Recursive descent parsing for Boolean grammars

Alexander Okhotin*

March 26, 2007

Abstract

The recursive descent parsing method for the context-free grammars is extended for their generalization, Boolean grammars, which include explicit set-theoretic operations in the formalism of rules and which are formally defined by language equations. The algorithm is applicable to a subset of Boolean grammars. The complexity of a direct implementation varies between linear and exponential, while memoization keeps it down to linear.

1 Introduction

Boolean grammars [16], a new formalism for specifying formal languages introduced by the author in 2003, are based on two fundamental concepts. One of these concepts is specification of languages by inductive definitions, as in context-free grammars. The other concept is propositional logic, in which conjunction, disjunction and negation of any conditions may be freely combined. The resulting extension of context-free grammars uses rules of the form

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n,$$

where α_i and β_i are strings comprised of terminal and nonterminal symbols, and the intuitive meaning of such a rule is that every string that satisfies all conditions represented by $\alpha_1, \ldots, \alpha_m$ and does not satisfy any of the conditions $\beta_1, \ldots, \beta_n$ is therefore generated by A.

The first formal definition of Boolean grammars was based upon language equations [16], and recently some alternative theoretical foundations for Boolean grammars were proposed by Wrona [22], by Kountouriotis et al. [9] and by the author [17]. The common result of these approaches is that every Boolean grammar can be converted to a generalized variant of Chomsky normal form and then parsed in time $O(n^3)$ by an extension of the Cocke–Kasami–Younger algorithm. A number of theoretical problems remain open: in particular, no methods for proving languages to be nonrepresentable by Boolean grammars are yet known.

The prospects of a practical use of Boolean grammars look quite good. The freedom of specifying any Boolean conditions in grammars provides them with higher expressive power than that of the context-free grammars, and makes them a more convenient tool for describing

*Academy of Finland *and* Department of Mathematics, University of Turku, Turku FIN–20014, Finland. E-mail: alexander.okhotin@utu.fi. Supported by the Academy of Finland under grant 118540.

languages. Since the upper bound for parsing complexity is the same, this extended power is given at little cost. In addition, more practical parsing algorithms for Boolean grammars are now being researched. The Generalized LR parsing method has been successfully applied to Boolean grammars by the author [18]: the resulting algorithm works in time $O(n^4)$, but the time remains linear on LR(1) context-free grammars as well as on some Boolean grammars that use conjunction and negation moderately. The use of Boolean grammars in software engineering has been attempted by Megacz [12], who developed a programmer-oriented parser generator using another variant of Generalized LR.

This paper applies another parsing technique, the *recursive descent*, to Boolean grammars. A recursive descent parser is a program containing a procedure for every symbol used in the grammar, where the code in each procedure mechanically transcribes the grammar rules for this symbol and is used to match a substring of the input string according to one of these rules. This is probably the most well-known and the most intuitively clear parsing technique, which has been in use since the early 1960s. This paper focuses at the deterministic case of the recursive descent method known as *predictive parsing* [1, pp. 44–48], though the hereby established results might be as well applicable to recursive descent with backtracking.

Quite a few extensions to the basic context-free recursive descent have been studied before, such as the top-down parsing method of Birman and Ullman [2] and its generalization proposed by Ford [3]. In fact, every advanced recursive descent parser generator, such as LLgen [6] and ANTLR [19], introduces its own extension to this method. Such extensions give the user a control over the flow of recursive descent parsing by the means of some directives attached to a grammar; in particular, Boolean operations on the results of computations [3] can be used to produce an effect that resembles Boolean grammars but is not equivalent to them. In this way the user can alter the computation of the parser to recognize a desired language, possibly a non-context-free one.

The parsing method proposed in this paper is, on one hand, similar to the aforementioned methods of LLgen [6] and ANTLR [19], as well as to the method proposed by Ford [3], since it can be regarded as recursive descent with additional control structures set up according to a Boolean grammar. On the other hand, there is a major difference: the proposed generalization of recursive descent is based upon a mathematically well-defined family of formal grammars, which has semantics independent of the computation of any parser [9, 16]. As such, the present work can be regarded as a theoretical approach to the same problem area, which has so far been addressed using engineering methods only.

A definition of Boolean grammars is given in Section 2. Fortunately, its full complexity considered in the literature [9, 16] is not needed: as shown in Section 3, a known necessary condition for recursive descent parsing (that of the absence of left recursion) implies the simplest of the known well-formedness conditions for Boolean grammars [16]. This much simplifies the treatment of the proposed method.

A recursive descent parser for a Boolean grammar, defined in Sections 4 and 5, is generally similar to its context-free prototype. In order to handle the extended generative power of Boolean grammars, it has to use some techniques not found in the standard recursive descent. In particular, the conjunction is implemented by scanning a part of the input multiple times in different procedures. The mechanism of exception handling found in imperative programming languages since Ada [7] is used to implement the negation. The method remains simple

enough to be implemented manually, and the generation of basic parsers can be automated as straightforwardly as in the context-free case.

Proving that the computation of such a parser indeed corresponds to the definition of a Boolean grammar is the subject of Section 6. Though at the first glance the context-free and the Boolean recursive descent look similar, their mathematical justifications don't, and the proof presents some challenges not found in the context-free case. The algorithm's complexity is analyzed in Section 7: for a direct implementation it is in the worst case exponential, but it is reduced to linear by a straightforward application of the memoization technique, that is, by storing the result of every call to a procedure $A()$ on any suffix, and by reusing this result instead of recomputing it if $A()$ is ever called on the same suffix again. The use of the algorithm to construct a parse tree of the input string is explained in Section 8.

In the conclusion, two directions of further research are suggested. Following the example of the context-free LL theory [8, 10, 11, 20, 21], one can proceed with the theoretical study of the language family to which the Boolean recursive descent is applicable. Another task is to study the applicability of the algorithm for practical parsing tasks and to implement it in software.

This paper supercedes the author's earlier generalization of recursive descent [15] for a subclass of Boolean grammars known as conjunctive grammars [14].

2 Boolean grammars and their semantics

Definition 1 ([16]). *A Boolean grammar 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*

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n, \tag{1}$$

where $m + n \geqslant 1$, $\alpha_i, \beta_i \in (\Sigma \cup N)^*$; $S \in N$ 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, and α_i and β_j are their* bodies. *Let $conjuncts(P)$ be the set of all conjuncts. $A \to \pm\alpha_i$ and $A \to \pm\beta_j$ are called* unsigned conjuncts *and are used to refer to a positive or a negative conjunct with the specified body.*

In this paper it will be further assumed that $m \geqslant 1$ and $n \geqslant 0$ in every rule (1). There is no loss of generality in this assumption, because it is always possible to add a nonterminal that generates Σ^*, and use this nonterminal as a formal first positive conjunct in every rule that lacks one.

A Boolean grammar is called a *conjunctive grammar* [14] if negation is never used, that is, $n = 0$ for every rule (1). It degrades to a context-free grammar if neither negation nor conjunction are allowed, that is, $m = 1$ and $n = 0$ for each rule. The semantics of the special case of conjunctive and context-free grammars is easy to define:

Definition 2 ([14]). *Let $G = (\Sigma, N, P, S)$ be a conjunctive grammar, that is, a Boolean grammar with all rules of the form*

$$A \to \alpha_1 \& \ldots \& \alpha_m. \tag{2}$$

Consider terms formed of symbols from Σ and N connected by concatenation and conjunction, and define a rewriting of such terms as follows:

1. *Every subterm A can be rewritten with $(\alpha_1 \& \ldots \& \alpha_m)$, for any rule (2);*
2. *Every subterm $(w \& \ldots \& w)$, for any $w \in \Sigma^*$, can be rewritten with w.*

Then, for any term $\mathcal{A}$, $L_G(\mathcal{A}) = \{w \mid w \in \Sigma^*, \alpha \text{ derives } w\}$, and $L(G) = L_G(S)$.

In order to define the semantics of Boolean grammars in the general case, let us represent them as systems of equations with formal languages as unknowns:

Definition 3. *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 Σ in variables N, in which the equation for each variable $A \in N$ is*

$$A = \bigcup_{A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P} \left[\bigcap_{i=1}^{m} \alpha_i \cap \bigcap_{j=1}^{n} \overline{\beta_j} \right] \tag{3}$$

Each instance of a symbol $a \in \Sigma$ in such a system defines a constant language $\{a\}$, while each empty string denotes a constant language $\{\varepsilon\}$. A solution of such a system is a vector of languages $(\ldots, L_C, \ldots)_{C \in N}$, such that the substitution of L_C for C, for all $C \in N$, turns each equation (3) into an equality.

If negation is never used in G, then the associated system (3) always has a least solution with respect to componentwise inclusion [4, 17], and the components of this solution are exactly the languages generated by nonterminals according to Definition 2. In contrast to the conjunctive and context-free cases, a system (3) in the general case can have no solutions or multiple pairwise incomparable solutions. Moreover, even if one considers only systems with a unique solution, then every recursive language can be represented [16], and this representation is much different from the intuitive meaning of a Boolean grammar. In order to give a satisfactory formal definition of Boolean grammars, some additional condition must be assumed [16], such as the following:

Definition 4. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar, let (3) be the associated system of language equations. Suppose that for every finite language $M \subset \Sigma^*$ (such that for every $w \in M$ all substrings of w are also in M) there exists a unique vector of languages $(\ldots, L_C, \ldots)_{C \in N}$ ($L_C \subseteq M$), such that a substitution of L_C for C, for each $C \in N$, turns every equation (3) into an equality modulo intersection with M (in the following such a vector will be referred as a solution modulo M).*

Then G complies to the semantics of a strongly unique solution, and, for every $A \in N$, the language $L_G(A)$ can be defined as L_A from the unique solution of this system. This notation is extended to any expressions formed of terminal and nonterminal symbols, concatenation and Boolean operations as follows: $L_G(\varepsilon) = \{\varepsilon\}$, $L_G(a) = \{a\}$, $L_G(\psi \mid \xi) = L_G(\psi) \cup L_G(\xi)$, $L_G(\psi \& \xi) = L_G(\psi) \cap L_G(\xi)$, $L_G(\neg\psi) = \overline{L_G(\psi)}$, $L_G(\psi \cdot \xi) = L_G(\psi) \cdot L_G(\xi)$. The language generated by the grammar is $L(G) = L_G(S)$.

