An extension of recursive descent parsing for Boolean grammars

Alexander Okhotin
okhotin@cs.queensu.ca

Technical report 2004–475
School of Computing, Queen's University,
Kingston, Ontario, Canada K7L 3N6

March 2004

Abstract

The well-known recursive descent parsing method for context-free grammars is extended for their generalization, Boolean grammars, which include explicit set-theoretic operations in the formalism of rules. Conjunction is implemented by scanning a part of the input multiple times, while the mechanism of exception handling is used to implement negation. A subclass of LL(k) Boolean grammars, for which recursive descent parsing is possible, is identified, the construction of a parsing table and parser code is formally specified and illustrated by an example. Correctness of the method is proved.

Contents

1 Introduction

The recursive descent method for context-free grammars is undoubtedly the most intuitively clear parsing technique, and the most well-known as well, being included in most university curricula in computer science. Used since the early 1960s (the credit is attributed to Lucas [1961] – see Aho et al. [1986, p.82]), it is also one of the first parsing techniques to be ever used. Surprisingly, its appeal has perhaps even grown over time.

Indeed, since 1980s, there has been a tendency of preferring recursive descent over the theoretically more powerful LR. Although LR parser generator software, such as YACC developed by Johnson [1975], was available, the generated parsers, which basically simulate pushdown automata, are not always easy to integrate into a program in a high-level programming language. In many cases it turns out to be easier to write a recursive descent parser manually. Hence, many new syntax specification and parser generation tools based on recursive descent were developed, which, unlike YACC, kept pace with the advances in the field of programming languages: for instance, S/SL by Holt et al. [1982], LLgen by Grune and Jacobs [1988] and ANTLR by Parr and Quong [1995].

In its original form, recursive descent is applicable to a proper subfamily of deterministic context-free languages called the LL(k) languages. The first theoretical treatment of LL(k) grammars and the first formal construction of parsing tables for tabular and recursive descent LL parsers (including the familiar First and Follow sets) is due to Knuth [1971]. Further fundamental properties of LL(k) languages and grammars were developed by Kurki-Suonio [1969], Lewis and Stearns [1968], Rozenkrantz and Stearns [1970] and Wood [1969–70]. Practical construction and use of recursive descent parsers is explained in the well-known textbook on compiler construction by Aho, Sethi and Ullman [1986] among other common syntax analysis techniques. Another book by Davie and Morrison [1981] specifically focuses on recursive descent as a guiding principle in compiler construction.

A generalization of recursive descent method for *conjunctive grammars* is known (Okhotin [2002]). These grammars, introduced by the author [2001], are an extension of context-free grammars with an intersection operation added to the formalism of rules. The generalized recursive descent relies upon scanning substrings multiple times to check all conjuncts of a rule.

This paper further generalizes the method for the class of *Boolean grammars*, which are themselves a further generalization of conjunctive grammars (Okhotin [2003]) that allows all set-theoretic operations, including negation. The formal semantics of Boolean grammars is defined using language equations (rather than derivation) in a way resembling the classical characterization of context-free grammars obtained by Ginsburg and Rice [1962]. In order to handle negation, the new recursive descent has to utilize the mechanism of exception handling found in most programming languages since Ada.

An brief introduction into Boolean grammars is given in Section 2. Section 3 defines the parsing table and gives an algorithm for constructing it. A formal construction of recursive descent parser code out of a grammar and a parsing table is defined in Section 4. The algorithm is proved correct in Section 5.

2 Boolean grammars

Definition 1 (Okhotin [2003]). *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*

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

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; $A \to \alpha_i$ and $A \to \beta_j$ are called unsigned conjuncts. Denote the set of all unsigned conjuncts in the rules from P as $uconjuncts(P)$.

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

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.

The semantics of Boolean grammars is defined using language equations.

