goes to the layer p and hence is p-reaching, while all the preceding arcs are p-reaching by the same straightforward argument. So, all arcs forming the path γ are reachable and p-reaching correct arcs that satisfy (13), and hence they must be in the graph.

Let us turn to the second part of the lemma.

$\Leftarrow$ If the condition of existence of (q, p_0, A) is satisfied, this means that there exists a rule (12), such that there is a path α_i from (q, p_0) to the layer p and $A \to \alpha_i \in R(\delta(q, \alpha_i), First_k(v))$ for every i, and at the same time the graph does not contain a path β_i from (q, p_0) to the layer p for every i. By the construction of the function R, $A \to \alpha_1 \in R(\delta(q, \alpha_1), First_k(v))$ means that $First_k(v) \in \textsc{Follow}_k(A)$, which allows us to apply the first part of the lemma to obtain $u \in \alpha_i(L^{(\ell)})$ and $u \notin \beta_i(L^{(\ell)})$. Hence, $u \in (\alpha_1 \& \dots \& \alpha_m \& \neg\beta_1 \& \dots \& \neg\beta_n)(L^{(\ell)})$.

$\Rightarrow$ Suppose that there exists a rule (12), such that

$$u \in \alpha_i(L^{(\ell)}) \quad (\text{for all } i) \tag{14a}$$

$$u \notin \beta_i(L^{(\ell)}) \quad (\text{for all } i) \tag{14b}$$

$$First_k(v) \in \textsc{Follow}_k(A) \tag{14c}$$

The latter allows us to apply the first part of the lemma to (14a) and to (14b) and thus determine the existence of the paths α_i and the non-existence of the paths β_i. In addition, (14c) implies $A \to \alpha_i \in R(\delta(q, \alpha_i), First_k(v))$ by the construction of

R. This completes the proof that the condition of existence of the arc (q, p_0, A) is satisfied. □

Sketch of a proof of Lemma 5. Let us construct such a computation. It is organized in the following way: first all correct arcs within the layer p are added, then all reachable correct arcs from the layer $p-1$ to the layer p, all reachable correct arcs from the layer $p-2$ to the layer p, and so on, ending with all correct reachable arcs from the layer 0 to the layer p.

Consider each layer p_0 $(0 \leqslant p_0 \leqslant p)$, let $u = a_{p_0+1}a_{p_0+2}\dots a_p$ and $v = a_{p+1}a_{p+2}\dots a_{|w|}$. As in Lemma 6, consider the monotone sequence of vectors corresponding to the definition of the naturally reachable solution modulo the set of substrings of u. Let $A_1, \dots, A_r$ be the sequence of the nonterminals, such that u is added to the component A_j in every j-th term of this sequence. Let $A_{j_1}, \dots, A_{j_t}$ be a subsequence of nonterminals A_{j_ℓ}, such that $v \in \textsc{Pfollow}_k(A_{j_\ell})$.

Now construct a t-step computation of the reduction phase, such that at every ℓ-th step all possible arcs of the form (q, p_0, A_{j_ℓ}) (for all relevant $q \in Q$) are added to the graph by simultaneous reductions. This is possible according to Lemma 6, which states that the condition of existence of these arcs should hold. After these t steps, the conditions of existence of these arcs still hold, and nothing can be invalidated. If any further reductions were applicable, then, by the converse of Lemma 6, the same would apply to the sequence $\{L^{(\ell)}\}$, which has presumably converged. □

The computation constructed in Lemma 5 uses reductions only, which may give a false impression that the operation of invalidation is redundant. This is not so. The composition of a model computation relies upon the knowledge of the actual languages generated by the nonterminals of the grammar, something that will not be readily available to an efficient parsing algorithm, which is supposed to determine the correct languages by itself. A real computation of the proposed algorithm involves some trial and error, represented by reductions followed by invalidations, but still constructs the same graph as in the model computation.