Example 1. *Consider a Boolean grammar* $G = (\{a, b, c\}, \{S, A, B, C, D\}, P, S)$*, where the rules in* P*, the associated system of language equations and its unique solution are as follows:*

$S \to AD\&\neg BC$	$S = AD \cap \overline{BC}$	$S = \{a^m b^n c^n \mid m, n \geqslant 0, m \neq n\}$
$A \to aA \mid \varepsilon$	$A = aA \ \cup \{\varepsilon\}$	$A = a^*$
$B \to aBb \mid \varepsilon$	$B = aBb \cup \{\varepsilon\}$	$B = \{a^m b^m \mid m \geqslant 0\}$
$C \to cC \mid \varepsilon$	$C = cC \cup \{\varepsilon\}$	$C = c^*$
$D \to bDc \mid \varepsilon$	$D = bDc \cup \{\varepsilon\}$	$D = \{b^n c^n \mid n \geqslant 0\}$

Compliance to Definition 4 is easy to verify, and hence $L(G) = \{a^m b^n c^n \mid m, n \geqslant 0, m \neq n\}$.

Note that this language is not context-free, which can be formally proved by pumping the string $a^{p+p!}b^p c^p \in L(G)$, where p is the constant given by the pumping lemma. Also note that if the negation in the rule for S is omitted, the resulting conjunctive grammar will regenerate the most common example of a non-context-free language, $\{a^n b^n c^n \mid n \geqslant 0\}$ [14].

Let us give a few examples of grammars deemed invalid according to Definition 4:

$$G_1 = (\{a\}, \{S\}, \{S \to \neg S\}, S)$$
$$G_2 = (\{a\}, \{S\}, \{S \to S\}, S)$$
$$G_3 = (\{a\}, \{S, A\}, \{S \to \neg S\&aA, A \to A\}, S)$$

For G_1, the associated language equation $S = \overline{S}$ has no solutions. The equation $S = S$ associated to G_2 has multiple solutions, in fact every language is a solution. Though the system of equations $\{S = \overline{S} \cap aA, A = A\}$ associated to G_3 has a unique solution $S = A = \varnothing$ (consider that if $w \in A$, then $aw \in S$ if and only if $aw \notin S$), the grammar is invalid, because the system has two solutions modulo every language $\{\varepsilon, a, \ldots, a^n\}$, namely, $(S = \varnothing, A = \varnothing)$ and $(S = \varnothing, A = \{a^n\})$.

Two other definitions of the semantics of Boolean grammars have been proposed [16, 9], which define slightly different sets of well-formed grammars. In particular, they cover all context-free grammars, unlike the semantics of strongly unique solution, which deems the above grammar G_2 invalid. However, these details of formal definition are irrelevant for the present paper. As we shall now see, a well-known necessary condition for recursive descent parsing, the absence of left recursion, implies the condition in Definition 4, and hence there is no need for any more general definition.

3 Strongly non-left-recursive grammars

Context-free recursive descent parsing requires the grammar to be free of *left recursion*, which means that no nonterminal A can derive $A\delta$ for any $\delta \in (\Sigma \cup N)^*$. The reason for that is that a parser can enter an infinite loop otherwise. Naturally, a generalization of recursive descent for a larger class of grammars has to impose a similar restriction. The notion of a left-recursive context-free grammar will now be generalized to the case of Boolean grammars.

Before the definition can be given, some technical notions need to be introduced. The first of these notions is an adaptation of context-free derivation to specify dependence of nonterminals in a Boolean grammar.

Definition 5. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar. The relation of context-free reachability, $\rightsquigarrow$, is a binary relation on the set of strings with a marked substring $\{\alpha\langle\beta\rangle\gamma \mid \alpha, \beta, \gamma \in (\Sigma \cup N)^*\}$ defined as the reflexive and transitive closure of the following set of derivation rules:*

$$\alpha\langle\beta A\gamma\rangle\delta \rightsquigarrow \alpha\beta\eta\langle\sigma\rangle\theta\gamma\delta,$$

for all $\alpha, \beta, \gamma, \delta \in (\Sigma \cup N)^*$, $A \in N$, $A \to \pm\eta\sigma\theta \in conjuncts(P)$.

Another technical notion is the following positive transform of a Boolean grammar:

Definition 6. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar. The grammar G_+ is a conjunctive grammar defined as $G_+ = (\Sigma, N, P_+, S)$, where*

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

For instance, for the grammar G from Example 1, the corresponding G_+ is a context-free grammar generating $L(G_+) = \{a^m b^n c^n \mid m, n \geqslant 0\}$.

Using the above terminology, the notion of left recursion can be extended for Boolean grammars as follows:

Definition 7. *A Boolean grammar $G = (\Sigma, N, P, S)$ is said to be strongly non-left-recursive if and only if for all $A \in N$ and $\theta, \eta \in (\Sigma \cup N)^*$, such that $\varepsilon\langle A\rangle\varepsilon \rightsquigarrow \theta\langle A\rangle\eta$, it holds that $\varepsilon \notin L_{G_+}(\theta)$.*

In particular, the non-left-recursiveness of the context-free grammar

$$G' = \big(\Sigma, N, \{A \to \alpha \mid A \to \pm\alpha \in conjuncts(P)\}, S\big)$$

is a sufficient condition for strong non-left-recursiveness of G, but Definition 7 is less restrictive. Using Lemma 6, it can now be established that Boolean grammars of this type are always well-defined:

Theorem 1. *Every strongly non-left-recursive Boolean grammar complies to the semantics of a strongly unique solution.*

The proof is included in the appendix.

An important question is whether left recursion in a Boolean grammar can be eliminated. For the context-free grammars the answer is affirmative, and in addition to the well-known simple procedure for eliminating left recursion [1, pp. 176–177] there exists a stronger normal form theorem due to Greibach [5]. However, the known transformations of context-free grammars are not applicable to Boolean grammars, and the question of whether for every Boolean grammar there exists an equivalent strongly non-left-recursive Boolean grammar remains open.

4 The LL(k) table and its construction

The computation of a context-free recursive descent parser is guided by a parsing table, which, for every $A \in N$ and $u \in \Sigma^*$ (where $|u| \leqslant k$, for a fixed $k > 0$), determines the rule to apply when A is to produce a string starting with u. This rule has to be "predicted" before seeing the entire substring derived from A, hence the name of *predictive parsing* [1]. The Boolean recursive descent uses tables of the same kind, and a method of constructing such a table for a given grammar will be described in this section.

Let us start from defining the general form of a parsing table. Let $k \geqslant 1$. For a 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 can be extended to languages as $First_k(L) = \{First_k(w) \mid w \in L\}$. Define $\Sigma^{\leqslant k} = \{w \mid w \in \Sigma^*, |w| \leqslant k\}$.

Definition 8. *A string $u \in \Sigma^*$ is said to follow $\sigma \in (\Sigma \cup N)^*$ if $\varepsilon\langle S\rangle\varepsilon \rightsquigarrow \theta\langle\sigma\rangle\eta$ for some $\theta, \eta \in (\Sigma \cup N)^*$, such that $u \in L_G(\eta)$.*

Definition 9 (Nondeterministic LL(k) table)**.** *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar, let $k > 0$. A nondeterministic LL(k) table for G is a function $T'_k : N \times \Sigma^{\leqslant k} \to 2^P$, such that for every rule $A \to \varphi$ and $u, v \in \Sigma^*$, for which $u \in L_G(\varphi)$ and v follows A, it holds that $A \to \varphi \in T'_k(A, First_k(uv))$.*

Let us note that the given condition of the membership of a rule in $T'_k(A, x)$ is undecidable already in the simpler case of conjunctive grammars [15], because so is the emptiness problem for these grammars. Therefore, there cannot exist an algorithm for computing the least collection of sets satisfying Definition 9. On the other hand, assuming $T'_k(A, x) = \Sigma^{\leqslant k}$, which would trivially satisfy the definition, is of no use, because the only parsing tables usable with the predictive recursive descent method are deterministic tables of the following kind:

Definition 10 (Deterministic LL(k) table)**.** *Let $|T'_k(A, u)| \leqslant 1$ for all A, u. Then the entries of a deterministic LL(k) table, $T_k : N \times \Sigma^{\leqslant k} \to P \cup \{-\}$, are defined as $T_k(A, u) = A \to \varphi$ if $T'_k(A, u) = \{A \to \varphi\}$, or as $T_k(A, u) = -$ if $T'_k(A, u) = \varnothing$.*

In view of the undecidability of the membership in the least T'_k, any algorithm for constructing suitable T'_k can only produce some supersets of the least collection. The algorithm given below strives to produce as small supersets as possible, and in favourable cases the resulting table will still be deterministic.