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

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

A system (2) can have no solutions or multiple pairwise incomparable solutions. In the former case it is clearly invalid, while if it has solutions, one of them has to be declared as "the right one". The problems with defining this solution have been studied by the author [2003], and following is one of the two methods that were developed:

Definition 3. *Let $X_i = \varphi_i(X_1, \ldots, X_n)$ ($1 \leqslant i \leqslant n$) be a system of language equations, where the expressions φ_i contain the operations of union, intersection, complement and concatenation, terminal symbols and variables.*

A vector of languages $L = (L_1, \ldots, L_n)$ is called a naturally reachable solution of the system if for every finite modulus M 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 $L^{(0)} = (L_1 \cap M, \dots, L_n \cap M)$ 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

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

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

If exists, the naturally reachable solution is unique, and can be used to define the semantics of Boolean grammars.

Definition 4. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar, let $X = \varphi(X)$ be the associated system of language equations, let this system have naturally reachable solution $L = (L_1, \dots, L_n)$.*

Then the language $L_G(\varphi)$ of a formula φ is defined as a substitution $\varphi(L)$, while the language of the grammar is $L(G) = L_G(S)$.

The following transform of a Boolean grammar shall be used in the following; given a Boolean grammar $G = (\Sigma, N, P, S)$, define

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

The grammar $positive(G) = (\Sigma, N, positive(P), S)$ is thus a conjunctive grammar.

Lemma 1. *For every Boolean grammar $G = (\Sigma, N, P, S)$, such that the associated system complies to the semantics of naturally reachable solution, it holds that $L_G(A) \subseteq L_{positive(G)}(A)$ for every $A \in N$.*

Proof. Let $L = (L_1, \dots, L_n)$ be the naturally reachable solution of the system $X = \varphi()$ associated with G, let $L' = (L'_1, \dots, L'_n)$ be the vector of languages defined by the nonterminals of $positive(G)$, which is the least solution of the system of language equations $X = \varphi'(X)$ associated with $positive(G)$. Note that

$$\varphi_j(L'') \subseteq \varphi'_j(L'') \quad \text{(for every vector of languages } L'') \tag{6}$$

by the construction of $positive(G)$.

It has to be proved that $L_j \subseteq L'_j$ for all j ($1 \leqslant j \leqslant n$). It suffices to prove that for every pair (M, u) as in Definition 3 and for every i-th term of the sequence (3), if u is in $L_j^{(i)}$, then $u \in L'_j$.

By the induction hypothesis,

$$L^{(i-1)} \preccurlyeq L' \tag{7}$$

Thus, if $u \in L_j^{(i-1)}$, then $u \in L'_j$. If $u \notin L_j^{(i-1)}$, then $u \in \varphi_j(L^{(i-1)})$. By (6), $\varphi_j(L^{(i-1)}) \subseteq \varphi'_j(L^{(i-1)})$, and hence $u \in \varphi'_j(L^{(i-1)})$.

Since φ'_j is monotone, (7) implies $\varphi'_j(L^{(i-1)}) \subseteq \varphi'_j(L')$. This means that $u \in \varphi'_j(L')$.

Finally, $\varphi'_j(L') = L'_j$, because L' is a solution of the system associated with $positive(G)$. Therefore, $u \in L'_j$. □

Context-free recursive descent parsing requires the grammar to be free of *left recursion*, which means that no nonterminal A can derive $A\alpha$ ($\alpha \in (\Sigma \cup N)^*$). The reason for that is that a parser can enter an infinite loop otherwise.

A generalization of recursive descent for a larger class of grammars still has to impose a similar restriction. Although the semantics of Boolean grammars is defined by language equations and not by derivation, a certain artificial derivation has to be introduced in order to formulate the generalized restriction.

Definition 5. *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar. Define the relation of in-depth context-free derivability, $\overset{\text{CF/d}}{\Longrightarrow}$, which a binary relation on the set $\{\alpha \cdot \beta \cdot \gamma \mid \alpha, \beta, \gamma \in \Sigma^*\}$, as the reflexive and transitive closure of the following set of derivation rules:*

$$\alpha \cdot \beta A \gamma \cdot \delta \overset{\text{CF/d}}{\Longrightarrow} \alpha\beta\eta \cdot \sigma \cdot \theta\gamma\delta \tag{8}$$

for every $A \to \eta\sigma\theta \in uconjuncts(P)$.

