# A preliminary report on generalized LR parsing for Boolean grammars

Alexander Okhotin
okhotin@cs.queensu.ca

Technical report 2004–486

School of Computing, Queen's University,
Kingston, Ontario, Canada K7L 3N6

July 2004

**Abstract**

The generalized LR parsing algorithm for context-free grammars, invented by Tomita in 1986, is extended for the case of Boolean grammars – a recently introduced generalization of context-free grammars with logical connectives added to the formalism of rules. A high-level description of the algorithm, an elaborate example of its operation and a suggested implementation are provided. The algorithm has been implemented in a parser generator. However, the proof of the algorithm's correctness is so far incomplete, hence the preliminary status of this report.

## 1 Introduction

Knuth's discovery of the LR parsing algorithm [4] in 1965 became one of the most significant contributions of formal language theory to software engineering. Being applicable to every deterministic context-free language and working in linear time, this algorithm possessed exactly the qualities in demand by the compiler industry, which ensured quick recognition and continued work in this direction.

For technical details on Knuth's LR the reader is directed to the well-known textbook on compilers by Aho, Sethi and Ullman [1]. In short, the algorithm uses stack memory, which contains a string $\alpha$ of terminals and nonterminals and the computation history of a certain DFA on $\alpha$. The algorithm performs actions of two types:

- *Shift*, which means reading the next symbol of the input string, pushing it onto stack and consuming it.

- *Reduce*, which pops the symbols forming the righthand side $\alpha$ of some rule $A \to \alpha \in P$ off the stack and then pushes $A$ onto the stack.

By the names of these actions, LR parsing is nicknamed *Shift–Reduce* parsing. The choice of action to apply at every step of the computation is determined by the contents of the stack and by $k$ lookahead symbols, accounted by the means of a parsing table. If the table unambiguously specifies the action to perform in any possible situation, the LR($k$) can proceed deterministically and do the parsing in linear time. The grammars for which such a table can be constructed is called LR($k$), and the language family covered by such grammars, for any $k \geqslant 1$, is the family of deterministic context-free languages.

If one attempts to construct an LR($k$) parser for a grammar that is not LR($k$), one obtains so-called *LR conflicts* – ambiguously defined entries of the parsing table, which correspond to the situations where the correct action to perform cannot be determined out of $k$ lookahead symbols and the contents of the stack. In such a situation the parser has to guess whether it should *shift* or *reduce*, and by which rule should it reduce if there are several candidates. Although this nondeterminism could be simulated by backtracking, that would yield worst-case exponential time, which is unacceptable.

A polynomial-time algorithm to simulate nondeterminism in LR was given by Tomita [11, 12]. Whenever a standard deterministic LR($k$) has to make a choice between two or more actions (Shift or Reduce), Tomita's parser, named *Generalized LR* by its author, performs both actions at the same time, storing all possible contents of the LR parser stack in the form of a graph. Although the number of possible computations of a nondeterministic LR parser can depend exponentially on the length of the string, the graph-structured stack never contains more than $O(n)$ vertices and therefore always fits in $O(n^2)$ memory. A lot of improvements and refinements of the original Tomita's algorithm have been proposed since it was introduced [2, 13]; in particular, it was shown that if the most efficient graph search algorithms are employed, then the algorithm can be made to work in time at most cubic of the length of the input. Also this algorithm has been adapted for conjunctive grammars by the author [6].

In this paper, this algorithm is further extended to handle the case of Boolean grammars [8]. This results in a method general enough, so that the conjunctive LR [6], Tomita's context-free generalized LR [12] and Knuth's deterministic LR [4] become its simple subcases. This makes the new algorithm technically quite complicated, yet computationally still feasible: it can be implemented to work in no more than cubic time on conjunctive and context-free grammars (by the virtue of performing exactly the same computation as the more specialized algorithms [12, 6] would perform), and in linear time on context-free LR($k$) grammars, while for Boolean grammars of the general form it works in cubic time under not at all restrictive assumptions upon a grammar.

## 2 Boolean grammars

Boolean grammars are context-free grammars with explicit *conjunction* and *negation* operations in the formalism of rule.

**Definition 1 ([8]).** *A Boolean grammar [8] is a quadruple $G = (\Sigma, N, P, S)$, where $\Sigma$ and $N$ are disjoint finite nonempty sets of terminal and nonterminal symbols respectively; $P$ is a finite set of rules of the form*

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

*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 $A \to \neg\gamma$".*

A Boolean grammar is called a conjunctive grammar [5], if negation is never used, i.e., $n = 0$ for every rule (1). It degrades to a context-free grammar if neither negation nor conjunction are allowed, i.e., $m = 1$ and $n = 0$ for all rules.

Defining the semantics of Boolean grammars – i.e., the language a grammar generates – turned out to be rather nontrivial [8]. The definition is based upon language equations with concatenation and all set-theoretic operations, roughly similar to the well-known characterization of the context-free grammars by a simpler class of language equations with union and concatenation only [3].

**Definition 2.** *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar. The system of language equations associated with $G$ is a resolved system of language equations over $\Sigma$ in variables $N$, in which the equation for each variable $A \in N$ is*

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

*A vector $L = (L_1, \ldots, L_n)$ is called a naturally reachable solution of (2) if for every finite $M \subseteq \Sigma^*$ closed under substring 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.*

In all practical cases (which exclude artistry like using a nonterminal $A$ with a rule $A \to \neg A \& B$), a system associated with a grammar will have a naturally reachable solution $(L_1, \ldots, L_n)$, and the language $L_G(A_i)$ generated by every $i$-th nonterminal $A$ can be defined as $L_i$ then. The language of the grammar is $L(G) = L_G(S)$.

Following is a Boolean grammar that generates the well-known non-context-free language $\{ww \mid w \in \{a,b\}^*\}$:

$$\begin{array}{ll} S \to \neg AB \& \neg BA \& C & A \to aAa \mid aAb \mid bAa \mid bAb \mid a \\ C \to aaC \mid abC \mid baC \mid bbC \mid \varepsilon & B \to aBa \mid aBb \mid bBa \mid bBb \mid b \end{array}$$

# 3 A restricted case of Boolean grammars

Define the following transform of a Boolean grammar: given a grammar $G = (\Sigma, N, P, S)$, consider the set of rules

$$positive(P) = \{A \to \alpha_1 \& \ldots \& \alpha_m \mid A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P\} \tag{5}$$

Using (5) as the set of rules of a new grammar, a grammar $positive(G) = (\Sigma, N, positive(P), S)$ is obtained. It is a conjunctive grammar [5].