Lemma 7 (The computation done by the reduction phase) *Let G be a Boolean grammar without negatively fed cycles, let (Q, q_0, δ, R) be an SLR(k) automaton, let $w \in \Sigma^*$ be an input string, let $p \geqslant 0$. Suppose the graph-structured stack contains exactly the following arcs:*

- *all reachable and p-reaching correct arcs going to the layers $0, 1, \dots, p-1$,*
- *all reachable and p-reaching correct terminal arcs coming to the layer p.*

Then every computation of the reduction phase terminates and constructs a graph with exactly the following arcs:

- *all reachable and p-reaching correct arcs going to the layers $0, 1, \dots, p-1, p$,*
- *depending on the computation, some set of nonterminal arcs within the layer p that are not reachable from the source node.*

The reduction phase followed by the removal of unreachable arcs yields a graph consisting of exactly all reachable and p-reaching correct arcs going to the layers $0, 1, \dots, p-1, p$.

This explains the operation of the reduction phase. Let us now determine what does the shift phase do.

Lemma 8 (Existence of a terminal arc) *Suppose a nonterminal arc A from a node (q_0, p_0) to a layer p' is correct and reachable. Then for every layer p ($p_0 \leqslant p < p'$) there exists a state q and a reachable correct terminal arc a_{p+1} from (q, p) to the layer $p+1$, such that there is a path from (q_0, p_0) to (q, p) formed by correct arcs.*

Sketch of a proof. Since the A-arc in question is correct, it is known that $u = a_{p_0+1} \dots a_{p'} \in L_G(A)$. Let ℓ be the least number of steps in the monotone sequence in the definition of naturally reachable solution, in which the string u is added to the component A. Induction on the lexicographically ordered pairs $(p' - p_0, \ell)$.

Consider the rule $A \to \alpha_1 \& \dots \& \alpha_m \& \neg\beta_1 \& \dots \& \beta_n$, such that $u \in \alpha_i(L^{(\ell-1)})$ and $u \notin \beta_i(L^{(\ell-1)})$. Let us use the first conjunct of this rule, which must be positive, and denote $\alpha_1 = s_1 \dots s_t$. Then there exists a factorization $u = u_1 \dots u_t$, such that $u_j = s_j(L^{(\ell-1)})$.

By the first part of Lemma 6, there is a path α formed by correct arcs that connects (q_0, p_0) to the layer p. Let j be the uniquely defined number, such that a_{p+1} is a part of u_j. If $s_j = a_{p+1}$, this arc s_j is the arc sought. If $s_j = B \in N$ then this B-arc is correct and reachable, and the induction hypothesis can be used as follows: if u_j is a proper substring of u, then the number of layers spanned by B is less than the number of layers spanned by A; if $u_j = u$, then $u \in B(L^{\ell-1})$ and the induction hypothesis is again applicable. In both cases, the existence of the terminal arc is deduced. □

Lemma 9 *Let the graph contain exactly all reachable and p-reaching correct arcs up to the layer p. Then the shift phase from the layer p to the layer $p+1$ constructs a graph with the following arcs:*

- *all reachable and $(p+1)$-reaching correct arcs to the layers up to p;*
- *all reachable and $(p+1)$-reaching correct terminal arcs to the layer $p+1$.*

Proof. Every terminal arc added in the shift phase is labelled with a_p and goes from (q, p) to $(\delta(q, a_{p+1}), p+1)$. It is correct, because $a_p \in L_G(a_{p+1})$. It is reachable, because the node (q, p) must have an incoming arc, while all arcs in the original graph are known to be reachable. It is $(p+1)$-reaching, because it goes to the layer $p+1$.

Conversely, consider any correct reachable terminal arc going to the layer $p+1$. Then it goes from a node (q, p), which has an incoming reachable correct arc, and $\delta(q, a_{p+1}) \neq -$. By the definition of the shift phase, the node (q, p) is eventually considered, and the arc a_{p+1} from this node to the layer $p+1$ is added.