Definition 6. *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 $\gamma, \delta \in (\Sigma \cup N)^*$, such that $\varepsilon \cdot A \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \gamma \cdot A \cdot \delta$, it holds that $\varepsilon \notin L_{positive(G)}(\gamma)$.*

Note that the non-left-recursivity of the context-free grammar $G' = (\Sigma, N, uconjuncts(P), S)$ is a sufficient condition for strong non-left-recursivity of G.

The following Boolean grammar will serve as a running example in this paper.

Example 1. *Consider the language $L = \{a^m b^n c^n \mid m \neq n\}$; it is non-context-free, while its complement is an inherently ambiguous context-free language.*

Following is a strongly non-left-recursive Boolean grammar tha denotes $L \cdot d \cdot (a^*b^*c^* \setminus L)$:

$S \to KdM$
$K \to AD \& \neg EC$
$M \to ABC \& \neg K$
$A \to aA \mid \varepsilon$
$B \to bB \mid \varepsilon$
$C \to cC \mid \varepsilon$
$D \to bDc \mid \varepsilon$
$E \to aEb \mid \varepsilon$

Formally, $(L \cdot d \cdot (a^*b^*c^* \setminus L), L, (a^*b^*c^* \setminus L), a^*, b^*, c^*, \{b^i c^i \mid i \geqslant 0\}, \{a^i b^i \mid i \geqslant 0\})$ is the naturally reachable solution of the associated system of language equations.

3 The LL(k) table and its construction

Let $k \geqslant 1$. For a string w, define

$$First_k(w) = \begin{cases} w, & \text{if } |w| \leqslant k \\ \text{first } k \text{ symbols of } w, & \text{if } |w| > k \end{cases} \tag{9}$$

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

Definition 7 (Nondeterministic LL(k) table). *Let $G = (\Sigma, N, P, S)$ be a Boolean grammar compliant to the semantics of naturally reachable solution, 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 A and w, for which $\varepsilon \cdot S \cdot \varepsilon \overset{CF/d}{\Longrightarrow} \delta \cdot A \cdot \eta$, and $w \in L_G(\varphi\eta)$, it holds that $A \to \varphi \in T'_k(A, First_k(w))$.

Definition 7 might look rather loose, since it does not specify any necessary conditions for being in $T'_k(A, First_k(w))$. However, the least (with respect to inclusion) collection of sets satisfying this definition is uncomputable, as shown by the author [2002] for the simpler case of conjunctive grammars, and this makes us think in terms of *suitable tables* rather than *the optimal table*.

The only tables usable with the new recursive descent algorithm are *deterministic tables* of the following form:

Definition 8 (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 $T_k(A, u) = -$ (if $T'_k(A, u) = \varnothing$)*

Let us describe a simple method of computing LL(k) tables. First, compute the sets PFIRST_k and PFOLLOW_k similar to those used in the case of conjunctive grammars (Okhotin [2002]).

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

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

Proof of correctness. Note that the algorithm completely ignores negative conjuncts, effectively using $positive(G)$ instead of G. A stronger claim holds: if $u \in L_{positive(G)}(s)$, then $First_k(u) \in \text{PFIRST}_k(s)$, which has been proved by the author [2002] for the case of conjunctive grammars. $\square$

Definition 9. *We shall say that $u \in \Sigma^*$ follows $\sigma \in (\Sigma \cup N)^*$ if $\varepsilon \cdot S \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \delta \cdot \sigma \cdot \eta$ and $u \in L_G(\eta)$.*

The function $\textsc{Pfollow}_k : N \to \Sigma^{\leqslant k}$ is defined by the following algorithm:

Algorithm 2. *For given G compliant to the semantics of naturally reachable solution and $k > 0$, 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 each $B \to \beta \in uconjuncts(P)$
        for each factorization $\beta = \mu A \nu$, where $\mu, \nu \in V^*$ and $A \in N$
            $\textsc{Pfollow}_k(A) = \textsc{Pfollow}_k(A) \cup First_k(\textsc{Pfirst}_k(\nu) \cdot \textsc{Pfollow}_k(B))$;
```

Proof of correctness. Let u follow A. Then $\varepsilon \cdot S \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \delta \cdot \sigma \cdot \eta$ and $u \in L_G(\eta)$.

The proof is an induction on the length of derivation of $\delta \cdot \sigma \cdot \eta$.

Basis: If the triple is $\varepsilon \cdot S \cdot \varepsilon$, then it has to be proved that $First_k(\varepsilon) = \varepsilon$ is in $\textsc{Pfollow}_k(S)$. It is added there by the first statement of the algorithm.

Induction step. Let $\varepsilon \cdot S \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \alpha \cdot B \cdot \beta \overset{\text{CF/d}}{\Longrightarrow} \alpha\mu \cdot A \cdot \nu\beta$ and $B \to \mu A \nu \in uconjuncts(P)$. and let $u \in L_G(\nu\beta)$.

Then there exists a factorization $u = xy$, such that $x \in L_G(\nu)$ and $y \in L_G(\beta)$. According to Algorithm 1, $First_k(x) \in \textsc{Pfirst}_k(\nu)$; by the induction hypothesis, $First_k(y)$ is added to $\textsc{Pfollow}_k(B)$ at some point of the computation of the algorithm. Then, at this point,

$$First_k(u) = First_k(First_k(x) \cdot First_k(y)) \in First_k(\textsc{Pfirst}_k(\nu) \cdot \textsc{Pfollow}_k(B)),$$

and hence $First_k(u)$ is added to $\textsc{Pfollow}_k(A)$ next time the unsigned conjunct $B \to \mu A \nu$ and the factorization $\mu A \nu = \mu \cdot A \cdot \nu$ are considered. $\square$

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

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

```
for each rule $A \to \varphi \in P$
    for each $x \in First_k(\textsc{Pfirst}_k(\varphi) \cdot \textsc{Pfollow}_k(A))$
        add the rule to $T'_k(A, x)$;
```

Proof of correctness. Consider a rule

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \quad (m \geqslant 1, n \geqslant 0) \tag{10}$$

that *should* be in $T'_k(A, x)$ in accordance to Definition 7. Then $\varepsilon \cdot S \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \delta \cdot A \cdot \eta$ and $w \in L_G(\varphi\eta)$, where $\varphi = \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n$ and $x = First_k(w)$.

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

$$First_k(u) \in \text{PFIRST}_k(\varphi) \tag{11}$$

Since v follows A,

$$First_k(v) \in \text{PFOLLOW}_k(A) \tag{12}$$

by the construction of $\text{PFOLLOW}_k(A)$.

Concatenating (11) and (12) yields

$$x = First_k(uv) = First_k(First_k(u) \cdot First_k(v)) \in First_k(\text{PIRST}_k(\varphi) \cdot \text{PFOLLOW}_k(A)),$$

which means that the rule (10) will be added to $T'_k(A, x)$ in the iteration $((10), x)$. $\square$

	PFIRST_1	PFOLLOW_1
S	$\{a, b, d\}$	$\{\varepsilon\}$
K	$\{\varepsilon, a, b\}$	$\{\varepsilon, d\}$
M	$\{\varepsilon, a, b, c\}$	$\{\varepsilon\}$
A	$\{\varepsilon, a\}$	$\{\varepsilon, b, c, d\}$
B	$\{\varepsilon, b\}$	$\{\varepsilon, c\}$
C	$\{\varepsilon, c\}$	$\{\varepsilon, d\}$
D	$\{\varepsilon, b\}$	$\{\varepsilon, c, d\}$
E	$\{\varepsilon, a\}$	$\{\varepsilon, b, c, d\}$

Table 1: PFIRST_1 and PFOLLOW_1.

Example 2. *Consider the grammar from Example 1. The sets* PFIRST_1 *and* PFOLLOW_1 *for this grammar are given in Table 1. A deterministic LL(1) table constructed using these sets is shown in Table 2.*

Note that the strings $d \in \text{PFIRST}_1(S)$ *and* $\varepsilon \in \text{PFIRST}_1(K)$ *are fictional, as no actual strings from* $L_G(S)$ *and* $L_G(K)$ *can start from these. Consequently,* $T(S, d) = S \to KdM$*, which 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.*

	ε	a	b	c	d
S		$S \to KdM$	$S \to KdM$	$-$	$S \to KdM$
K	$K \to AD\&\neg EC$	$K \to AD\&\neg EC$	$K \to AD\&\neg EC$	$-$	$K \to AD\&\neg EC$
M	$M \to ABC\&\neg K$	$M \to ABC\&\neg K$	$M \to ABC\&\neg K$	$M \to ABC\&\neg K$	$-$
A	$A \to \varepsilon$	$A \to aA$	$A \to \varepsilon$	$A \to \varepsilon$	$A \to \varepsilon$
B	$B \to \varepsilon$	$-$	$B \to bB$	$B \to \varepsilon$	$-$
C	$C \to \varepsilon$	$-$	$-$	$C \to cC$	$C \to \varepsilon$
D	$D \to \varepsilon$	$-$	$D \to bDc$	$D \to \varepsilon$	$D \to \varepsilon$
E	$E \to \varepsilon$	$E \to aEb$	$E \to \varepsilon$	$E \to \varepsilon$	$E \to \varepsilon$

Table 2: LL(1) table.

4 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.

There will be a procedure for each terminal and nonterminal symbol in the grammar, and two static variables accessible to all procedures: the input string w and a positive integer i pointing at a position in this string. Each procedure $s()$ (corresponding to a symbol $s \in \Sigma \cup N$) starts with some initial value of this pointer, $i = i'$, and, after doing some computation and making some recursive calls,

- either returns, setting the pointer to $i = i''$ (where $i' \leqslant i'' \leqslant |w|$), thus reporting a successul parse of $w_{i'} \ldots w_{i''-1}$ from s,
- or raises an exception, which means that a suitable i'', such that the symbol s could generate $w_{i'} \ldots w_{i''-1}$, was not found; in this case i points to 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 $w_i = a$, then
        $i = i + 1$;
    else
        raise exception;
}
```