**Definition 3.** *Let $G$ be a Boolean grammar. A sequence of conjuncts*

$$\begin{array}{rcl} A_1 & \to & \pm\eta_1 A_2 \theta_1, \\ A_2 & \to & \pm\eta_2 A_3 \theta_2, \\ & \vdots & \\ A_\ell & \to & \pm\eta_\ell A_{\ell+1} \theta_\ell, \end{array} \tag{6}$$

*such that $\ell \geqslant 1$, $\varepsilon \in L_{positive(G)}(\eta_i)$ and $\varepsilon \in L_{positive(G)}(\theta_i)$ for all $i$, is called a chain from $A_1$ to $A_{\ell+1}$. A cycle is a chain from a nonterminal to itself.*

*If the condition $\varepsilon \in L_{positive(G)}(\eta_i)$ is lifted, while the rest of the conditions remain (including the requirement that $\varepsilon \in L_{positive(G)}(\theta_i)$ for all $i$), the sequence (6) is called a right-chain from $A_1$ to $A_{\ell+1}$.*

A chain (or a right-chain) from $A$ to $B$ means a certain dependency of $A$ on $B$ – or, in other words, an influence of $B$ on $A$.

**Definition 4.** *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 loop negatively fed by a right-chain will be referred to as just a negatively fed loop.*

The following grammar has a cycle (a chain from $A$ to $A$, going through $T$ and $S$), which is negatively fed by a right-chain from $A$ to $B$, where a rule for $B$ has a negative conjunct:

$S \to A$
$A \to T \mid X$
$T \to S$
$X \to aB$
$B \to \varepsilon \& \neg E$
$E \to \varepsilon$

The dependence of nonterminals on each other is illustrated in Figure 1.

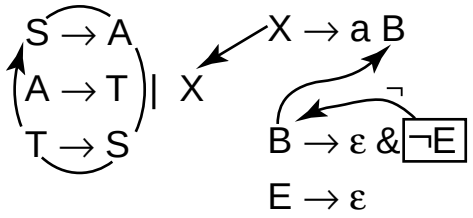


Figure 1: How a cycle is negatively fed.

Note that the associated system of language equations has a naturally reachable solution $(\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \{\varepsilon\})$. Hence this is a well-formed Boolean grammar that generates the language $\varnothing$. However, this grammar and any other grammars with negatively fed loops are problematic for the LR parsing algorithm described below.

In the following it will be assumed that Boolean grammars used with the algorithm have no negatively fed loops. This is a very weak condition. For instance, every Boolean grammar in the binary normal form [8] does not have any chains at all, not to mention any loops, especially negatively fed ones. Since every Boolean grammar can be effectively transformed to the binary normal form, this requirement causes no loss of generality.

The absense of negatively fed loops is a syntactical condition that does not rely upon the semantics of Boolean grammars. The following useful result can be proved:

**Theorem 1.** *Every Boolean grammar without negatively fed cycles complies to the semantics of naturally reachable solution.*

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

**Definition 5.** *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.

**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*

$$(b_1^{(0)}, \ldots, b_n^{(0)}), (b_1^{(1)}, \ldots, b_n^{(1)}), \ldots, (b_1^{(j)}, \ldots, b_n^{(j)}), \ldots, \tag{7}$$

*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 $b^{(j+1)_{t_i}} = p_{t_i}(b_1^{(j)}, \ldots, b_\ell^{(j)})$ (for $1 \leqslant t_i \leqslant \ell$) $b^{(j+1)_{t_i}} = f_{t_i}(b_1^{(j)}, \ldots, b_{t_i-1}^{(j)})$ (for $\ell < t_i \leqslant n$) 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 sets of positions at each step, this sequence converges to a solution of this system of Boolean equations 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.*

*Proof of Theorem 1.* For all $M, w$ and $L \pmod M$ as in Definition 2, 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.** $M = \varnothing$, $w = \varepsilon$.

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

$$conjunct(s_1 \ldots s_k) = \begin{cases} x_{s_1} \wedge \ldots \wedge x_{s_k}, & \text{if } \varepsilon \in L_{positive(G)}(s_i) \text{ for all } i \\ 0 & \text{otherwise} \end{cases} \tag{8b}$$

Let the equation for a variable $x_A$ refer to a variable $x_B$ in its right hand side; this implies $\varepsilon \in L_{positive(G)}(s_i)$ for all $i$, and hence a conjunct $A \to \pm\eta B\theta$, such that $\varepsilon \in L_{positive(G)}(\eta)$ and $\varepsilon \in L_{positive(G)}(\theta)$. Hence, every chain in the dependencies (8) implies a chain in the grammar in the sense of Definition 3, and every negatively fed cycle in (8) 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 (8) has no negatively fed cycles. Therefore, Lemma 1 holds with respect to (8), and the condition of Definition 2 is satisfied.

**Induction step.** Let $L_M$ be the naturally reachable solution modulo $M$. Construct a system of Boolean equations as follows:

$$x_A = \bigvee_{A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P} \left( \bigwedge_{i=1}^{m} conjunct(\alpha_i) \ \wedge \ \bigwedge_{i=1}^{m} \neg conjunct(\beta_i) \right) \tag{9a}$$

$$conjunct(s_1 \ldots s_k) = \begin{cases} 1, & \text{if } w \in s_1 \ldots s_k(L_M) \\ \bigvee\limits_{i: \varepsilon \in L_M(s_j) \text{ for all } j \neq i} x_{s_k} & \text{otherwise} \end{cases} \tag{9b}$$

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

# 4 The parsing table

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 $\Sigma^{\leqslant k}$, 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 multitude of different table construction techniques for Knuth's deterministic algorithm, applicable to slightly different classes of grammars and yielding tables of different size. In the case of Tomita's nondeterministic algorithm, the difference between these methods is not very essential, and the simplest of them, SLR(1) [1], is typically used. Let us adapt the context-free SLR($k$) table construction method 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* dc$(P)$ *denote the (finite) set of all dotted conjuncts.*

Let the set of states be $Q = 2^{\text{dc}(P)}$. 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 classical context-free LR theory, 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

$$goto(X, s) = \{A \to \alpha s \cdot \beta \mid A \to \alpha \cdot s\beta \in X\} \tag{10}$$

$closure(X)$ is defined as the minimal 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 $q_0 = closure(\{S \to \cdot\sigma \mid S \to \pm\sigma \in conjuncts(P)\})$, while the transition from a state $q \subseteq$ dc$(P)$ by a symbol $s \in \Sigma \cup N$ is defined as follows: $\delta(q, s) = closure(goto(q, s))$. The state $\varnothing \subseteq$ 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 "ACTION$[q, a]$ = Shift $q'$", while $\delta(q, A) = q'$ ($A \in N$) means "GOTO$(q, A) = q'$".