Let us now prove the claim on the handling of the nonterminal arcs during the shift phase. No new nonterminal arcs are added during the shift phase, but some of them can get removed: these must be exactly the arcs that are p-reaching, but not $(p+1)$-reaching. Every nonterminal arc that remains in the graph is a reachable correct arc by assumption. It is $(p+1)$-reaching, because it is continued by a terminal arc going to the layer $p+1$.

Conversely, if a reachable correct nonterminal arc A is $(p+1)$-reaching, then it could be continued by a sequence of correct arcs leading to a certain correct arc labelled s that goes to the layer $p+1$. If $s = B \in N$, then, according to Lemma 8, there must be a path from the original arc A to a correct terminal arc from the layer p to the layer $p+1$. Then this terminal arc will be added during the shift phase, and hence the nonterminal arc in question will not be removed. If $s \in \Sigma$, the arc A will not be removed by the same reason. □

These lemmata can be combined to prove the algorithm's correctness.

Theorem 2 *Let G be a Boolean grammar without negatively fed cycles, let (Q, q_0, δ, R) be the SLR(k) automaton for G, let $w \in \Sigma^*$ be an input string. Then the generalized LR algorithm constructed with respect to G and (Q, δ, R) terminates on w, and:*

- *After i iterations of the main loop, the graph-structured stack contains all reachable and i-reaching correct arcs to the layers $0, 1, \ldots, i$ (see Definition 13).*
- *The input string is accepted if and only if $w \in L(G)$.*

Proof. The first claim is proved using an induction on i, the number of iterations of the loop.

Basis: $i = 0$**.** When the algorithm starts, there are no arcs in the graph, and hence it satisfies the condition of Lemma 7 with $p = 0$. Thus the reduction phase followed by the removal of unreachable arcs leads to a graph of all reachable and 0-reaching correct arcs within the layer 0. This is the state in which the algorithm enters its main loop.

Induction step: $i \to i+1$**.** Suppose the graph contains all reachable and i-reaching correct arcs, and consider the computation of the $(i+1)$-th iteration of the algorithm's main loop.

By Lemma 9, after the shift phase the graph contains all reachable and $(i+1)$-reaching correct arcs to the layers up to i, and all reachable and $(i+1)$-reaching correct terminal arcs to the layer $i+1$. This allows us to apply Lemma 7 to conclude that the reduction phase followed by the removal of unreachable arcs gives a graph consisting of all reachable and $(i+1)$-reaching correct arcs.

Thus it has been proved that the outer loop terminates and the final graph contains exactly all reachable and $|w|$-reaching correct arcs.

Consider the acceptance condition. First, there is a case when the algorithm prematurely terminates if all nodes in the top layer fail to shift during the shift phase in some i-th layer. It has to be proved that this implies $w \notin L(G)$. Suppose the contrary, $w \in L_G(S)$. Then the arc S from $(q_0, 0)$ to $(\delta(q_0, S), |w|)$ is a reachable correct arc, and hence, by Lemma 8, there exists a reachable correct terminal arc a_{i+1} from the layer i to the layer $i+1$. This arc should have been added at the mentioned shift phase. The fact it was not forms a contradiction.

	δ			R	
	a	S	A	ε	a
0	1	5	2	$A \to \varepsilon$	
1	1	3	4	$A \to \varepsilon$	
2				$S \to A$	
3				$A \to aS$ $A \to \neg aS$	
4				$S \to A$ $A \to \neg aA$ $A \to aA$	
5					

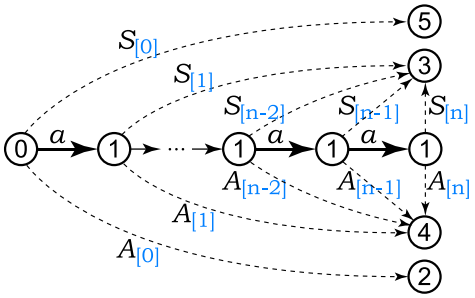

Figure 8: LR table in Example 6; the possible arcs at the last reduction phase.