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

```
$A()$
{
    switch($T(A, First_k(w_i w_{i+1} \ldots))$)
    {
        case $A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n$:
            *(code for conjunct $A \to \alpha_1$)*
            *(code for conjunct $A \to \alpha_2$)*
            ⋮
            *(code for conjunct $A \to \alpha_m$)*
            *(code for conjunct $A \to \neg\beta_1$)*
            ⋮
            *(code for conjunct $A \to \neg\beta_n$)*
        case $A \to \ldots$
          ⋮
        default:
            raise exception;
        }
    }
```

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

```
    let $start = i$;        /* omit if this first conjunct is the only one ($m = 1$, $n = 0$) */
    $s_1()$;
    ⋮
    $s_\ell()$;
    let $end = i$;                          /* omit if this first conjunct is the only one */
```

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

```
    $i = start$;
    $s_1()$;
    ⋮
    $s_\ell()$;
    if $i \neq end$, then raise exception;
```

and the code for every negative conjunct $A \to \neg t_1 \ldots t_\ell$ is

```
    boolean $failed = false$;
    try
    {
        $i = start$;
        $s_1()$;
        ⋮
        $s_\ell()$;
        if $i \neq end$, then raise exception;
    }
```

```
exception handler:
    $failed = true$;
if $\neg failed$ raise exception;
$i = end$;                                   /* if this is the last conjunct in the rule */
```

The main procedure is:

```
try
{
    int $i = 1$;
    $S()$;
    if $i \neq n + 1$, then raise exception;
}
exception handler:
    Reject;
Accept;
```

Example 3. *Let us construct a recursive descent parser for the Boolean grammar from Example 1, using the LL(1) table constructed in Example 2.*