An important property of this DFA is that each state has unique accessing string – in the sense that if $\delta(q_1, w_1) = \delta(q_2, w_2)$ and $|w_1| = |w_2|$, then $w_1$ and $w_2$ coincide. Hence for every state $q$ and for every number $n$, if $q$ has an $n$-symbol accessing string, this string is uniquely defined.

Let $\Sigma^{\leqslant k} = \{w \mid w \in \Sigma^*, |w| \leqslant k\}$. For any string $w$, define

$$First_k(w) = \begin{cases} w, & \text{if } |w| \leqslant k \\ \text{first } k \text{ symbols of } w, & \text{if } |w| > k \end{cases} \tag{11}$$

This definition can be 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 absense of derivation, somewhat harder to define:

**Definition 7 ([9]).** *Let us say that $u \in \Sigma^*$ follows $\sigma \in (\Sigma \cup N)^*$ if there exists a sequence of conjuncts $A_0 \to \alpha_1 A_1 \beta_1$, $A_1 \to \alpha_2 A_2 \beta_2$, $\dots$ $A_{\ell-1} \to \alpha_\ell A_\ell \beta_\ell$, $A_\ell \to \eta\sigma\theta$, such that $A_0 = S$ and $u \in L_G(\theta\beta_\ell \dots \beta_1)$*

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 [6]. 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. Algorithms to construct supersets $\textsc{PFirst}_k(A) \supseteq \textsc{First}_k(A)$ and $\textsc{PFollow}_k(A) \supseteq \textsc{Follow}_k(A)$ have been developed for top-down parsing of Boolean grammars and can be found in the corresponding technical report [9]; let us reuse them for LR parsing.

The sets $\textsc{PFollow}_k(A)$ are used to define the *reduction function* $R : Q \times \Sigma^{\leqslant k} \to \text{dc}(P)$ which tells the reductions possible 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:

$$R(q, u) = \{A \to \alpha \mid A \to \alpha\cdot \in q,\ u \in \textsc{PFollow}_k(A)\} \tag{12}$$

for every $q \in Q$ and $u \in \Sigma^{\leqslant k}$. In the notation of Aho, Sethi and Ullman, $A \to \alpha \in R(q, u)$ means "$\textsc{Action}[q, u] = \text{Reduce } A \to \alpha$".

As in the context-free case, the states from $Q \setminus \{q_{error}\}$ can be enumerated with consecutive numbers $0, 1, \dots, |Q| - 1$, where 0 refers to the state $q_0$. The set of dotted conjuncts (*items*) corresponding to a state $q \in Q$ will be denoted $I(q)$ then.

# 5 The algorithm

The new LR-style algorithm for Boolean grammars is a generalization of the algorithm for conjunctive grammars [6], which in turn extends Tomita's algorithm [12] for context-free grammars.

All three algorithms share a common data structure: a *graph-structured stack*, introduced by Tomita [12] as a compact representation of the contents of the linear stack of Knuth's LR algorithm in all possible branches of nondeterministic computation. The graph-structured stack is an oriented graph with a designated *source node*. The nodes are labeled with the states of the LR automaton (such as the SLR($k$) automaton constructed in the previous section), with the source node labeled with the initial state. The arcs are labeled 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 from the top layer. The labels of these nodes should be pairwise distinct, and hence there can be at most $|Q|$ top layer nodes.

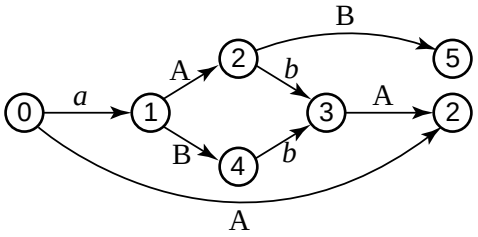


Figure 2: Sample contents of the graph-structured stack.

Consider the graph in Figure 2: the leftmost node labeled 0 is the source node; the two rightmost nodes labeled 5 and 2 form the top layer. There is another node labeled 2 – the direct 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*, which modify the nodes in the top layer without reading the input, and *shift phases*, each reading and consuming a single input symbol and forming an entirely new top layer 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 labeled with a state $q$, the algorithm looks up the transition table, $\delta(q, a)$. If $\delta(q, a) = q' \in Q$, then a new node labeled $q'$ is created and $q$ is connected to $q'$ with an arc labeled $a$; this action is called *Shift* $q'$. If $\delta(q, a) = -$, no new nodes are created, this condition is called "Local error"; if this is the case for all the nodes, the algorithm terminates, reporting a syntax error. The nodes created during a shift phase form the new top layer of the graph, while the previous top layer nodes are demoted into regular nodes.

In Figure 3(a) the top layer contains the nodes 1, 2, 3 and 4, and $\delta(1, a) = 5$, $\delta(2, a) = -$ and $\delta(3, a) = \delta(4, a) = 6$. Thus a new top layer formed of 5 and 6 is created, while 2 and its

predecessors that are now disconnected from the new top layer are removed from the stack.

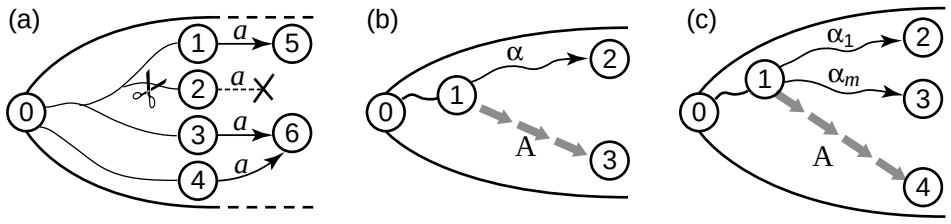


Figure 3: (a) Shifting; (b) Reductions for context-free and (c) conjunctive grammars.

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

In the **context-free** case [12], the only operation is *reduction*. Suppose there exists a top layer node $q$, a node $q'$ and a rule $A \to \alpha$, such that $A \to \alpha \in R(q)$ and $q'$ is connected to $q$ by a path $\alpha$. Then the algorithm can perform the operation "Reduce $A \to \alpha$" at $q'$, adding a new arc labeled with $A$, which goes from $q'$ to a top layer node labeled $\delta(q', A)$. If there is no node $\delta(q', A)$ in the top layer, it is created. This case is illustrated in Figure 3(b).

In the **conjunctive** case [6], *reduction* is still the only operation. However, now rules may consist of multiple conjuncts, and hence the conditions of performing a reduction are slightly more complicated. Let $A \to \alpha_1 \& \ldots \& \alpha_m$ be a rule, let $q$ be a node and let $q_{i_1}, \ldots, q_{i_m}$ be top layer nodes, such that, for all $j$, $A \to \alpha_j \in R(q_{i_j})$ and $q$ is connected to each $q_{i_j}$ by a path $\alpha_j$. 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 $\delta(q, A)$. See Figure 3(c).