The algorithm is modelled upon the well-known method of constructing parsing tables in context-free case [1]. A context-free LL(k) table is constructed on the basis of the sets $\textsc{First}_k(A) = \{First_k(w) \mid w \in L(A)\}$ and $\textsc{Follow}_k(A) = \{First_k(\beta) \mid S \Longrightarrow^* \alpha A\beta\}$, for all $A \in N$, which can be computed by a simple procedure. The definition of $\textsc{First}_k$ can be used for Boolean grammars as it is, an analogue of $\textsc{Follow}_k$ can be defined as $\{First_k(w) \mid w \text{ follows } A\}$ [18], but this does not help much, because in the case of Boolean grammars these sets are not effectively computable [18] due to the aforementioned undecidability of the emptiness problem.

However, since the goal is to construct a superset of the optimal parsing table, this superset can be computed on the basis of supersets of $\textsc{First}_k(s)$ and $\textsc{Follow}_k(s)$. These supersets are called $\textsc{Pfirst}_k(s)$ and $\textsc{Pfollow}_k(s)$, where P stands for *potential*, and the algorithms for computing these sets are included for completeness below. For the proof of their correctness the reader is referred to the cited paper [18].

Algorithm 1 ([18]). *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar. Let $k > 0$. For all $s \in \Sigma \cup N$, compute the set $\textsc{Pfirst}_k(s)$, such that for all $u \in L_G(s)$, $First_k(u) \in \textsc{Pfirst}_k(s)$.*

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 $A \to s_{11} \dots s_{1\ell_1} \& \dots \& s_{m1} \dots s_{m\ell_m} \& \neg\beta_1 \& \dots \& \neg\beta_n \in P$
    $\textsc{Pfirst}_k(A) = \textsc{Pfirst}_k(A) \cup \bigcap_{i=1}^{m} First_k(\textsc{Pfirst}_k(s_{i1}) \cdot \ldots \cdot \textsc{Pfirst}_k(s_{i\ell_i}))$;

It is convenient to extend the definition of $\textsc{Pfirst}$ *to composite expressions as follows:* $\textsc{Pfirst}_k(\psi \mid \xi) = \textsc{Pfirst}_k(\psi) \cup \textsc{Pfirst}_k(\xi)$, $\textsc{Pfirst}_k(\psi \& \xi) = \textsc{Pfirst}_k(\psi) \cap \textsc{Pfirst}_k(\xi)$, $\textsc{Pfirst}_k(\neg\psi) = \Sigma^{\leqslant k}$ *and* $\textsc{Pfirst}_k(\psi\xi) = First_k(\textsc{Pfirst}_k(\psi) \cdot \textsc{Pfirst}_k(\xi))$.

Algorithm 2 ([18]). *For a given strongly non-left-recursive G and for $k > 0$, compute the sets $\textsc{Pfollow}_k(A)$ for all $A \in N$, such 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 every $B \to \pm\beta \in conjuncts(P)$
    for every factorization $\beta = \mu A \nu$ $(\mu, \nu \in (\Sigma \cup N)^*)$
      $\textsc{Pfollow}_k(A) = \textsc{Pfollow}_k(A) \cup First_k(\textsc{Pfirst}_k(\nu) \cdot \textsc{Pfollow}_k(B))$;

Now these sets can be used to construct the $\mathrm{LL}(k)$ parsing table in the same way as in the context-free case:

Algorithm 3. *Let G be a Boolean grammar. Compute $T'_k(A)$ for all $A \in N$.*

for each rule $A \to \alpha_1 \& \dots \& \alpha_m \& \neg\beta_1 \& \dots \& \neg\beta_n \in P$
  for each $x \in \textsc{Pfirst}_k(\alpha_1 \& \dots \& \alpha_m) \cdot \textsc{Pfollow}_k(A)$
    add the rule to $T'_k(A, x)$*;*

Proof of correctness. Using the properties of $\textsc{Pfirst}_k$ and $\textsc{Pfollow}_k$, let us prove that every rule that should be in $T'_k(A, x)$ according to Definition 9 is in fact put there by the algorithm.

Consider a rule

$$A \to \alpha_1 \& \dots \& \alpha_m \& \neg\beta_1 \& \dots \& \neg\beta_n \quad (m \geqslant 1, n \geqslant 0) \tag{4}$$

that should be in $T'_k(A, x)$ in accordance to Definition 9. Then $\varepsilon\langle S \rangle\varepsilon \rightsquigarrow \delta\langle A \rangle\eta$ and $w \in L_G(\varphi\eta)$, where $\varphi = \alpha_1 \& \dots \& \alpha_m \& \neg\beta_1 \& \dots \& \neg\beta_n$ and $x = First_k(w)$.

	$\textsc{Pfirst}_1$	$\textsc{Pfollow}_1$	ε	a	b	c
S	$\{a, b\}$	$\{\varepsilon\}$	$S \to AD\&\neg BC$	$S \to AD\&\neg BC$	$S \to AD\&\neg BC$	$-$
A	$\{\varepsilon, a\}$	$\{\varepsilon, b\}$	$A \to \varepsilon$	$A \to aA$	$A \to \varepsilon$	$-$
B	$\{\varepsilon, a\}$	$\{\varepsilon, b, c\}$	$B \to \varepsilon$	$B \to aBb$	$B \to \varepsilon$	$B \to \varepsilon$
C	$\{\varepsilon, c\}$	$\{\varepsilon\}$	$C \to \varepsilon$	$-$	$-$	$C \to cC$
D	$\{\varepsilon, b\}$	$\{\varepsilon, c\}$	$D \to \varepsilon$	$-$	$D \to bDc$	$D \to \varepsilon$

Table 1: $\textsc{Pfirst}_1$, $\textsc{Pfollow}_1$ and the parsing table for the grammar from Example 1.

Then there exists a factorization $w = uv$, such that $u \in L_G(\varphi)$ and $v \in L_G(\eta)$. By the construction of $\textsc{Pfirst}_k$,

$$First_k(u) \in \textsc{Pfirst}_k(\varphi) \tag{5}$$

Since v follows A,

$$First_k(v) \in \textsc{Pfollow}_k(A) \tag{6}$$

by the construction of $\textsc{Pfollow}_k(A)$.

Concatenating (5) and (6) yields

$$x = First_k(uv) = First_k(First_k(u) \cdot First_k(v)) \in First_k(\textsc{Pirst}_k(\varphi) \cdot \textsc{Pfollow}_k(A)),$$

which means that the rule (4) will be added to $T'_k(A, x)$ at the iteration $((4), x)$. □

Consider the grammar from Example 1, which generates the non-context-free language $\{a^m b^n c^n \,|\, m, n \geqslant 0, m \neq n\}$ The sets $\textsc{Pfirst}_1$ and $\textsc{Pfollow}_1$ and a deterministic LL(1) table for this grammar are shown in Table 1. Note that the string $\varepsilon \in \textsc{Pfirst}_1(S)$ is actually not in $\textsc{First}_1(S)$, because $\varepsilon \notin L(G)$. Consequently, $T(S, d) = S \to KdM$, is is also a fictional entry of the table that could have been replaced with $-$. However, such fictional entries do not prevent the algorithm from being correct.

Since Algorithm 3 constructs supersets of the least sets $T'(A, u)$ satisfying Definition 9, and since these sets must be singletons or empty sets, this algorithm is applicable to a smaller class of grammars than the basic definition. Improving the table construction algorithm and extending its applicability can now be left as a separate problem to study. The rest of this paper is based upon Definition 9 and, accordingly, is not bound by the limitations of Algorithm 3.

5 Recursive descent parser

Having constructed a parsing table, let us now define a recursive descent parser: a collection of procedures that recursively call each other and analyze the input. The procedures will be given as a pseudocode featuring a relatively uncommon flow control construct: the mechanism of exception handling.

A basic exception handling model in the spirit of Ada and C++ is assumed. There is a statement for *raising an exception*, and any compound statement can be supplied with an *exception handler*. If an exception is raised within a compound statement that has an exception handler, then the control is passed to that handler. Otherwise, the return from the

current procedure is forced, and an exception handler is similary searched for in the calling procedure. If there is no exception handler there as well, a return is forced again, and so on, until an exception handler is located. In other words, exceptions are used as a multi-level return statement, which returns to the most recent procedure in the call stack that has an exception handler defined. A recursive descent parser defined in this paper raises exceptions upon syntax errors, while negative conjuncts in the grammar are represented by exception handlers.

Let us define the recursive descent parser. As in the context-free case, it contains a procedure for each terminal and nonterminal symbol in the grammar, and two global variables accessible to all procedures: the input string w and a positive integer p pointing at a position in this string. The latter, by an abuse of programming terminology, will be called a pointer. Each procedure $s()$ corresponding to a symbol $s \in \Sigma \cup N$ starts with some initial value of this pointer, $p = i$, and, after doing some computation and making some recursive calls,

- either returns, setting the pointer to $p = j$ (where $i \leqslant j \leqslant |w|$), thus reporting a successul parse of $w_i \dots w_{j-1}$ from s,
- or raises an exception, which means that a suitable j, such that the symbol s could generate $w_i \dots w_{j-1}$, was not found.

In the latter case the resulting value of the pointer is undefined, though it can be interpreted as the position in the string where a syntax error was encountered.