Following is the C++ program written using the method given above. The deviations from the model parser are minor. A C pointer `char *p` *to a position in the string is used instead of an integer i. Switch statements directly use the lookahead symbol instead of looking up a physical LL(1) table; T is thus hardcoded into the program. A structure* `parse_error` *is used for the objects generated when exceptions are raised; this structure could have been left empty, but in this implementation it records the name of the procedure where the exception was raised, which is a straightforward method of error diagnostics.*

```
#include <iostream>
#include <string.h>

struct parse_error
{
  char *s;
  parse_error(char *s1="") : s(s1) {}
};

char *p;

void S();
void K();
void M();
void A();
void B();
void C();
void D();
void E();

void a()
{
  if(*p=='a')
    p++;
  else
    throw parse_error("a");
}
void b()
{
  if(*p=='b')
    p++;
  else
    throw parse_error("b");
}
void c()
{
  if(*p=='c')
    p++;
  else
    throw parse_error("c");
}
void d()
```

```
{
  if(*p=='d')
    p++;
  else
    throw parse_error("d");
}
void S()
{
  switch(*p)
  {
  case 'a':
  case 'b':
  case 'd':
    K();
    d();
    M();
    break;
  default:
    throw parse_error("S");
  }
}
void K()
{
  switch(*p)
  {
  case 0:
  case 'a':
  case 'b':
  case 'd': {
    char *start=p;
    A();
    D();
    char *end=p;

    bool failed=false;
    try
    {
      p=start;
      E();
      C();
      if(p!=end) throw parse_error();
    }
    catch(parse_error) { failed=true; }
    if(!failed) throw parse_error("K:~EC");

    p=end;
    }
    break;
  default:
    throw parse_error("K");
  }
}
void M()
{
  switch(*p)
  {
  case 0:
  case 'a':
  case 'b':
  case 'c': {
    char *start=p;
    A();
    B();
    C();
    char *end=p;

    bool failed=false;
    try
    {
      p=start;
      K();
      if(p!=end) throw parse_error();
    }
    catch(parse_error) { failed=true; }
    if(!failed) throw parse_error("M:~K");

    p=end;
    }
    break;
  default:
    throw parse_error("M");
  }
}
void A()
{
  if(*p=='a')
  {
    a();
    A();
  }
}
void B()
{
  if(*p=='b')
  {
    b();
    B();
  }
}
void C()
{
  if(*p=='c')
  {
    c();
    C();
  }
}
void D()
{
  switch(*p)
  {
  case 'b':
    b();
    D();
    c();
    break;
  case 0:
  case 'c':
  case 'd':
    break;
  default:
    throw parse_error("D");
  }
}
```

```
void E()
{
  switch(*p)
  {
  case ’a’:
    a();
    E();
    b();
    break;
  case 0:
  case ’b’:
  case ’c’:
  case ’d’:
    break;
  default:
    throw parse_error("E");
  }
}

int main()
{
  char *w="aabcdabbc";
  try
  {
    p=w;
    S();
    if(p!=w+strlen(w))
      throw parse_error("$");
  }
  catch(parse_error &err)
  {
    std::cout << "Error in position " << (p-w)
              << " (" << err.s << ").\n";
    return false;
  }

  std::cout << "Accept.\n";
  return true;
}
```

This program invokes the parser on the string $w = aabcdabbc$, which is in the language, and the computation leads to acceptance. If verbose prints are added to the program, the following computation history is revealed:

```
S() on .aabcdabbc: S->KdM
  K() on .aabcdabbc: K->AD&˜EC
    A() on .aabcdabbc: A->aA
      a() on .aabcdabbc: Ok.
      A() on a.abcdabbc: A->aA
        a() on a.abcdabbc: Ok.
        A() on aa.bcdabbc: A->e
          Done (aa.bcdabbc).
        Done (aa.bcdabbc).
      Done (aa.bcdabbc).
    D() on aa.bcdabbc: D->bDc
      b() on aa.bcdabbc: Ok.
      D() on aab.cdabbc: D->e
        Done (aab.cdabbc).
      c() on aab.cdabbc: Ok.
      Done (aabc.dabbc).
    Done (aabc.dabbc).
    E() on .aabcdabbc: E->aEb
      a() on .aabcdabbc: Ok.
      E() on a.abcdabbc: E->aEb
        a() on a.abcdabbc: Ok.
        E() on aa.bcdabbc: E->e
          Done (aa.bcdabbc).
        b() on aa.bcdabbc: Ok.
        Done (aab.cdabbc).
      b() on aab.cdabbc: Error.
    Caught an exception.
  d() on aabc.dabbc: Ok.
  M() on aabcd.abbc: M->ABC&˜K
    A() on aabcd.abbc: A->aA
      a() on aabcd.abbc: Ok.
      A() on aabcda.bbc: A->e
        Done (aabcda.bbc).
      Done (aabcda.bbc).
    B() on aabcda.bbc: B->bB
      b() on aabcda.bbc: Ok.
      B() on aabcdab.bc: B->bB
        b() on aabcdab.bc: Ok.
        B() on aabcdabb.c: B->e
          Done (aabcdabb.c).
        Done (aabcdabb.c).
      Done (aabcdabb.c).
    C() on aabcdabb.c: C->cC
      c() on aabcdabb.c: Ok.
      C() on aabcdabbc.: C->e
        Done (aabcdabbc.).
      Done (aabcdabbc.).
    Done (aabcdabbc.).
    K() on aabcd.abbc: K->AD&˜EC
      A() on aabcd.abbc: A->aA
        a() on aabcd.abbc: Ok.
        A() on aabcda.bbc: A->e
          Done (aabcda.bbc).
        Done (aabcda.bbc).
      D() on aabcda.bbc: D->bDc
        b() on aabcda.bbc: Ok.
        D() on aabcdab.bc: D->bDc
          b() on aabcdab.bc: Ok.
          D() on aabcdabb.c: D->e
            Done (aabcdabb.c).
          c() on aabcdabb.c: Ok.
          Done (aabcdabbc.).
        c() on aabcdabbc.: Error.
    Caught an exception.
  Done (aabcdabbc.).
Accept.
```