The case of **Boolean** grammars requires more complicated handling. There are two operations: *reduction*, which is the same as in the previous cases, but with yet more complicated prerequisites, 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 node $q$ should be connected to the top layer by each of the paths $\alpha_1, \ldots, \alpha_m$ and by none of the paths $\beta_1, \ldots, \beta_n$. This nonexistence of paths is shown in Figure 4(left) by dotted lines ending with crosses. This allows the algorithm to do "Reduce $A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \beta_n$", adding an arc labeled $A$ from $q$ to $\delta(q, A)$ in the top layer.

Invalidation is the opposite of reduction. Suppose there exists a node $q$ and an arc labeled with $A$ 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 $A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \beta_n$ for $A$ either for some $i$ there is no path from $q$ to the top layer labeled $\alpha_i$, or for some $j$ there exists a path from $q$ to the top layer with the labels forming $\beta_j$. Then the earlier made 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.

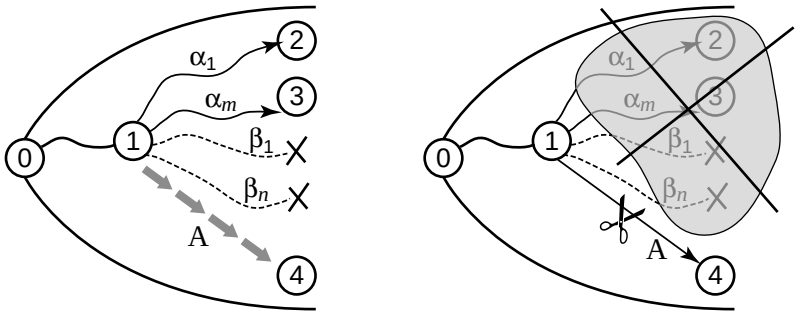


Figure 4: Reduction phase for Boolean grammars: *Reduce* and *Invalidate*

Let us note that in the case of context-free and conjunctive grammars, in the absense of negation, arcs can only be added, and the conditions for invalidation would never hold. On the other hand, if there is 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 $\alpha$ from $q$ to the top layer, but there is no path $\beta$; however, subsequent reductions may cause this path $\beta$ to appear, rendering the earlier made reduction invalid. This is something that does not have an analog in LR parsing for negation-free grammars.

The reduction phase as a whole can be partially formalized by the following nondeterministic algorithm:

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

In Section 7 below the algorithm is claimed to be correct regardless of the choice of reductions/invalidations at every step.

One possible implementation of the reduction phase is to do just one action at once; this is the most intuitively clear approach, and it is used in the example in Section 6. The implementation described in Section 8 does all valid reductions and invalidations at every step, which allows to prove a polynomial complexity upper bound.

Except for these major differences in the reduction phase, the three algorithms are identical in all other respects. Following is the general schedule of the algorithm:

> Input: a string $w = a_1 \dots a_n$.
> Let the stack contain a single node $x$ labeled $q_0$, let the top layer be $\{x\}$.
> Do the *Reduction phase* using lookahead $First_k(w)$
> for $i = 1$ to $|w|$
>  Do the *Shift phase* using $a_i$
>  If the top layer is empty, 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 labeled $S$ from the source node to $\delta(q_0, S)$ in the top layer
    Accept
else
    Reject

# 6 Example

.

Consider the following grammar for the language $\{a^n b^n c^n \mid n \geqslant 0\}$:

$$
\begin{array}{l}
S \rightarrow AP\&QC \\
A \rightarrow Aa \mid \varepsilon \\
C \rightarrow Cc \mid \varepsilon \\
P \rightarrow bPc \mid \varepsilon \\
Q \rightarrow aQb \mid \varepsilon
\end{array}
$$

It does not use negation at all, and hence is a conjunctive grammar [5]; the generalized LR parsing algorithm for this class of grammars [6] is just a simple subcase of the algorithm being investigated now. Let us artificially modify this grammar by restating the pair of context-free rules for the nonterminal $Q$ using de Morgan's law. The resulting grammar generates the same language, but makes a heavy use of negation, which would be interesting to trace in action:

$$
\begin{array}{l}
S \rightarrow AP\&QC \\
A \rightarrow Aa \mid \varepsilon \\
C \rightarrow Cc \mid \varepsilon \\
P \rightarrow bPc \mid \varepsilon \\
Q \rightarrow \neg R \\
R \rightarrow \neg aQb\&\neg\varepsilon
\end{array}
$$

In order to comply to the requirement of the positive first conjunct, let us add a new nonterminal $X$ that generates $\{a, b\}^*$ and then use conjunction with $X$ in the rules for $Q$ and $R$. This gives the following grammar, which still generates $\{a^n b^n c^n \mid n \geqslant 0\}$:

$$
\begin{array}{l}
S \rightarrow AP\&QC \\
A \rightarrow Aa \mid \varepsilon \\
C \rightarrow Cc \mid \varepsilon \\
P \rightarrow bPc \mid \varepsilon \\
Q \rightarrow X\&\neg R \\
R \rightarrow X\&\neg aQb\&\neg\varepsilon \\
X \rightarrow Xa \mid Xb \mid \varepsilon
\end{array}
$$

Table 1 contains the sets $\textsc{First}_1$ and $\textsc{Follow}_1$ constructed for this grammar.

| | $First_1$ | $Follow_1$ |
|---|---|---|
| $S$ | $\{\varepsilon, a, b\}$ | $\{\varepsilon\}$ |
| $A$ | $\{\varepsilon, a\}$ | $\{\varepsilon, a, b\}$ |
| $C$ | $\{\varepsilon, c\}$ | $\{\varepsilon, c\}$ |
| $P$ | $\{\varepsilon, b\}$ | $\{\varepsilon, c\}$ |
| $Q$ | $\{\varepsilon, a, b\}$ | $\{\varepsilon, b, c\}$ |
| $R$ | $\{a, b\}$ | $\{\varepsilon, b, c\}$ |
| $X$ | $\{\varepsilon, a, b\}$ | $\{\varepsilon, a, b, c\}$ |

Table 1: PFIRST$_1$ and PFOLLOW$_1$ tables.