The procedure corresponding to every terminal $a \in \Sigma$ is defined as

$a()$
{
    if $w_p = a$, then
        $p = p + 1$;
    else
        raise exception;
}

For every nonterminal $A \in \Sigma$ the procedure is

$A()$
{
    switch($T(A, First_k(w_p w_{p+1} \dots))$)
    {
    case $A \to \alpha_1 \& \dots \& \alpha_m \& \neg\beta_1 \& \dots \& \neg\beta_n$:
        (code for the conjunct $A \to \alpha_1$)
        $\vdots$
        (code for the conjunct $A \to \alpha_m$)
        (code for the conjunct $A \to \neg\beta_1$)
        $\vdots$
        (code for the conjunct $A \to \neg\beta_n$)
        return;

```
        case A → ...
          ⋮
        default:
            raise exception;
        }
    }
```

where the code for the first positive conjunct $A \to s_1 \ldots s_\ell$ is

```
    let start = p;              /* omit if this is the only conjunct (m = 1, n = 0) */
    s_1();
    ⋮
    s_ℓ();
    let end = p;                                  /* omit if this is the only conjunct */
```

the code for every consecutive positive conjunct $A \to s_1 \ldots s_\ell$ is

```
    p = start;
    s_1();
    ⋮
    s_ℓ();
    if i ≠ end, then raise exception;
```

and the code for every negative conjunct $A \to \neg s_1 \ldots s_\ell$ is

```
    boolean failed = false;
    try
    {
        p = start;
        s_1();
        ⋮
        s_ℓ();
        if p ≠ end, then raise exception;
    }
    exception handler:
        failed = true;
    if ¬failed, then raise exception;
    p = end;                                  /* if this is the last conjunct in the rule */
```

Note that the compound statement with an exception handler is an exact duplicate of what the code for a positive conjunct $A \to s_1 \ldots s_\ell$ would be. If the inner compound statement successfully returns, the whole fragment raises an exception; on the other hand, if the inner statement raises an exception, that exception is handled, *failed* is set to *true* and the whole fragment successfully returns. Provided that none of the invoked procedures $s_t()$ ever goes into an infinite loop (which will be proved in the next section), this effectively implements negation.

The main procedure is:

```
try
{
    int p = 1;
    S();
    if p ≠ |w| + 1, then raise exception;
}
exception handler:
    Reject;
Accept;
```

So far the algorithm has been defined as a recognizer rather than a parser. It can be turned into a parser by a straightforward modification, which will be given later in Section 8.

6 Proof of the algorithm's correctness

It should be proved that the algorithm (a) always terminates, and (b) accepts a string if and only if this string is in the language. The termination of the algorithm can be proved under the sole assumption of strong non-left-recursiveness of the grammar, without making any claims on the mechanism of choosing the rules T. This allows abstracting from the goal of parsing and even from the semantics of the grammar, concentrating on the general structure of the computation.

Lemma 1. *Let $G = (\Sigma, N, P, S)$ be an arbitrary Boolean grammar, and consider the conjunctive grammar $G_+ = (\Sigma, N, P_+, S)$. Let $k \geqslant 1$; let $T : N \times \Sigma^{\leqslant k} \to P \cup \{-\}$ be an arbitrary function (with the sole assumption that $T(A, u)$ always gives a rule for A or $-$), let the set of procedures be constructed with respect to G and T. Then*

I. For every $s \in \Sigma \cup N$ and $u, v \in \Sigma^*$, if $A()$ invoked on the input uv returns, consuming u, then $u \in L_{G_+}(A)$.

II. For every $A, B \in N$ and $u, v \in \Sigma^*$, if $A()$ is invoked on uv, and the resulting computation eventually leads to a call to $B()$ on the input v, then $\varepsilon\langle A\rangle\varepsilon \rightsquigarrow \gamma\langle B\rangle\delta$, where $u \in L_{G_+}(\gamma)$.

Proof. The first part of the lemma is proved inductively on the height of the tree of recursive calls made by $A()$ on the input uv. Since $A()$ terminates by the assumption, this tree is finite and its height is well-defined.

Basis: $s()$ makes no recursive calls and returns. If $s = a \in \Sigma$ and $a()$ returns on uv, consuming u, then $u = a$ and obviously $a \in L_{G_+}(a)$.

If $s = A \in N$ and $A()$ returns without making any recursive calls, then the rule chosen upon entering $A()$ may contain one positive conjunct, $A \to \varepsilon$, and possibly a negative conjunct $A \to \neg\varepsilon$. Then $u = \varepsilon$ and $\varepsilon \in L_{G_+}(A)$.

Induction step. Let $A()$ return on uv, consuming u, and let the height of the tree of recursive calls made by $A()$ be h. The first thing $A()$ does is looking up $T(A, First_k(uv))$, to find a rule

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \quad (m \geqslant 1,\ n \geqslant 0,\ \alpha_i, \beta_i \in (\Sigma \cup N)^*) \tag{7}$$

there. Then the code fragments for all conjuncts of the rule are executed.

Consider every positive conjunct $A \to \alpha_t$, and let $\alpha_t = s_1 \dots s_\ell$. Then the code $s_1(); \dots; s_\ell()$ is executed on uv, and it returns, consuming u. Consider a factorization $u = u_1 \dots u_\ell$ defined by the positions of the pointer to the input after each $s_i()$ returns and before the next $s_{i+1}()$ is called. Thus each $s_i()$ returns on $u_i u_{i+1} \dots u_\ell v$, consuming u_i, and the height of recursive calls made by $s_i()$ does not exceed $h-1$. By the induction hypothesis, $u_i \in L_{G_+}(s_i)$. Concatenating the latter for all i, the following is obtained:

$$u = u_1 \dots u_\ell \in L_{G_+}(s_1) \dots L_{G_+}(s_\ell) = L_{G_+}(\alpha_t) \tag{8}$$

Now use (8) for all t to produce the following derivation in the conjunctive grammar G_+:

$$A \overset{G_+}{\Longrightarrow} (\alpha_1 \& \dots \& \alpha_m) \overset{G_+}{\Longrightarrow} \dots \overset{G_+}{\Longrightarrow} (u \& \dots \& u) \overset{G_+}{\Longrightarrow} u, \tag{9}$$

thus proving the induction step.

Turning to the second part of the lemma, if $A()$ starts with input uv, and $B()$ is called on v at some point of the computation, then consider the partial tree of recursive calls made up to this point. Let h be the length of the path from the root to this last instance of $B()$. The proof is an induction on h.

Basis $h = 0$**.** If $A()$ coincides with $B()$, and thus $B()$ is called on the same string $uv = v$, then $u = \varepsilon$, $\varepsilon\langle A\rangle\varepsilon \rightsquigarrow \varepsilon\langle A\rangle\varepsilon$ and $u = \varepsilon \in L_{G_+}(\varepsilon)$.

Induction step. $A()$, called on uv, begins with determining a rule (7) using $T_k(A, First_k(uv))$ and then proceeds with calling the subroutines corresponding to the symbols in the right hand side of (7). Some of these calls terminate (return or, in the case of negative conjuncts, raise exceptions that are handled inside $A()$), while the last one recorded in our partial computation history leads down to $B()$. Let $A \to \pm\gamma C\delta$ ($\gamma C\delta \in \{\alpha_t\}$ or $\gamma C\delta \in \{\beta_t\}$) be the unsigned conjunct in which this happens, and $C()$ be this call leading down. Consider a factorization $u = xy$, such that $C()$ is called on yv.

Let $\gamma = s_1 \dots s_\ell$. The call to $C()$ is preceded by the calls to $s_1(); \dots; s_\ell()$, where each $s_i()$ is called on $x_i \dots x_\ell yv$ and returns, consuming x_i (where $x = x_1 \dots x_\ell$). By part I of this lemma, this implies $x_i \in L_{G_+}(s_i)$, and hence $x \in L_{G_+}(\gamma)$.

By the existence of the unsigned conjunct $A \to \pm\gamma C\delta$, $\varepsilon\langle A\rangle\varepsilon \rightsquigarrow \gamma\langle C\rangle\delta$. For the partial computation of $C()$ on yv (up to the call to $B()$), the distance between $C()$ and $B()$ is $h-1$, which allows applying the induction hypothesis to conclude that $\varepsilon\langle C\rangle\varepsilon \rightsquigarrow \mu\langle B\rangle\eta$, such that $y \in L_{G_+}(\mu)$.

Combining these two reachabilities according to Definition 7, we get $\varepsilon\langle A\rangle\varepsilon \rightsquigarrow \gamma\mu\langle B\rangle\eta\delta$, while the two conjunctive derivations in G_+ can be merged to obtain $u \in L_{G_+}(\gamma\mu)$. □

Lemma 2. *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar. Let $k \geqslant 1$; let $T : N \times \Sigma^{\leqslant k} \to P \cup \{-\}$ be an arbitrary function, with the sole assumption that $T(A, u)$ always gives a rule for A or $-$. Let the set of procedures be constructed with respect to G*

and T. Then, for every $s \in \Sigma \cup N$ and $w \in \Sigma^*$, the procedure $s()$ terminates on the input w, either by consuming a prefix of w and returning, or by raising an exception.

Proof. Suppose there exists $s \in \Sigma \cup N$ and w, such that $s()$ does not halt on the input w. Consider the (infinite) tree of recursive calls, the nodes of which are labeled with pairs (t, u), where $t \in \Sigma \cup N$ and u is some suffix of w.

Since the right hand side of every rule is finite, every procedure makes finitely many recursive calls, and the tree has finite branching. Hence, by König's lemma, this tree should contain an infinite path

$$(A_1, u_1), (A_2, u_2), \ldots, (A_i, u_i), \ldots \tag{10}$$

where $(A_1, u_1) = (s, w)$, $A_i \in N$ and each procedure $A_i()$ is invoked on u_i and, after calling some procedures that terminate, eventually calls A_{i+1} on the string u_{i+1}, which is a suffix of u_i. This means that $|u_1| \leqslant |u_2| \leqslant \ldots \leqslant |u_i| \leqslant \ldots$

Since w is of finite length, it has finitely many different suffixes, and the decreasing second component in (10) should converge to some shortest reached suffix of w. Denote this suffix as u, and consider any node (A, u) that appears multiple times on the path (10). Consider any two instances of (A, u); then, by Lemma 1, $\varepsilon\langle A\rangle\varepsilon \rightsquigarrow \gamma\langle A\rangle\delta$, where $\varepsilon \in L_{G_+}(\gamma)$. This contradicts the assumption that G is strongly non-left-recursive. □

Now the correctness of the algorithm can be established. Before approaching the proof, let us consider the following example showing that the parser may in some cases behave contrary to the grammar, and hence its correctness is not that obvious.