Suppose the algorithm accepts the string. This can only happen if all iterations of the main loop are successfully completed and there is an arc S from $(q_0, 0)$ to $(\delta(q_0, S), |w|)$. Since this arc is correct, this implies $w \in L_G(S) = L(G)$.

Conversely, suppose $w \in L(G) = L_G(S)$. Since $\varepsilon \in \textsc{Follow}_k(S)$, the condition of the correctness of the arc S from $(q_0, 0)$ to $(\delta(q_0, S), |w|)$ is satisfied. This arc is reachable, because it starts from the source node, and it is $|w|$-reaching, since it goes to the layer $|w|$. Hence, this arc will be in the graph constructed by the algorithm, and therefore the algorithm accepts the string w. □

8. Implementation and complexity

The overall composition of the algorithm has been defined in Section 4, but an important detail of its implementation has been intentionally omitted: it has not been defined how actions are selected at the reduction phase. Instead, the algorithm was proved correct for any possible sequence of valid actions. Though the order of actions does not influence the result of the computation, it certainly has an impact on the complexity of the algorithm and needs to be defined to consider the algorithm complete.

The most natural method would be to do reductions and invalidation in series, that is, one operation at once. Though this will probably work fine in practice, there exists an example, in which one of the possible computations takes exponential time.

Example 6 *Consider a Boolean grammar over the alphabet* $\{a\}$ *that contains the following rules:*

$$S \to A$$
$$A \to aS\&\neg aA \mid \neg aS\&aA \mid \varepsilon$$

The grammar, which generates $\{\varepsilon\}$*, has no negatively fed cycles, and its LR parsing table is given in Figure 8. However, for every string* a^n *there exists a computation of the last reduction phase that takes at least* 2^{n+1} *steps.*

Proof. Like in Example 3, the algorithm first shifts the entire input a^n to the stack and then enters the final reduction phase. The possible arcs in the stack are given in

Figure 8, and are exactly as in Example 3. Following are the conditions of existence for each arc, where $\oplus$ denotes sum modulo two:

$$\begin{array}{lcll} S_{[i]} & = & A_{[i]} & (0 \leqslant i \leqslant n) \\ A_{[i]} & = & S_{[i+1]} \oplus A_{[i+1]} & (0 \leqslant i < n) \\ A_{[n]} & = & 1 & \end{array}$$

It is easy to see that if $A_{[n]}$ and $S_{[n]}$ are added by two consecutive reductions, then the final configuration is obtained. By Theorem 4, any other course of the reduction phase leads to the same result. Let us construct a different sequence of configurations that computes this result in no fewer than 2^{n+1} steps.

For every number k ($0 \leqslant k < 2^{n+1}$), denote each i-th digit in its binary notation by x_i, and define the configuration of possible arcs corresponding to k as follows:

$$\begin{array}{lcll} S_{[i]} & = & x_i \oplus x_{i+1} & (0 \leqslant i \leqslant n) \\ A_{[i]} & = & \overline{x_{i+1}} & (0 \leqslant i \leqslant n) \end{array} \tag{15}$$

Obviously, different numbers correspond to different configurations.

The configuration corresponding to the number 0 contains all the A-arcs and none of the S-arcs; it can be reached by doing $n+1$ reductions by $A_{[n-1]}, \ldots, A_{[0]}$, exactly in this order.

Consider any two configurations corresponding to consecutive numbers k and $k+1$, where $k+1 < 2^{n+1}$. Let j be the number of the least significant zero bit in the binary notation of k, so that $x_0 = \ldots = x_{j-1} = 1$ and $x_j = 0$; note that $j < n$, since $k < 2^{n+1} - 1$. Denote each i-th bit in the binary notation of $k+1$ by y_i. It is easy to see that $y_0 = \ldots = y_{j-1} = 0$, $y_j = 1$ and $y_i = x_i$ for all $i > j$.