| | $\delta$ | | | | | | | | | | $R$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $a$ | $b$ | $c$ | $S$ | $A$ | $C$ | $P$ | $Q$ | $R$ | $X$ | $\varepsilon$ | $a$ | $b$ | $c$ |
| 0 | 1 | | | 17 | 2 | | | 3 | 4 | 5 | $A \to \varepsilon$<br>$R \to \neg\varepsilon$<br>$X \to \varepsilon$ | $A \to \varepsilon$<br>$X \to \varepsilon$ | $A \to \varepsilon$<br>$R \to \neg\varepsilon$<br>$X \to \varepsilon$ | $R \to \neg\varepsilon$<br>$X \to \varepsilon$ |
| 1 | 1 | | | | | | | 6 | 4 | 5 | $R \to \neg\varepsilon$<br>$X \to \varepsilon$ | $X \to \varepsilon$ | $R \to \neg\varepsilon$<br>$X \to \varepsilon$ | $R \to \neg\varepsilon$<br>$X \to \varepsilon$ |
| 2 | 7 | 8 | | | | | 9 | | | | $P \to \varepsilon$ | | | $P \to \varepsilon$ |
| 3 | | | | | | 10 | | | | | $C \to \varepsilon$ | | | $C \to \varepsilon$ |
| 4 | | | | | | | | | | | $Q \to \neg R$ | | $Q \to \neg R$ | $Q \to \neg R$ |
| 5 | 11 | 12 | | | | | | | | | $Q \to X$<br>$R \to X$ | | $Q \to X$<br>$R \to X$ | $Q \to X$<br>$R \to X$ |
| 6 | | 13 | | | | | | | | | | | | |
| 7 | | | | | | | | | | | $A \to Aa$ | $A \to Aa$ | $A \to Aa$ | |
| 8 | | 8 | | | | | 14 | | | | $P \to \varepsilon$ | | | $P \to \varepsilon$ |
| 9 | | | | | | | | | | | $S \to AP$ | | | |
| 10 | | | 15 | | | | | | | | $S \to QC$ | | | |
| 11 | | | | | | | | | | | $X \to Xa$ | $X \to Xa$ | $X \to Xa$ | $X \to Xa$ |
| 12 | | | | | | | | | | | $X \to Xb$ | $X \to Xb$ | $X \to Xb$ | $X \to Xb$ |
| 13 | | | | | | | | | | | $R \to \neg aQb$ | | $R \to \neg aQb$ | $R \to \neg aQb$ |
| 14 | | | 16 | | | | | | | | | | | |
| 15 | | | | | | | | | | | $C \to Cc$ | | | $C \to Cc$ |
| 16 | | | | | | | | | | | $P \to bPc$ | | | $P \to bPc$ |
| 17 | | | | | | | | | | | | | | |

Table 2: The LR table.

The LR table for this grammar is given in Table 2. Let us trace the computation of the corresponding Boolean LR parser on the input $w = abc$.

**Reduction phase:** $\cdot abc$ This is the computation done before consuming any input symbols. Initially, the stack contains the source node only, as shown in Figure 5 (left).

The lookahead symbol is $a$, the reduction function for the only node evaluates to $R(0, a) = \{A \to \varepsilon,\ X \to \varepsilon\}$. Each of these conjuncts is a rule in itself, and hence the following two context-free reductions can be done:

- Reduce by $A \to \varepsilon$ from 0 to 2.
- Reduce by $X \to \varepsilon$ from 0 to 5.

The stack is now as in Figure 5 (right). No other reductions are possible, since $R(2, a) = R(5, a) = \varnothing$.

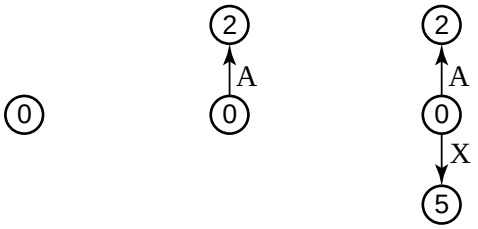


Figure 5: Example of LR parsing: $\cdot abc$.

**Shift phase:** $\boldsymbol{a}bc$ There are three nodes in the top layer: 0, 2 and 5 – see Figure 5 (right). For each of them the transition by $a$ is defined.

- Shift 0 to $1 = \delta(1, a)$.
- Shift 2 to 7.
- Shift 5 to 11.

The resulting contents of the tree-structured stack is shown in Figure 6 (leftmost).

**Reduction phase:** $a \cdot bc$ The nodes in the top layer are 7, 1 and 11, the lookahead symbol is $b$. The reduction function provides the following conjuncts: $R(7, b) = \{A \to Aa\}$, $R(1, b) = \{R \to \neg\varepsilon, X \to \varepsilon\}$, $R(11, b) = \{X \to Xa\}$. Among these four conjuncts, $R \to \neg\varepsilon$ is a part of the rule $R \to X\&\neg aQb\&\neg\varepsilon$, by which we cannot reduce in the absence of the other two conjuncts, while the other three conjuncts form context-free rules, which can be handled as in the context-free case:

- Reduce by $A \to Aa$ from 0 to 2.
- Reduce by $X \to Xa$ from 0 to 5.

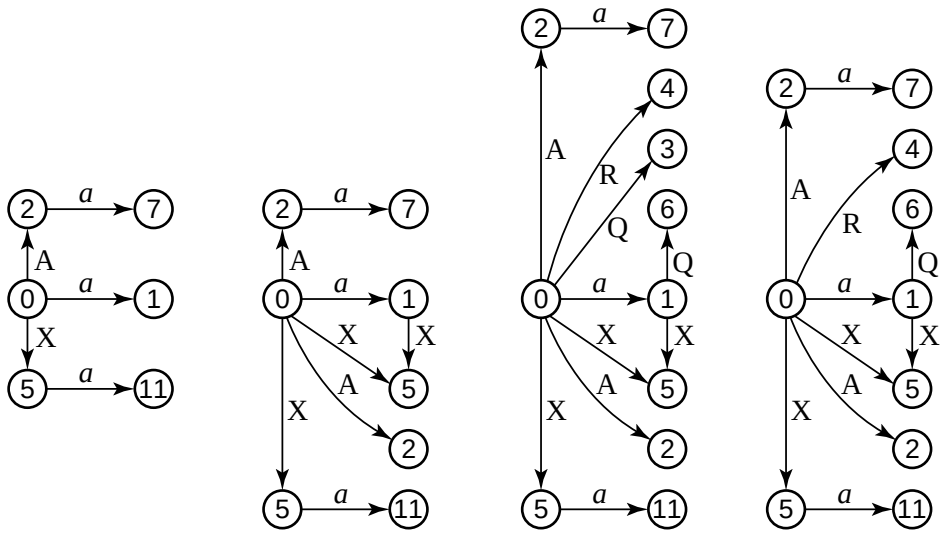


Figure 6: Example of LR parsing: $a \cdot bc$.

- Reduce by $X \to \varepsilon$ from 1 to 5. Since the state 5 already exists in the top layer, no new nodes are created, and the new arc arrives at the node added at the previous step.

The contents of the stack at this point is shown in Figure 6 (the second graph).