Example 2. *Let $\Sigma = \{a, b\}$ and consider the following Boolean grammar the corresponding sets* $\textsc{Pfirst}_1$ *and* $\textsc{Pfollow}_1$*, and the resulting LL(1) table.*

$$\begin{array}{lcl} S & \to & Ab \\ A & \to & B\&\neg bC \\ B & \to & a \mid b \\ C & \to & \varepsilon \end{array}$$

	$\textsc{Pfirst}_1$	$\textsc{Pfollow}_1$	ε	a	b
S	$\{\varepsilon, a, b\}$	$\{\varepsilon\}$	–	$S \to Ab$	$S \to Ab$
A	$\{a, b\}$	$\{b\}$	–	$A \to B\&\neg bC$	$A \to B\&\neg bC$
B	$\{a, b\}$	$\{b\}$	–	$B \to a$	$B \to b$
C	$\{\varepsilon\}$	$\{b\}$	–	–	$C \to \varepsilon$

in which $L(B) = \{a, b\}$, $L(C) = \{\varepsilon\}$, $L(A) = \{a\}$ and $L(S) = \{ab\}$. However, the procedure $A()$ returns on the input ba, consuming b, even though $b \notin L(A)$.

Consider that $A()$ executed on ba first invokes $B()$ on ba (which successfully returns, consuming b), and then, for the second conjunct, invokes $C()$ on a. Instead of returning and consuming ε, the procedure $C()$ raises an exception because of an unexpected lookahead; this exception is handled by the code within $A()$, which considers that the negative conjunct $A \to \neg bC$ has failed, and therefore the entire rule has been matched. The pointer is then set where the earlier called instance of $B()$ has set it, and $A()$ successfully returns, consuming a prefix not in $L(A)$.

Even though this kind of behaviour looks quite suspicious, in the end, no wrong parses are obtained: $S()$ rejects ba, and in fact $S()$ returns on a string $w \in \Sigma^*$, consuming the entire w, if and only if $w = ab$. As it will now be proved, the algorithm is correct, though the proof of its correctness requires a more precise formulation than in the context-free and conjunctive cases:

Lemma 3. *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar. Let $k \geqslant 1$. Let $T : N \times \Sigma^{\leqslant k} \to P \cup \{-\}$ be a deterministic LL(k) table for G, let the set of procedures be constructed with respect to G and T. Let $y, z \in \Sigma^*$ and $s_1, \ldots, s_\ell \in \Sigma \cup N$ ($\ell \geqslant 0$), and assume there exists $\widehat{z} \in \Sigma^*$, such that $\widehat{z}$ follows $s_1 \ldots s_\ell$ and $First_k(\widehat{z}) = First_k(z)$. Then the code $s_1(); \ldots; s_\ell()$ returns on the input yz, consuming y, if and only if $y \in L_G(s_1 \ldots s_\ell)$.*

Proof. According to Lemma 2, the code $s_1(); \ldots; s_\ell()$ terminates in this or that way, that is, by returning or by raising an exception. Consider the tree of recursive calls of the code $s_1(); \ldots; s_\ell()$ executed on the input yz, as in Lemma 2. This tree is finite; let h be its height.

The proof is an induction on the pair (h, ℓ), where pairs are ordered as $(h, \ell) \prec (h', \ell')$ if $h < h'$ or if $h = h'$ and $\ell < \ell'$. Besides the trivial base case $h = \ell = 0$, the natural base case would be $h = 0, \ell = 1$; to simplify the presentation of the proof, the case of $A()$ is handled together with induction step (without referring to the induction hypothesis), while $a()$ is formulated as the basis.

Basis I: $(0, 0)$**, that is,** $s_1 \ldots s_\ell = \varepsilon$**.** Obviously, $L_G(\varepsilon) = \{\varepsilon\}$, and accordingly the empty statement returns on yz, consuming y, if and only if $y = \varepsilon$.

Basis II: $(0, 1)$ **and** $s \in \Sigma$**.** The procedure $a()$ is constructed so that it returns on the input yz, consuming y, if and only if $y = a$. This is equivalent to $y \in L_G(a) = \{a\}$.

Induction step I: $(h - 1, \ldots) \to (h, 1)$**, or** $(0, 1)$ **and** $s \in N$**.** Let $\ell = 1$ and $s_1 = A \in N$, let $\widehat{z}$ follow A, with $First_k(\widehat{z}) = First_k(z)$, and let h be the height of the tree of recursive calls made by $A()$ executed on yz.

$\Leftrightarrow$ If $A()$ returns on yz, consuming y, then $T(A, First_k(yz))$ gives some rule

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \tag{11}$$

(where $m \geqslant 1$, $n \geqslant 0$), and then the following computations take place:

1. For every positive conjunct $A \to \alpha_i$ ($\alpha_i = s_1 \ldots s_\ell$), the code $s_1(); \ldots; s_\ell()$ is called on yz. It returns, consuming y.

   Since the computation of $s_1(); \ldots; s_\ell()$ on yz is a subcomputation of the computation of $A()$ on yz, the height of the tree of recursive calls corresponding to this subcomputation does not exceed $h - 1$. The string $\widehat{z}$ follows $s_1 \ldots s_\ell$ just because $\widehat{z}$ follows A and $A \to \pm s_1 \ldots s_\ell \in conjuncts(P)$. Hence, by the induction hypothesis, $y \in L_G(\alpha_i)$.
2. For each negative conjunct $A \to \neg\beta_j$ ($\beta_j = s_1 \ldots s_\ell$), the code $s_1(); \ldots; s_\ell()$ is invoked on yz. It either raises an exception, or returns, consuming a prefix other than y: putting together, it is not the case that this code returns, consuming y.

   Similarly to the previous case, this computation has to be a part of the computation of $A()$, hence the depth of recursion is at most $h - 1$; again, $\widehat{z}$ follows $s_1 \ldots s_\ell$. This allows one to invoke the induction hypothesis to obtain that $y \notin L_G(\beta_i)$.

Combining the results for individual conjuncts, $y \in L_G(\alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n)$ and thus $y \in L_G(A)$.

$\Leftarrow$ If $y \in L_G(A)$, then there exists a rule (11), such that

$$y \in L_G(\alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n) \tag{12}$$

Since $\widehat{z}$ follows A, $\varepsilon\langle S\rangle\varepsilon \rightsquigarrow \delta\langle A\rangle\eta$, where $\widehat{z} \in L_G(\eta)$. Combining (12) with this, we obtain that $y\widehat{z} \in L_G(\varphi\eta)$, where $\varphi = \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n$. Then, by Definition 9, $T'(A, First_k(y\widehat{z}))$ contains the rule (11). Since $First_k(y\widehat{z}) = First_k(yz)$, (11) is in $T'(A, First_k(yz))$, and therefore, by the definition of a deterministic LL(k) table, this rule is given by $T(A, First_k(yz))$.

Hence the computation of $A()$ on yz starts from choosing the alternative (11). Consider all the conjuncts of the rule (11) in the order of the corresponding code fragments, and let us prove that each of these fragments is successively passed:

1. For each positive conjunct $A \to \alpha_i$ ($\alpha_i = s_1 \ldots s_\ell$), $y \in L_G(s_1 \ldots s_\ell)$ by (12) and $\widehat{z}$ follows $s_1 \ldots s_\ell$ (since $\widehat{z}$ follows A and $A \to \pm s_1 \ldots s_\ell \in conjuncts(P)$). Therefore, by the induction hypothesis, the code $s_1(); \ldots; s_\ell()$ returns on yz, consuming y.
2. For every negative conjunct $A \to \neg\beta_j$ ($\beta_j = s_1 \ldots s_\ell$), since $y \notin L_G(s_1 \ldots s_\ell)$ and $\widehat{z}$ follows $s_1 \ldots s_\ell$, by the induction hypothesis, it is not the case that the code $s_1(); \ldots; s_\ell()$ returns on yz, consuming y. On the other hand, by Lemma 2, the code $s_1(); \ldots; s_\ell()$ terminates on yz, either by returning or by raising an exception. This implies that $s_1(); \ldots; s_\ell()$, invoked on the input yz, either returns, consuming a prefix other than y, or raises an exception. In the former case an exception is manually triggered in the code fragment corresponding to β_j. Thus an exception is effectively raised in both cases. The exception handler included in the code fragment sets a local variable $failed$ to true, and in this way the whole code fragment terminates without raising unhandled exceptions.

In this way all the conjuncts are successfully handled; the final assignment in the code for the alternative (11) restores the pointer to the location where it was put by the code for the first conjunct. Then $A()$ returns, having thus consumed exactly y.

Induction step II: $(h, \ell - 1) \to (h, \ell)$**.** Let $\ell \geqslant 2$, let $s_1 \ldots s_\ell \in (\Sigma \cup N)^*$ and let $\widehat{z}$ follow $s_1 \ldots s_\ell$, where $First_k(\widehat{z}) = First_k(z)$.