By (15), the status of arcs in the configuration corresponding to k is as follows: $S_{[0]} = \ldots = S_{[j-2]} = 0$, $S_{[j-1]} = 1$, $S_{[j]} = x_{j+1}$, $A_{[0]} = \ldots = A_{[j-2]} = 0$, $A_{[j-1]} = 1$ and $A_{[j]} = \overline{x_{j+1}}$. The configuration corresponding to $k+1$ has the following arcs: $S'_{[0]} = \ldots = S'_{[j-2]} = 0$, $S'_{[j-1]} = 1$, $S'_{[j]} = x_{j+1}$, $A'_{[0]} = \ldots = A'_{[j-2]} = 0$, $A'_{[j-1]} = 1$ and $S'_{[j]} = \overline{x_{j+1}}$, while the rest of the arcs are the same as in the previous configuration, since $y_i = x_i$ for all $i > j$. These bit vectors are given in the following table:

i	0	$\ldots$	$j-2$	$j-1$	j	$j+1$	$\ldots$
x_i	1	$\ldots$	1	1	0	x_{j+1}	$\ldots$
y_i	0	$\ldots$	0	0	1	x_{j+1}	$\ldots$
$S_{[i]}$	0	$\ldots$	0	1	x_{j+1}	$x_{j+1} \oplus x_{j+2}$	$\ldots$
$S'_{[i]}$	0	$\ldots$	0	1	$\boldsymbol{\overline{x_{j+1}}}$	$x_{j+1} \oplus x_{j+2}$	$\ldots$
$A_{[i]}$	0	$\ldots$	0	1	$\overline{x_{j+1}}$	$\overline{x_{j+2}}$	$\ldots$
$A'_{[i]}$	**1**	$\ldots$	**1**	**0**	$\overline{x_{j+1}}$	$\overline{x_{j+2}}$	$\ldots$

The transition from the k-th to the $(k+1)$-th configuration can be done as follows. First, the possible arc $S_{[j]}$ is negated: added by a reduction by $S \to A$, if $x_{j+1} = 0$, or removed by an invalidation otherwise; this can be done because $S_{[j]} \neq A_{[j]}$. Next, the condition of existence of the arc $A_{[j-1]}$ becomes violated, and it is removed by an invalidation. This in turn allows us to add the following arcs,

one by one: $A_{[j-2]}$, $A_{[j-3]}$, ..., $A_{[0]}$; every next arc in this list depends upon the previous one. Then the $(k+1)$-th configuration is obtained.

We have constructed a sequence of over 2^{k+1} distinct configurations that can be visited one after another in a single reduction phase. Therefore, this computation contains at least 2^{k+1} steps, which proves the correctness of Example 6. □

Since doing one action at a time can result in an exponentially long computation, a different implementation is needed to ensure polynomial time complexity. The proposed method is based upon doing all possible actions at every step of the reduction phase, which reduces the total number of elementary actions to $O(n^2)$. Let us combine this method with some straightforward breadth-first search techniques in the graph to obtain a complete implementation of the algorithm, the complexity of which can be analyzed.

Definition 14 *Consider any contents of the graph-structured stack. For each vertex v and for each number $\ell \geqslant 0$, let $predecessors_\ell(v)$ be the set of vertices that are connected to v with a path that is exactly ℓ arcs long.*

This set can be computed iteratively on ℓ by taking $predecessors_0(v) = \{v\}$, and then constructing $predecessors_{\ell+1}(v)$ as the set of all vertices v', such that there is an arc from v' to some $v'' \in predecessors_\ell(v)$. Now the algorithm for doing the reduction phase reads as follows:

while the graph can be modified
    // Conjunct gathering
    let $x[\,]$ be an array of sets of vertices, indexed by conjuncts.
    for each node $v = (q, p_{top})$ in the top layer
        for each $A \to \alpha \in R(q, u)$
            $x[A \to \alpha] = x[A \to \alpha] \cup predecessors_{|\alpha|}(v)$