For the newly added nodes, 2 and 5, the relevant entries of the reduction table are: $R(2, b) = \varnothing$ and $R(5, b) = \{Q \to X, R \to X\}$. Now having the conjunct $Q \to X$, it is possible to "assemble" the rule $Q \to X\&\neg R$, while the conjunct $R \to X$ allows to consider reduction by the rule $R \to X\&\neg aQb\&\neg\varepsilon$:

- Reduce by $Q \to X\&\neg R$ from 0 to 3. This reduction is possible, because (1) there is a path from 0 to the top layer with labels on the arcs forming $X$ (which satisfies the conjunct $Q \to X$), *and* (2) there is *no* path from 0 to the top layer with labels on the arcs forming $R$ (which satisfies the conjunct ).
- Reduce by $Q \to X\&\neg R$ from 1 to 6. This case is similar to the previous one.
- Reduce by $R \to X\&\neg aQb\&\neg\varepsilon$ from 0 to 4. This is possible, because: (1) 0 is connected to the top layer by a path "$X$" (which satisfies the conjunct $R \to X$), *and* (2) 0 is *not* connected to the top layer by a path "$aQb$", *and* (3) 0 is *not* connected to the top layer by a path "$\varepsilon$".

  Note that we cannot similarly reduce by $R \to X\&\neg aQb\&\neg\varepsilon$ from 1 (to 4). That is because the node 1 *is* connected to the top layer by a path "$\varepsilon$" and $R \to \neg\varepsilon \in R(1, b)$. Thus the condition (3) does not hold for the node 1, and the reduction is impossible.

Three more nodes, 3, 6 and 4, have been added to the top layer; the stack is now as displayed in Figure 6 (the third graph). The following conjuncts are given by the reduction function: $R(3, b) = \varnothing$, $R(6, b) = \varnothing$ and $R(4, b) = \{Q \rightarrow \neg R\}$. It is the last of these conjuncts that calls for a change, this time a removal of an arc.

The earlier made reduction by $Q \rightarrow X\&\neg R$ from 0 to 3 relied upon the non-existence of a path from 0 to the top layer labeled with $R$. Once the arc from 0 to 4 labeled with $R$ has been added, this condition no longer holds, and hence the *invalidation* operation has to be executed: Hence the earlier made reduction has to be reversed:

- Invalidate the reduction by $Q \rightarrow X\&\neg R$ from 0 to 3.

At this point the graph (the rightmost in Figure 6) is stable: no new arcs can be added, and no arcs can be removed either.

**Shift phase:** $a\boldsymbol{b}c$ There are seven nodes in the top layer: 7, 4, 6, 1, 5, 2, 11. The transition by $b$ is defined for the states 6, 5 and 2 only.

- Local error in 7: both 7 and its predecessor 2 are removed from the stack.
- Local error in 4: it is removed.
- Shift 6 to 13.
- Local error in 1, but it is not removed, since it has successors.
- Shift 5 to 12.
- Shift 2 to 8.
- Local error in 11: both 11 and its predecessor 5 are removed.

The contents of the graph-structured stack after shifting the symbol $b$ is shown in Figure 7(left).

**Reduction phase:** $ab \cdot c$ The top layer contains the nodes 13, 12 and 8. $R(13, c) = \{R \rightarrow \neg aQb\}$, $R(12, c) = \{X \rightarrow Xb\}$, $R(8, c) = \{P \rightarrow \varepsilon\}$. $P \rightarrow \varepsilon$ and $X \rightarrow Xb$ form single-conjunct rules, while the negative conjunct $R \rightarrow \neg aQb$ is a part of a three-conjunct rule. The following actions can be done:

- Reduce by $X \rightarrow Xb$ from 0 to 5.
- Reduce by $X \rightarrow Xb$ from 1 to the existing state 5.
- Reduce by $P \rightarrow \varepsilon$ from 8 to 14.

Two new states were added, 5 and 14; the new conjuncts are $R(5, c) = \{Q \rightarrow X, R \rightarrow X\}$ and $R(14, c) = \varnothing$. The conjuncts $Q \rightarrow X$ and $R \rightarrow X$ allow to consider the rules $Q \rightarrow X\&\ R$ and $R \rightarrow X\&\neg aQb\&\neg\varepsilon$:

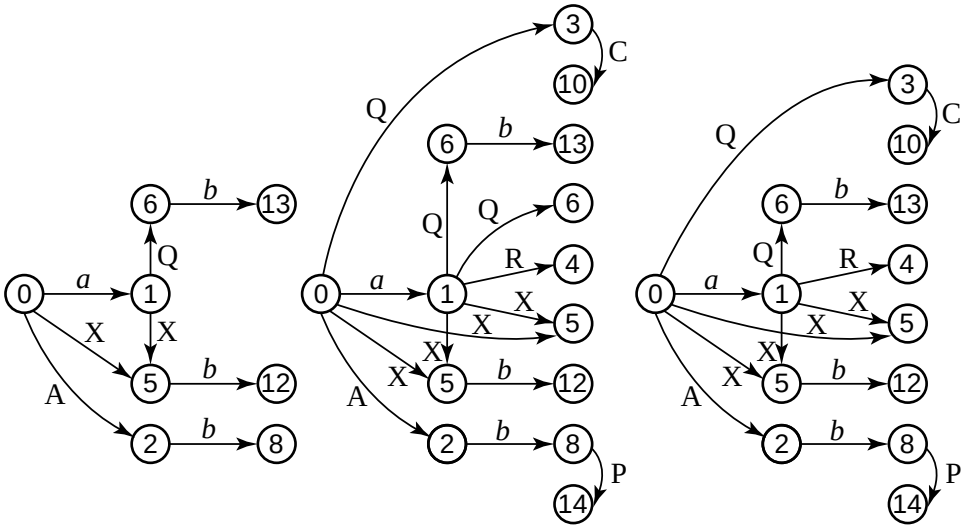


Figure 7: Example of LR parsing: $ab \cdot c$.

- Reduce by $Q \to X\& \ R$ from 0 to 3.
- Reduce by $Q \to X\& \ R$ from 1 to 6: there is an arc $X$ from 1 to 5 in the top layer (it has been added a couple of steps before), but there is no arc $R$ from 1 to the top layer.
- Reduce by $R \to X\&\neg aQb\&\neg\varepsilon$ from 1 to 4, justified by the arc $X$ from 1 to 5 in the top layer and by the absense of paths $aQb$ and $\varepsilon$ from 1 to the top layer.

Now the condition that "there is no arc $R$ from 1 to the top layer" is no longer true, and hence the previous reduction has to be invalidated. Also, the other arc $Q$ from 0 to 3 in the top layer yields a conjunct $\{C \to \varepsilon\} = R(3, c)$, which makes one more reduction possible.

- Invalidate reduction by $Q \to X\& \ R$ from 1 to 6. The state 6 is removed.
- Reduce by $C \to \varepsilon$ from 3 to 10.