$\Rightarrow$ Let the code $s_1(); \ldots; s_{\ell-1}(); s_\ell()$ return on input yz, consuming y. Consider the value of the pointer to the input string after $s_{\ell-1}()$ returns and before $s_\ell()$ is called; this value defines a factorization $y = uv$, such that the code $s_1(); \ldots; s_{\ell-1}()$ returns on uvz, consuming u, while the procedure $s_\ell()$ returns on vz, consuming v. The height of the recursive calls in these subcomputations obviously does not exceed that of the whole computation.

Since $\widehat{z}$ follows s_ℓ, the induction hypothesis is directly applicable to the computation of $s_\ell()$ on vz, yielding

$$v \in L_G(s_\ell) \tag{13}$$

In order to use the induction hypothesis for the former $\ell - 1$ calls, first it has to be established that the string $v\widehat{z}$ follows $s_1 \ldots s_{\ell-1}$. We know that $\widehat{z}$ follows $s_1 \ldots s_\ell$, which

means that $\varepsilon\langle S\rangle\varepsilon \rightsquigarrow \delta\langle s_1 \dots s_\ell\rangle\eta$, where $\widehat{z} \in L_G(\eta)$. Hence, $\varepsilon\langle S\rangle\varepsilon \rightsquigarrow \delta\langle s_1 \dots s_{\ell-1}\rangle s_\ell\eta$, while concatenating (13) with $\widehat{z} \in L_G(\eta)$ yields $v\widehat{z} \in L_G(s_\ell\eta)$.

Since $First_k(v\widehat{z}) = First_k(vz)$, now the induction hypothesis can be used for the computation of $s_1(); \dots; s_\ell()$ on uvz (which returns, consuming u), giving

$$u \in L_G(s_1 \dots s_{\ell-1}) \tag{14}$$

By (14) and (13), $y = uv \in L_G(s_1 \dots s_{\ell-1}s_\ell)$.

$\Leftarrow$ Conversely, if $y \in L_G(s_1 \dots s_{\ell-1}s_\ell)$, then there exists a factorization $y = uv$, such that $u \in L_G(s_1 \dots s_{\ell-1})$ and $v \in L_G(s_\ell)$.

The computation of $s_1(); \dots; s_{\ell-1}()$ on uvz is obviously a subcomputation of the computation of $s_1(); \dots; s_{\ell-1}(); s_\ell()$. Hence, the recursion depth for the subcomputation does not exceed that for the whole computation. On the other hand, since $v \in L_G(s_\ell)$ and $\widehat{z}$ follows $s_1 \dots s_{\ell-1}s_\ell$, $v\widehat{z}$ follows $s_1 \dots s_{\ell-1}$ (established as in the previous part of the proof). This allows applying the induction hypothesis to this subcomputation and obtaining that $s_1(); \dots; s_{\ell-1}()$ returns on uvz, consuming u.

Once the subcomputation $s_1(); \dots; s_{\ell-1}()$ returns on uvz, the computation of $s_1(); \dots; s_{\ell-1}(); s_\ell()$ proceeds with invoking $s_\ell()$ on vz. Hence, $s_\ell()$ on vz is also a subcomputation, which has height no greater than that of the whole computation. Since $\widehat{z}$ follows s_ℓ, the induction hypothesis is now applicable, and $v \in L_G(s_\ell)$ implies that $s_\ell()$ returns on vz, consuming v.

Therefore, the sequential composition of these two computations, $s_1(); \dots; s_{\ell-1}(); s_\ell()$, returns on yz, consuming $uv = y$. □

Now let $y = w$, $z = \widehat{z} = \varepsilon$, $\ell = 1$ and $s_1 = S$. Then the conditions of Lemma 3 are satisfied, and the following result is obtained:

Corollary 1. *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar. Let $k \geqslant 1$. Let $T : N \times \Sigma^{\leqslant k} \to P \cup \{-\}$ be a deterministic LL(k) table for G, let the set of procedures be constructed with respect to G and T.*

Then, for every string $w \in \Sigma^*$, the procedure $S()$ executed on w

- *Returns, consuming the whole input, if $w \in L(G)$;*
- *Returns, consuming less than the whole input, or raises an exception, if $w \notin L(G)$.*

Out of this there follows the statement on the correctness of the algorithm:

Theorem 2. *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar. Let $k \geqslant 1$. Let $T : N \times \Sigma^{\leqslant k} \to P \cup \{-\}$ be a deterministic LL(k) table for G, let a recursive descent parser be constructed with respect to G and T. Then, for every string $w \in \Sigma^*$, the parser, executed on w, terminates and accepts if $w \in L(G)$, rejects otherwise.*

7 Complexity of the algorithm

The definition of a recursive descent parser given in Section 5 is an implementation in itself, so its complexity can be directly estimated. It is not hard to establish the following asymptotical bounds:

Lemma 4. *Under the conditions of Theorem 2, the depth of recursion in course of the computation never exceeds* $(|w|+1)\cdot|N|+1$.

Proof. Suppose that the tree of recursive calls contains a path of at least $(|w|+1)\cdot|N|+2$ nodes. This implies that at least the first $(|w|+1)\cdot|N|+1$ nodes in this path are nonterminals. As in Lemma 2, let us denote these procedure calls by

$$(A_0, u_0), (A_1, u_1), \ldots, (A_{(|w|+1)\cdot|N|}, u_{(|w|+1)\cdot|N|}) \tag{15}$$

where $s_0 = S$, $u_0 = w$, and for each $m > 0$, $A_m \in N$ and u_m is a suffix of w. Since there exist only $(|w|+1)\cdot|N|$ distinct pairs of this form, the pairs (15) cannot be pairwise distinct, and $A_i = A_j$ and $u_i = u_j$ for some $i < j$. That is, the computation of $A_i()$ on u_i eventually leads to a call to $A_i()$ on u_i, and, according to Lemma 1 (part I), there is a left recursion in the grammar, which contradicts the assumption. □

Theorem 3. *Under the conditions of Theorem 2, the parser, implemented on a random access machine, uses space* $O(n)$ *and works in time* $2^{O(n)}$*, where* $n = |w|$*. Both bounds are precise.*

Proof. Consider the execution state of the parser. It is comprised of the value of the pointer, the line currently being executed, the values of variables *start* and *end*, as well as the execution stack, which contains a return address for each procedure call and the caller's local variables (*start* and *end*). Using the estimation of the recursion depth from Lemma 4, this sums up to $3\cdot(|w|+1)\cdot|N| + const$ memory cells, which is enough to prove the first claim.

To prove the exponential upper bound for time, consider that when a procedure $A()$ is called, it uses at most $|G|$ time for internal processing and makes at most $|G|$ calls to other procedures, while $a()$ executes two statements. This implies that at most $|G|^{(|w|+1)\cdot|N|+1}$ procedures are executed in total, each of them using at most $|G|$ time internally. With the size of the grammar taken as a constant, this is $2^{O(n)}$.

This upper bound is reached on the following grammar for a^+ [15]:

$S \to AS\&BS \mid \varepsilon$
$A \to a$
$B \to a$

Indeed, $S()$ executed on any nonempty string calls two instances of $S()$ on a suffix one symbol shorter, hence the execution time of $S()$ on a^n $(n \geqslant 1)$ is greater than twice the execution time of $S()$ on a^{n-1}, and a 2^n lower bound follows. □

Exponential time complexity is caused by recomputing the same procedures with the same value of the pointer in different branches of the computation. This can be easily avoided by

using a simple programming technique is known as *memoization*. This technique, which was first used for parsing by Norvig [13], consists of storing the result of the first computation in memory, and then looking it up for every subsequent call to the same procedure with the same arguments. In our case this simple modification is enough to reduce the complexity to linear.

For this purpose the *memoized recursive descent* uses an array $m[A][i]$ indexed by a pair of a nonterminal and a position in the string. Each element belongs to $\{1, \ldots, n, n+1\} \cup \{\mathit{undefined}, \mathit{error}\}$. If $m[A][i] = j$, this means that $A()$ executed with the pointer at i returns, setting the pointer at j. In the case $m[A][i] = \mathit{error}$, the call to $A()$ with $p = i$ is recorded as having raised an exception. Finally, $m[A][i] = \mathit{undefined}$ means that $A()$ has not been called with $p = i$ so far. In the beginning, $m[A][i] = \mathit{undefined}$ for all $A \in n$, $1 \leqslant i \leqslant n+1$.

For every nonterminal A, the procedure $A()$ will be redefined as follows:

```
A()
{
    if m[A][p] ∈ {1, ..., n, n+1}
        p = m[A][p];
    else if m[A][p] = error;
        raise an exception;
    else
    {
        let start = p;
        m[A][start] = error;
        (the original code for A() given in Section 5)
        m[A][start] = p;
    }
}
```

Consider the computation of this procedure. If $A()$ has never been called with this value of the pointer before, the value $m[A][i]$ (where $i = p$) is speculatively set to *error*, and the code from Section 5 is invoked to perform the recursive descent. That code will not access the value of $m[A][i]$, because that can happen only if the parser enters an infinite loop, which is not possible by Lemma 2. If the invoked code successfully returns, $m[A][i]$ is reassigned with an appropriate value. If the invoked code raises an exception, it is not handled in the new code for $A()$, and hence $A()$ raises an exception, while $m[A][i]$ retains the earlier assigned value *error*. Every subsequent call to $A()$ with the same value of the pointer will obtain the result of the first computation from $m[A][i]$, and replicate the earlier behaviour.

Let us take the correctness of the memoization technique for granted. Then this modification of the recursive descent parser satisfies the correctness statements of Lemma 3 and Theorem 2. In order to argue for its linear time complexity, let us first note that in course of a computation of a string w, every line of the "original code" in every procedure $A()$ is executed at most $|G| \cdot (|w| + 1)$ times. Furthermore, each time the "new code" in $A()$ or the code in a procedure $a()$ is executed, this is done because the corresponding procedures were called by the "original code" of some $B()$, hence the number of times each of these lines is executed is bounded by the same value.