    // Reductions
    let *valid* be a set of arcs, initially empty
    for each rule $A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P$
        for each node $v \in \bigcap_{i=1}^{m} x[A \to \alpha_i] \setminus \bigcup_{i=1}^{n} x[A \to \beta_i]$
            if v is not connected to the top layer by an arc labelled A, **then**
                add an arc from v to the top layer labelled A
            $valid = valid \cup \{$the arc from v to the top layer labelled $A\}$

    // Invalidations
    for each arc (v', v) going to the top layer and labelled A
        if this arc is not in the set *valid*, **then**
            remove the arc from the graph

end while

At the stage of conjunct gathering, the algorithm scans the stack and determines, which reductions and invalidations can possibly be done. The set $x[\,]$ of gathered conjuncts stores this information, which reflects the state of the graph at the time of conjunct gathering. Then these operations are applied sequentially on the basis of the gathered conjuncts. This ensures that all reductions and invalidations are performed independently.

Let us give an upper bound for the complexity of this implementation of the reduction phase, and of the algorithm as a whole.

Lemma 10 *The above algorithm for the reduction phase performs $O(n^2)$ reductions and invalidations, using time $O(n^3)$.*

Sketch of a proof. The number of iterations is $O(n)$, because the dependencies of the possible arcs upon each other are $O(n)$ deep, and hence $O(n)$ parallel applications of all possible reductions and invalidations are sufficient: this has been stated in Lemma 1.

The conjunct gathering stage computes $predecessors_\ell$ a constant number of times, and ℓ is also bounded by a constant. Computing the set of immediate predecessors involves considering $O(n)$ nodes, each of which has $O(n)$ predecessors, and thus each conjunct gathering stage takes $O(n^2)$ steps. The subsequent reduction and invalidation stages take $O(n)$ time. Therefore, each iteration of the *while* loop terminates in time $O(n^2)$.

It can be concluded that the algorithm for doing the reduction phase terminates in at most $C \cdot n^3$ steps. □

The cubic upper bound for the complexity of the reduction phase yields an $O(n^4)$ upper bound for the whole algorithm.

Theorem 3 *Under the conditions of Theorem 2 and using the above implementation on a random-access machine, the generalized LR parsing algorithm, given an input string of length n, performs $O(n)$ shifts, $O(n)$ local errors, $O(n^3)$ reductions and $O(n^3)$ invalidations. The number of elementary operations on a random access machine is $O(n^4)$, and the memory used is $O(n^2)$.*

This upper bound for the complexity of the given implementation is actually precise, as witnessed by the following grammar:

Example 7 *Consider the following one-nonterminal Boolean grammar over the alphabet $\{a\}$ that generates the language $\{a\} \cup (aa)^+$:*

$$S \to SS\&\neg aS \mid aa \mid a$$

Using the above implementation, the generalized LR parsing algorithm executed on the string a^n performs $C \cdot n$ shifts, $C' \cdot n^3$ reductions and $C'' \cdot n^3$ invalidations. The number of elementary operations is $C''' \cdot n^4$, and quadratic space is used.

Whether the worst-case execution time can be improved to $O(n^3)$ or not, remains unknown. Under the current implementation, the bottleneck is the procedure of conjunct gathering: as mentioned in Lemma 10, it uses $C \cdot n^2$ time, because this is the time needed to compute the list of predecessors of a given set of vertices. If one could organize a more efficient computation of this list, the performance of the algorithm could be improved.

For instance, for conjunctive grammars there exists a method of doing the reduction phase in time $O(n^2)$ [10], which is based upon maintaining data structures for the sets $predecessors_\ell(v)$, for each vertex v in the graph and for all numbers ℓ, such that $0 \leqslant \ell \leqslant \max_{A \to \pm\gamma \in conjuncts(G)} |\gamma|$. Every time a new arc (v, v') is added to the graph, the predecessors of v are inherited by v' and by all successors of v'.