The graph, presented in Figure 7(right), is now stable.

**Shift phase:** $abc$ The top layer contains eight nodes: 3, 10, 13, 4, 5, 12, 8, 14. The transition by $b$ is defined for 10 and 14.

- Local error in 3: it is not removed because it has a successor.
- Shift 10 to 15.
- Local error in 13: it is removed along with its predecessor 6.
- Local error in 4: it is removed.

- Local error in 5: it is removed.
- Local error in 12: it is removed, along with its direct predecessor 5. This time the node 1 is removed as well, because all of its successors are now gone.
- Local error in 8: it is not removed because of a successor.
- Shift 14 to 16.

This yields a stack with two nodes in the top layer, 15 and 16. It is shown in Figure 8(left).

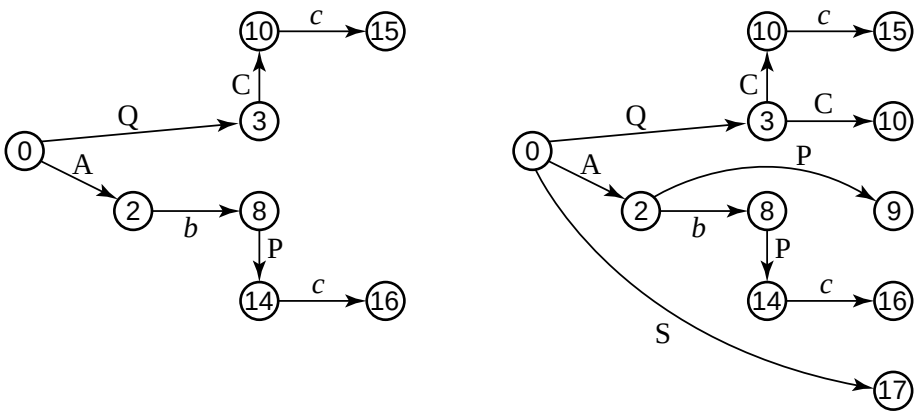


Figure 8: Example of LR parsing: $abc\cdot$.

**Reduction phase:** $abc\cdot$ At this point the string has been entirely consumed and the lookahead string is $\varepsilon$. The sets of conjuncts given by the reduction function are $R(15, \varepsilon) = \{C \to Cc\}$ and $R(16, \varepsilon) = \{P \to bRc\}$. Both conjuncts are alone in their respective rules, and hence these two steps are done as in the context-free case:

- Reduce by $C \to Cc$ from 3 to 10.
- Reduce by $P \to bPc$ from 2 to 9.

The current contents of the stack is shown in Figure 8(middle).

The two new nodes bring forth new conjuncts: $R(10, \varepsilon) = \{S \to QC\}$, $R(9, \varepsilon) = \{S \to AP\}$. These two conjuncts are the halves of the rule $S \to AP\&QC$, which allows to carry out the following reduction:

- Reduce by $S \to AP\&QC$ from 0 to 17.

The contents of the stack, given in Figure 8(right), has now stabilized.

**The acceptance condition** The whole input has been consumed and the stack is stable. Since it contains an arc from the source node to the top layer labeled with $S$, the string is accepted.

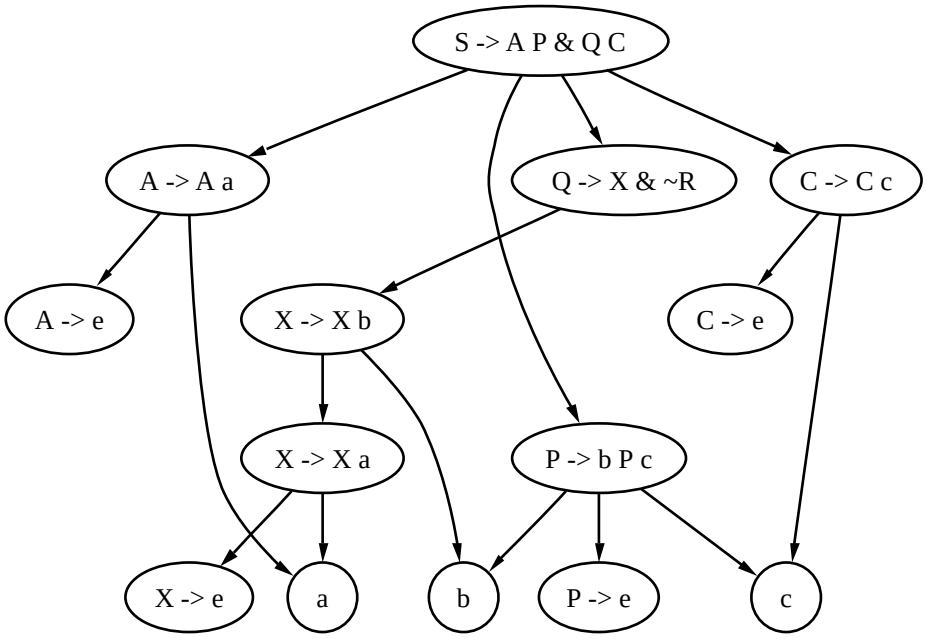


Figure 9: The parse tree of $abc$ constructed by an implementation.

# 7 On proving the correctness

Though no formal proof of the algorithm's correctness has been written so far, some observations on a possible proof are provided in this section.

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 8.** *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar without negatively fed cycles. let $w = a_1 \dots a_{|w|}$ be the input string, let $(\Sigma, N, Q, q_0, \delta, R)$ be the SLR($k$) automaton.*

*The graph-structured stack is an acyclic graph with the set of vertices $V = Q \times \{0, 1, \dots, |w|\}$ and with the arcs labeled with symbols from $\Sigma \cup N$, such that the following conditions hold: for every arc from $(q', p')$ to $(q'', p'')$ labeled 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. The nonempty layer with the maximal number is called the top layer.*

Consider the reduction phase in a layer $p$ ($0 \leqslant p \leqslant |w|$): the symbols $a_1 \dots a_p$ have

already been read, while $First_k(a_{p+1} \ldots a_{|w|})$ is the lookahead string. Any arcs going to layers less than $p$ (i.e., other than to the top layer) are fixed and cannot be changed in course of the reduction phase. The terminal arcs leading to the top layer are also fixed. These will be called *permanent arcs* in this proof.

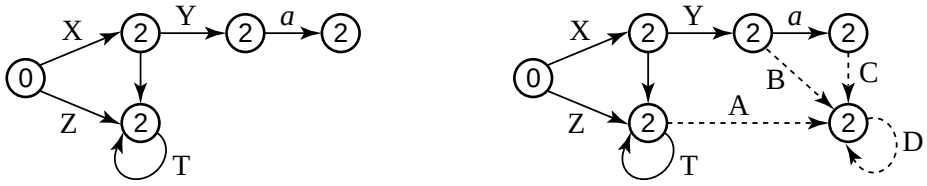


Figure 10: A graph-structured stack before a reduction phase; the possible arcs.