Theorem 4. *The memoized recursive descent parser, implemented on a random access machine, works in time $O(n)$, where n is the length of the input string.*

8 Constructing parse trees

Let us now modify the recognizer defined in Section 5 to construct a parse tree of the input string.

Parse trees in the case of Boolean grammars [16] are, strictly speaking, finite acyclic graphs rather than trees. Let $w = a_1 \dots a_{|w|}$ be a string generated by a grammar $G = (\Sigma, N, P, S)$. A parse tree of w contains a leaf labelled a_i for every i-th position in the string, while the rest of the vertices are labelled with rules from P. The subtree accessible from any given vertex of the tree contains leaves in the range between $i + 1$ and j, and thus corresponds to a substring $a_{i+1} \dots a_j$. In particular, each leaf a_i corresponds to itself.

For each vertex labelled with a rule

$$A \to \alpha_1 \& \dots \& \alpha_m \& \neg\beta_1 \& \dots \& \neg\beta_n$$

and associated to a substring $a_{i+1} \dots a_j$, the following conditions should hold:

1. It should have exactly $|\alpha_1 \dots \alpha_m|$ linearly ordered direct descendants corresponding to the symbols in positive conjuncts. For each nonterminal in $\alpha_1 \dots \alpha_m$, the corresponding descendant must be labelled with some rule for that nonterminal, and for each terminal $a \in \Sigma$, the descendant should be a leaf labelled with a.

2. For each t-th positive conjunct of this rule, let $\alpha_t = s_1 \dots s_\ell$. There should exist numbers $i_1, \dots, i_{\ell-1}$, where $i = i_0 \leqslant i_1 \leqslant \dots \leqslant i_{\ell-1} \leqslant i_\ell = j$, such that each descendant corresponding to s_r encompasses the substring $a_{i_{r-1}+1} \dots a_{i_r}$.

3. For each t-th negative conjunct of this rule, it should hold that $a_{i+1} \dots a_j \notin L_G(\beta_t)$.

The root is the unique node with no incoming arcs; it should be labelled with any rule for the start symbol and all leaves should be reachable from it. The subtree accessible from any node corresponding to a substring $u = a_{i+1} \dots a_j$ is called a parse tree of u from the symbol it is labelled with.

Note that the information on negative conjuncts in not reflected in the tree, and actually no reasonable way to include it is known [16]. Such a tree looks as if a tree for a conjunctive grammar [14]; for the grammar in Example 1 the corresponding trees even resemble context-free parse trees. However, the above condition 3 forbids all trees that do not comply to the Boolean grammar.

To construct such a tree, a recursive descent parser requires minimal modifications: the parse tree will be isomorphic to a fragment of the tree of recursive calls in the computation of the parser. A new data type of a tree node is introduced, which contains the label and either a position in the input (in case of a terminal), or pointers to descendant nodes (in case of a nonterminal). Each procedure $s()$, where $s \in \Sigma \cup N$, returns a value of this type. The procedure corresponding to every terminal $a \in \Sigma$ is redefined as follows:

```
Tree a()
{
    if w_p = a, then
    {
        p = p + 1;
        return a new leaf labelled a and numbered p;
    }
    else
        raise exception;
}
```

For every procedure $A()$, the code implementing each rule

$$A \to s_{11} \dots s_{1\ell_1} \& \dots \& s_{m1} \dots s_{m\ell_m} \& \neg\beta_1 \& \dots \& \neg\beta_n \tag{16}$$

undergoes the following changes. First, every call to every procedure $s_{ij}()$ shall store the return value in a local variable:

Tree $T_{ij} = s_{ij}()$;

Second, once all conjuncts are checked, the return statement assembles a new tree as follows:

Return a new node labelled (16), with pointers to $T_{11}, T_{12}, \dots, T_{m\ell_m}$;

The proof of the correctness of this approach basically constitutes a remark to the proof of Lemma 3.

Lemma 5. *As in Lemma 3, let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar, let $k \geqslant 1$, let $T : N \times \Sigma^{\leqslant k} \to P \cup \{-\}$ be a deterministic LL(k) table for G, and consider the set of procedures for G and T modified for creating a parse tree.*

Then, for every $s \in \Sigma \cup N$, $y \in L_G(s)$ and $z \in \Sigma^*$, such that there exists $\widehat{z}$ that follows s and $First_k(\widehat{z}) = First_k(z)$, the procedure $s()$ executed on yz returns a parse tree of y from s.

Proof. The modified recursive descent behaves in the same way as the original, with the added functionality of returning some values. Thus Lemma 3 applies, and it follows that $s()$ returns on yz, consuming y. The statement on the constructed tree is proved by induction on the height of the tree of recursive calls.

In the base case, $s = a \in \Sigma$, the procedure $a()$, under the assumption that it returns, clearly returns the appropriate leaf.

Let $s = A \in N$. Then $A()$ executed on yz starts with choosing a rule (16), and then, for every i-th positive conjunct $A \to s_{i1} \dots s_{i\ell_i}$, the procedures $s_{ij}()$ are called for $j = 1, \dots, \ell_i$ in a row. Following the argument in the proof of Lemma 3, there exists a factorization $y = y_{i1} \dots y_{i\ell_i}$, such that each $s_{ij}()$ is called on $y_{ij} y_{i,j+1} \dots y_{i\ell_i} z$ and returns, consuming y_{ij}. At the same time, $y_{i,j+1} \dots y_{i\ell_i} z$ follows s_{ij}, and hence $y_{ij} \in L_G(s_{ij})$.

Therefore, by the induction hypothesis, each procedure $s_{ij}()$ returns a parse tree of y_{ij} from s_{ij}. Then the return statement of $A()$ constructs a correct parse tree by definition. □

The tree constructed in this way has one additional property: the in-degree of every non-root node labelled with a rule is 1. That is, if the same substring is parsed multiple times from the same nonterminal, the parse subtrees will not be shared. However, this problem can be overcome by memoization in the same way as described in Section 7, and the resulting parser will share the subtrees as much as possible.

9 Conclusion

A generalization of the most well-known parsing method has been devised. It retains much of the simplicity of the context-free recursive descent, but achieves a higher expressive power. In contrast to other extensions to recursive descent used in software engineering, the proposed method has a theoretical basis represented by Boolean grammars, and the operators in a grammar are logical connectives with a clear semantics. Recursive descent is thus backed up by other completely different parsing algorithms for Boolean grammars [12, 16, 18], which implement the same functionality.

The applicability of the Boolean recursive descent to practical parsing is worth being investigated. Note that the proposed changes to the basic recursive descent method are "orthogonal" to the common practical techniques for engineering context-free recursive descent parsers [19]. For instance, while this paper considered only the deterministic case of recursive descent known as *predictive parsing*, the Boolean recursive descent with backtracking could be defined and proved correct by analogy with the context-free case. So it should not be hard to implement Boolean grammars in the existing software.

Another direction of further study concerns the subclass of Boolean grammars to which the recursive descent parsing is applicable. Let us call them *LL(k) Boolean grammars*, and consider the family of *LL(k) Boolean languages* generated by such grammars. This family obviously contains all LL(k) context-free languages, as well as some non-context-free languages, see Example 1. It is possible to prove (and will be proved in an upcoming paper) that Boolean LL(k) languages over a one-letter alphabet are regular, while Boolean grammars of the general form can generate the nonregular language $\{a^{2^n} \mid n \geqslant 0\}$ [16]. However, it remains unknown whether any practically relevant languages cannot be parsed by Boolean recursive descent, and, in particular, whether there exists any context-free language that is not Boolean LL(k).

Further questions about the family of Boolean LL(k) languages can be naturally asked. For instance, do there exist any languages that are Boolean LL($k+1$), but not Boolean LL(k), for some $k > 0$? (cf. Kurki-Suonio [10]) Developing a complete theory of Boolean LL(k) languages similar to the context-free LL theory [8, 10, 11, 20, 21] is a good task for future theoretical research.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: principles, techniques and tools*, Addison-Wesley, Reading, Mass., 1986.

[2] A. Birman, J. D. Ullman, "Parsing algorithms with backtrack", *Information and Control*, 23:1 (1973), 1–34.

[3] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation", *Proceedings of POPL 2004* (Venice, Italy, January 14–16, 2004), 111–122.

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

[5] S. A. Greibach, "A new normal-form theorem for context-free phrase structure grammars", *Journal of the ACM*, 12 (1965), 42–52.

[6] D. Grune, C. J. H. Jacobs, "A programmer-friendly LL(1) parser generator", *Software–Practice and Experience*, 18:1 (1988), 29–38.

[7] J. Ichbiah, J. G. P. Barnes, R. J. Firth, M. Woodger, *Rationale for the Design of the Ada Programming Language*, Cambridge University Press, 1991.

[8] D. E. Knuth, "Top-down syntax analysis", *Acta Informatica*, 1 (1971), 79–110.

[9] V. Kountouriotis, Ch. Nomikos, P. Rondogiannis, "Well-founded semantics for Boolean grammars", *Developments in Language Theory* (DLT 2006, Santa Barbara, USA, June 26–29, 2006), LNCS 4036, 203–214.

[10] R. Kurki-Suonio, "Notes on top-down languages", *BIT*, 9 (1969), 225–238.

[11] P. M. Lewis, R. E. Stearns, "Syntax-directed transduction", *Journal of the ACM*, 15:3 (1968), 465–488.

[12] A. Megacz, "Scannerless Boolean parsing", *Electronic Notes in Theoretical Computer Science*, 164:2 (2006), 97–102.