This has two consequences: on one hand, conjunct gathering is done in time $O(n)$, because instead of computing $predecessors_\ell(v)$ one can just take the value from the memory; on the other hand, the addition of every arc can no longer be done in time $O(1)$, and $O(n)$ operations are necessary to update the new data structures. However, since in the conjunctive case every arc is added only once and is never removed, there will be at most $C \cdot n^2$ arc additions, and thus the time spent maintaining stored values of $predecessors_\ell(v)$ sums up to $O(n^3)$ for the entire algorithm. This shows that the Generalized LR algorithm for conjunctive grammars can be made to work in time $O(n^3)$ [10].

The above approach will not work in the case of Boolean grammars for two reasons. First, it is not clear how to update the set $predecessors_\ell(v)$ for invalidations: when an arc (v_0, v) is removed, but the vertex v has other incoming arcs, how can the algorithm efficiently determine what should be removed from $predecessors_\ell(v)$? Second, the number of reductions need not be $O(n^2)$ any longer, and thus if the addition of an arc requires time $C \cdot n$, this will become the new bottleneck of the algorithm, which will subsequently require at least $C \cdot n^4$ time. But perhaps one could still invent a way to overcome these difficulties and design a square-time implementation of the reduction phase.

9. On parsing schemata for Boolean grammars

From an implementation point of view, the algorithm is very similar to its context-free prototype, the Generalized LR [19]. However, a quick glance at the proof of its correctness shows that its mathematical justification is entirely different from that of any context-free parsing algorithms. Let us try to consider the given algorithm in terms of the theory of *parsing schemata*, developed by Sikkel [17], which represents the main ideas of context-free parsing.

A *parsing schema* consists of a set of elementary propositions, called *items*, a set of items assumed to be true, called *axioms*, and a set of inference rules, which are used to deduce the truth of other items [17] in line with the principle of *parsing as deduction* proposed by Shieber, Schabes and Pereira [16]. In the case of Boolean LR parsing, the items are the arcs in the stack, and the parser manipulates them as bits. However, the bits are not only set, but also reset and set back again, which certainly does not fit into the deductive paradigm.

The given algorithm actually conducts a search for a solution of a system of Boolean equations. The latter description is also true with respect to any context-free parsing schema, though the system of Boolean equations is monotone in this case, and hence the search for a solution degrades to a search for a least fixed point, which can be done using the corresponding methods. The fact that quite a representative context-free parsing algorithm, the generalized LR, was extended for the nonmonotone case demonstrates that the context-free parsing theory does not absolutely depend upon the monotonicity and the underlying fixed point techniques, and perhaps more ideas from the context-free parsing would hold in the more general context of underlying Boolean equations. Extending the framework of parsing schemata to the nonmonotone case, which would allow one to analyze such

algorithms as the one described in this paper, is proposed as an interesting research problem.

10. Conclusion

The first practically useful parsing algorithm for Boolean grammars has been developed. It is as easy to implement and use as its prototypes [19, 10], and performs as efficiently. Taking into account the greater expressive power of Boolean grammars, this is a good argument to use Boolean grammars instead of context-free grammars in those numerous applications, where the use of context-free generalized LR parsing is considered justified. In particular, the new algorithm is a mathematically well-defined alternative to simulating negation in the context-free LR by the means of ad hoc kludges.

Designing an efficient implementation of the given algorithm, or inventing its more efficient variant, is proposed as a research problem. It would be interesting to study whether any of the advanced practical techniques developed for the context-free case [3, 6, 8] could be extended for the Boolean case. In particular, could the generalized LR algorithm for Boolean grammars be implemented to work in time $O(n^3)$?

It is worth mention that the given algorithm has been used to parse a simple programming language specified entirely by a Boolean grammar [14]; the automatically generated parser worked in quadratic time [11]. Though this example is certainly far from being practically applicable, it nevertheless shows that the combination of Boolean grammars and generalized LR parsing can be used for quite nontrivial tasks, such as ensuring declaration before use and checking scope rules for variables. This gives a further evidence that this theoretically defined family of grammars is suitable for practical use.