The reduction phase manipulates the nonterminal arcs going to the top layer. Initially, there are none of them. A reduction adds an arc. An invalidation removes an arc. This process can be viewed as flipping the bits in the bit vector of possible arcs, starting from a zero vector.

A triple $(q, p_0, A)$, where $q \in Q$, $0 \leqslant p_0 \leqslant p$ and $A \in N$, is called a *possible arc* (implying an arc from $(q, p_0)$ to $(\delta(q, A), p)$ labeled $A$), if 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. It is easy to see that there are at most $p \cdot |Q|^2$ possible arcs.

Every possible arc $(q, p_0, A)$ has a *condition of existence* – a certain configuration of permanent and possible arcs in the graph that causes a reduction by some rule for $A$ at the vertex $(q, p_0)$, or prevents an invalidation of $A$ at $(q, p_0)$. A possible arc $(q', p', A)$ is said to *influence* a possible arc $(q'', p'', A)$, if the condition of existence of $(q'', p'', A)$ essentially depends upon $(q', p', A)$.

These dependencies can be written down in the form of a system of Boolean equations, and then the behaviour of the algorithm at each reduction phase corresponds to a search for a solution of this Boolean system – exactly as in Lemma 1.

**Proposition 1 (Termination and confluence).** *The reduction phase in every layer $p$ terminates, and the contents of the graph-structured stack after every phase does not depend upon the order of reductions and invalidations.*

**Definition 9.** *A possible arc $(q', p', A)$ is called tentatively correct if and only if $\delta(q', A) \neq -$, $a_{p'+1} \ldots a_p \in L_G(A)$, $First_k(a_{p+1} \ldots a_n) \in \textsc{Pfollow}_k(A)$.*

**Proposition 2 (A model computation).** *For each layer $p$, there exists a computation of the reduction plase in the layer $p$ that employs only reductions and no invalidations, and constructs a stable graph, which contains the maximal subset of tentatively correct arcs, such that the graph is connected.*

A formal proof of the algorithm's correctness can apparently be constructed around these two propositions.

# 8 Implementation

The overall composition of the algorithm has been given in Section 5, and it can be directly used in an implementation. The only question is how to implement the reduction phase.

The results of Section 7 clearly imply that the reduction phases can be implemented in many different ways, and the algorithm would still be correct. One possible implementation is suggested in this section. This implementation is based upon doing all possible actions at every step of the reduction phase; the rest are straightforward graph search techniques.

**Definition 10.** *Consider a fixed state 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 $\ell$ arcs long.*

This set can be computed inductively on $\ell$ in the following way:

- $predecessors_0(v) = \{v\}$.
- $predecessors_{\ell+1}(v)$ consists 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 → α ∈ R(q, u)
            x[A → α] = x[A → α] ∪ predecessors_|α|(v)
    // Reductions.
    let valid be a set of arcs, initially empty
    for each rule A → α_1 & ... & α_m & ¬β_1 & ... & ¬β_n ∈ P
        for each node v ∈ ⋂_{i=1}^{m} x[A → α_i] \ ⋃_{i=1}^{n} x[A → β_i]
            if v is not connected to the top layer by an arc labeled A
                add an arc from v to the top layer labeled A
            valid = valid ∪ {the arc from v to the top layer labeled A}
    // Invalidations.
    for each node v = (q, p_top) in the top layer
        for each incoming arc (v', v) labeled A
            if this arc is not in the set valid
                remove the arc from the graph
}
```

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

Let us give an upper bound for the complexity of this implementation of the reduction phase, and of the algorithm as a whole. The number of iterations in each reduction phase should be $O(n)$: the idea is that 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 should be enough. Note that if reductions and invalidations were applied randomly one by one, then it would be easy to construct an example on which exponentially many operations are required.

The conjunct gathering stage computes $predecessors_\ell$ a constant number of times, and $\ell$ is also bounded by a constant. Computing the set of predecessors involves considering $O(n)$ nodes, each of which has $O(n)$ predecessors. Hence, each conjunct gathering stage takes $O(n^2)$ steps. The reduction and invalidation stages take $O(n)$ time.

This gives a cubic upper bound for the complexity of the reduction phase, while the complexity of the whole algorithm is $O(n^4)$. The worst-case execution time can be improved to $O(n^3)$ by using a clumsy method for doing the conjunct gathering developed for the conjunctive LR [6].

# 9 Conclusion

A practically usable algorithm for the recently introduced Boolean grammars has been obtained. Though the proof of its correctness is not yet complete, the algorithm itself has been implemented in an ongoing parser generator project [7] and tested on numerous grammars, including a grammar for the set of well-formed programs in a simple programming language [10], which is being parsed in quadratic time.

# References


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

[2] J. Aycock, R. N. Horspool, J. Janousek, B. Melichar, "Even faster generalized LR parsing", *Acta Informatica*, 37:9 (2001), 633–651.

[3] S. Ginsburg, H. G. Rice, "Two families of languages related to ALGOL", *Journal of the ACM*, 9 (1962), 350–371.

[4] D. E. Knuth, “On the translation of languages from left to right”, *Information and Control*, 8 (1965), 607–639.

[5] A. Okhotin, “Conjunctive grammars”, *Journal of Automata, Languages and Combinatorics*, 6:4 (2001), 519–535.

[6] A. Okhotin, “LR parsing for conjunctive grammars”, *Grammars* 5:2 (2002), 81–124.

[7] A. Okhotin, “Whale Calf, a parser generator for conjunctive grammars”, *Implementation and Application of Automata* (Proceedings of CIAA 2002, Tours, France, July 3–5, 2002), LNCS 2608, 213–220.

[8] A. Okhotin, “Boolean grammars”, *Information and Computation*, to appear; preliminary version in: *Developments in Language Theory* (Proceedings of DLT 2003, Szeged, Hungary, July 7–11, 2003), LNCS 2710, 398–410.

[9] A. Okhotin, “An extension of recursive descent parsing for Boolean grammars”, Technical Report 2004–475, School of Computing, Queen’s University, Kingston, Ontario, Canada.

[10] A. Okhotin, “A Boolean grammar for a simple programming language”, Technical Report 2004–478, School of Computing, Queen’s University, Kingston, Ontario, Canada.

[11] M. Tomita, *Efficient Parsing for Natural Language*, Kluwer, 1986.

[12] M. Tomita, “An efficient augmented context-free parsing algorithm”, *Computational Linguistics*, 13:1 (1987), 31–46.

[13] M. Tomita (Ed.), *Generalized LR Parsing*, Kluwer, 1991.