[13] P. Norvig, "Techniques for automatic memoization with applications to context-free parsing", *Computational Linguistics*, 17:1 (1991), 91–98.

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

[15] A. Okhotin, "Top-down parsing of conjunctive languages", *Grammars*, 5:1 (2002), 21–40.

[16] A. Okhotin, "Boolean grammars", *Information and Computation*, 194:1 (2004), 19–48.

[17] A. Okhotin, "The dual of concatenation", *Theoretical Computer Science*, 345:2–3 (2005), 425–447.

[18] A. Okhotin, "Generalized LR parsing algorithm for Boolean grammars", *International Journal of Foundations of Computer Science*, 17:3 (2006), 629–664.

[19] T. J. Parr, R. W. Quong, "ANTLR: a predicated-LL(k) parser generator", *Software–Practice and Experience*, 25:7 (1995), 789–810.

[20] D. J. Rozenkrantz, R. E. Stearns, "Properties of deterministic top-down grammars", *Information and Control*, 17 (1970), 226–256.

[21] D. Wood, "The theory of left factored languages" (I, II), *Computer Journal*, 12:4 (1969), 349–356; 13:1 (1970) 55–62.

[22] M. Wrona, "Stratified Boolean grammars", *Mathematical Foundations of Computer Science* (Proceedings of MFCS 2005, Gdansk, Poland, August 29–September 2, 2005), LNCS 3618, 801–812.

Appendix: proof of Theorem 1

This appendix presents a proof of Theorem 1, which states that every strongly non-left-recursive Boolean grammar satisfies the well-formedness condition given in Definition 4.

Lemma 6. *For every strongly non-left-recursive Boolean grammar $G = (\Sigma, N, P, S)$, for every solution $(\ldots, L_C, \ldots)$ of the associated system of language equations, and for every $A \in N$, if $\varepsilon \in L_A$, then $\varepsilon \in L_{G_+}(A)$.*

Proof. For every $A \in N$, such that $\varepsilon \in L_A$, there exists a rule

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n, \tag{17}$$

such that $\varepsilon \in \alpha_1 \ldots \alpha_m(\ldots, L_C, \ldots)$. Let $\alpha_1 \ldots \alpha_m = B_1 \ldots B_\ell$, where $\ell \geqslant 0$ and $B_t \in N$; we know that $\varepsilon \in L_{B_t}$ for all t.

Let us construct a tree (possibly an infinite one), in which every node is labelled by a nonterminal A, with $\varepsilon \in L_A$, and its descendants are the nonterminals $B_1, \ldots, B_\ell$ chosen by the above principle. The following properties of this tree can be established.

Claim I. If a subtree starting from a node labelled by A is finite, then $\varepsilon \in L_{G_+}(A)$.

Induction on the height of this subtree. If its height is 0, then the rule (17) is of the form $A \to \varepsilon \& \ldots \& \varepsilon \& \neg\beta_1 \& \ldots \& \neg\beta_n$, hence $A \to \varepsilon \& \ldots \& \varepsilon \in P_+$ and therefore $\varepsilon \in L_{G_+}(A)$.

If the rule for A is (17), with $\alpha_1 \ldots \alpha_m = B_1 \ldots B_\ell$, then, by construction, the descendants of this node are labelled $B_1, \ldots, B_\ell$ and each of them starts a finite subtree. By the induction hypothesis ℓ times, $\varepsilon \in L_{G_+}(B_t)$ for all t. Then the rule $A \to \alpha_1 \& \ldots \& \alpha_m \in P_+$ can be used to derive ε from A, which completes the proof of the claim.

Claim II. If the tree contains an infinite path, then G is not strongly non-left-recursive.

Consider the leftmost infinite path. For each node on this path, labelled A_i, the corresponding rule (17) contains a positive conjunct $A_i \to E_1 \ldots E_\ell B \eta_i$, where B is the next node on this path. Other paths that continue to $E_1, \ldots, E_\ell$ ($\ell \geqslant 0$) are to the left of the path under consideration, and hence they are finite by assumption. Therefore, by Claim I, $\varepsilon \in L_{G_+}(E_t)$ for all t, and $\varepsilon\langle A_i\rangle\varepsilon \rightsquigarrow \theta_i\langle A_{i+1}\rangle\eta_i$, where $\theta = E_1 \ldots E_\ell$ and $\varepsilon \in L_{G_+}(\theta_i)$.

Since the path is infinite, while there are finitely many nonterminals in the grammar, some nodes have identical labels. Let $A_i = A_j$ for $i < j$. Then $\varepsilon\langle A_i\rangle\varepsilon \rightsquigarrow \theta_i \ldots \theta_{j-1}\langle A_j\rangle\eta_{j-1} \ldots \eta_i$,

where $\varepsilon \in L_{G_+}(\theta_i \dots \theta_{j-1})$, which contradicts the definition of a strongly non-left-recursive grammar, and Claim II follows.

Let us construct this kind of tree starting from any $A_0 \in N$, such that $\varepsilon \in L_{A_0}$. It follows from Claim II that the tree is finite. Indeed, if it is infinite, then it contains an infinite path by König's lemma, and hence, by Claim II, we come to a contradiction with the assumption that the grammar is strongly non-left-recursive. Now the finiteness of the tree, by Claim I, implies that $\varepsilon \in L_{G_+}(A_0)$. □

The proof of the theorem can now be given.

Proof of Theorem 1. Let $G = (\Sigma, N, P, S)$ be a Boolean grammar and suppose that the associated system of language equations (3) has two or more solutions modulo some finite substring-closed language. Let M be the largest modulus for which there is a unique solution $(\dots, L_C, \dots)$, and let $(\dots, L'_C, \dots)$ and $(\dots, L''_C, \dots)$ be distinct solutions modulo $M \cup \{w\}$, for some $w \notin M$, such that all proper substrings of w are in M.

Consider any $A \in N$, such that $L'_A \neq L''_A$ (at least one such nonterminal exists, because these two solutions modulo $M \cup \{w\}$ are different). Since these languages coincide modulo M, the string w is in the symmetric difference of L'_A and L''_A. Suppose, without loss of generality, that $w \in L'_A$ and $w \notin L''_A$. Then there exists a rule

$$A \to \alpha_1 \& \dots \& \alpha_m \& \neg\beta_1 \& \dots \& \neg\beta_n,$$

such that $w \in \alpha_i(\dots, L'_C, \dots)$ for all i and $w \notin \beta_j(\dots, L'_C, \dots)$ for all j. On the other hand, since $w \notin L''_A$, we know that $w \notin \alpha_i(\dots, L'_C, \dots)$ for some i or $w \in \beta_j(\dots, L'_C, \dots)$ for some j. In either case we obtain that w is in the symmetric difference of $\gamma(\dots, L'_C, \dots)$ and $\gamma(\dots, L''_C, \dots)$, for some $A \to \pm\gamma \in \mathit{conjuncts}(P)$.

Let $\gamma = s_1 \dots s_\ell$, where $s_t \in \Sigma \cup N$, and let us again assume without loss of generality that $w \in s_1 \dots s_\ell(\dots, L'_C, \dots)$ and $w \notin s_1 \dots s_\ell(\dots, L''_C, \dots)$. Then there is a factorization $w = u_1 \dots u_\ell$, such that $u_t \in s_t(\dots, L'_C, \dots)$ for all t. Let us prove that there exists such t_0 $(1 \leqslant t_0 \leqslant \ell)$ that $u_{t_0} = w$.

Suppose the contrary, that each u_t is a proper substring of w. Then they are all in M, and subsequently $u_i \in s_t(\dots, L'_C, \dots)$ implies $u_i \in s_t(\dots, L''_C, \dots)$ for each s_t. This would mean $w \in \gamma(\dots, L''_C, \dots)$, which contradicts the assumption.

It has thus been proved that $u_{t_0} = w$ for some t_0. Then $u_t = \varepsilon$ for all $t \neq t_0$, and hence $\varepsilon \in s_t(\dots, L'_C, \dots)$. By Lemma 6 it follows that $\varepsilon \in G_+(s_t)$ for all $t \neq t_0$. On the other hand, $u_{t_0} = w$ implies $s_{t_0} \in N$ (otherwise $u_{t_0} = w \in \Sigma$ and $w \in s_{t_0}(\dots, L''_C, \dots)$, contradicting the assumption).

Altogether it has been proved that there exists a nonterminal $B = s_{t_0}$, such that $\varepsilon\langle A\rangle\varepsilon \rightsquigarrow \eta\langle B\rangle\theta$, where $\varepsilon \in L_{G_+}(\eta)$, $\varepsilon \in L_{G_+}(\theta)$, and w is in the symmetric difference of L'_B and L''_B. The latter fact allows one to apply the same argument for B, and so on, obtaining a sequence of nonterminals $A = A_0, A_1, \dots, A_n, \dots$, where

$$\varepsilon\langle A_n\rangle\varepsilon \rightsquigarrow \eta_n\langle A_{n+1}\rangle\theta_n, \tag{18}$$

$\varepsilon \in L_{G_+}(\eta_n)$ and $\varepsilon \in L_{G_+}(\theta_n)$ for every $n \geqslant 0$. Since there are finitely many nonterminals in the grammar, there exist m and m', with $m < m'$, such that $A_m = A_{m'}$. Combining (18) for

$n = m \ldots, m' - 1$ yields $\varepsilon\langle A_m\rangle\varepsilon \rightsquigarrow \eta_m \ldots \eta_{m'-1}\langle A_m\rangle\theta_{m'-1} \ldots \theta_m$. This is a left recursion in the grammar, see Definition 7, which contradicts the assumption of this theorem. □