Acknowledgements

I would like to thank Kai Salomaa for his helpful comments on the first revisions of this paper, which formed a part of my Ph.D. thesis. I am grateful to Oksana Yakimova for suggesting the first example of exponential time [21], which can be reached on the following grammar:

$$S \to aS\&aaS \mid A\&\neg aS\&\neg aaS$$
$$A \to Aa \mid \varepsilon$$

though the proof of that is not as easy as the proof of Example 6. Thanks are due to the referees for careful reading and for good advices regarding the presentation. In the end, I wish to thank a former colleague for putting me, in August 2001, in a perfect environment for inventing and implementing this algorithm.

References

1. A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

2. J. Aycock, “Why Bison is becoming extinct”, *ACM Crossroads*, 7 (2001).
3. J. Aycock, R. N. Horspool, J. Janousek, B. Melichar, “Even faster generalized LR parsing”, *Acta Informatica*, 37:9 (2001), 633–651.
4. M. van den Brand, J. Scheerder, J. J. Vinju, E. Visser, “Disambiguation filters for scannerless generalized LR parsers”, *Compiler Construction* (CC 2002, Grenoble, France, April 8–12, 2002), LNCS 2304, 143–158.
5. S. Ginsburg, H. G. Rice, “Two families of languages related to ALGOL”, *Journal of the ACM*, 9 (1962), 350–371.
6. J. R. Kipps, “GLR parsing in time $O(n^3)$”, in: M. Tomita (Ed.), *Generalized LR Parsing*, Kluwer, 1991, 43–59.
7. D. E. Knuth, “On the translation of languages from left to right”, *Information and Control*, 8 (1965), 607–639.
8. S. McPeak, G. C. Necula, “Elkhound: a fast, practical GLR parser generator”, *Compiler Construction* (CC 2004, Barcelona, Spain, March 29–April 2, 2004), LNCS 2985, 73–88.
9. A. Okhotin, “Conjunctive grammars”, *Journal of Automata, Languages and Combinatorics*, 6:4 (2001), 519–535.
10. A. Okhotin, “LR parsing for conjunctive grammars”, *Grammars* 5 (2002), 81–124.
11. A. Okhotin, “Whale Calf, a parser generator for conjunctive grammars”, *Implementation and Application of Automata* (CIAA 2002, Tours, France, July 3–5, 2002), LNCS 2608, 213–220.
12. A. Okhotin, “Decision problems for language equations with Boolean operations”, *Automata, Languages and Programming* (ICALP 2003, Eindhoven, The Netherlands, June 30–July 4, 2003), LNCS 2719, 239–251.
13. A. Okhotin, “Boolean grammars”, *Information and Computation*, 194:1 (2004), 19–48.
14. A. Okhotin, “On the existence of a Boolean grammar for a simple programming language”, *Proceedings of AFL 2005* (May 17–20, 2005, Dobogókő, Hungary).
15. D. J. Salomon, G. V. Cormack, “Scannerless NSLR(1) parsing of programming languages”, *SIGPLAN Notices*, 24:7 (1989), 170–178.
16. S. M. Shieber, Y. Schabes, F. C. N. Pereira, “Principles and implementation of deductive parsing”, *Journal of Logic Programming*, 24 (1995), 3–36.
17. K. Sikkel, *Parsing Schemata*, Springer-Verlag, 1997.
18. M. Tomita, *Efficient Parsing for Natural Language*, Kluwer, 1986.
19. M. Tomita, “An efficient augmented context-free parsing algorithm”, *Computational Linguistics*, 13:1 (1987), 31–46.
20. M. Wrona, “Stratified Boolean grammars”, *Mathematical Foundations of Computer Science* (MFCS 2005, Gdansk, Poland, August 29–September 2, 2005), LNCS 3618, 801–812.
21. O. S. Yakimova, personal communication, July 2005.