If the string $w = aabcdabbc \in L(G)$ used in Example 3 is replaced with the string

$w = aabcdaabc \notin L(G)$, then the program terminates, issuing the error message

```
Error in position 9 (M:~K).
```

which gives out the position in the string where the syntax error was encountered (the end of the string) and the reason of error: the negative conjunct $M \to \neg K$ failed, because something was derived from K.

5 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 it is in the language.

The termination of the algorithm can be proved under quite weak assumptions on the grammar: strong non-left-recursivity alone is sufficient, and no claims on the mechanism of choosing the rules T are made, while the grammar itself is not even required to comply to Definition 3. This allows us to abstract from the semantics of the grammar and the goal of parsing, concentrating on the general structure of the computation.

Lemma 2. *Let $G = (\Sigma, N, P, S)$ be an arbitrary Boolean grammar, and consider the conjunctive grammar $G' = positive(G) = (\Sigma, N, positive(P), S)$. Let $k \geqslant 1$; let $T : N \times \Sigma^{\leqslant k} \to P$ be an arbitrary function, 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 A derives u in G'.*

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 $\varepsilon \cdot A \cdot \varepsilon \overset{CF/d}{\Longrightarrow} \gamma \cdot B \cdot \delta$, where γ derives u in G'.*

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 $a \overset{G'}{\Longrightarrow}^* a$.

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 $A \overset{G'}{\Longrightarrow}^* \varepsilon$.

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

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \quad (m \geqslant 1,\ n \geqslant 0,\ \alpha_i, \beta_i \in (\Sigma \cup N)^*), \tag{13}$$

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

Consider every positive conjunct $A \to \alpha_i$, and let $\alpha_i = s_1 \ldots s_\ell$. Then the code $s_1(); \ldots; s_\ell()$ is executed on uv, and it returns, consuming u. Consider a factorization $u = u_1 \ldots u_\ell$ 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 $u_i u_{i+1} \ldots u_\ell v$, consuming u_i, and the height of recursive calls made by $s_i()$ does not exceed $h-1$. By the induction hypothesis, $s_i \overset{G'}{\Longrightarrow}{}^* u_i$. These derivations according to G' can be combined into the following derivation:

$$\alpha_i = s_1 \ldots s_\ell \overset{G'}{\Longrightarrow} \ldots \overset{G'}{\Longrightarrow} u_1 \ldots u_\ell = u \tag{14}$$

Now use (14) for all i to produce the following derivation in the conjunctive grammar G':

$$A \overset{G'}{\Longrightarrow} (\alpha_1 \& \ldots \& \alpha_m) \overset{G'}{\Longrightarrow} \ldots \overset{G'}{\Longrightarrow} (u \& \ldots \& u) \overset{G'}{\Longrightarrow} u, \tag{15}$$

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 $u = \varepsilon$, $\varepsilon \cdot A \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \varepsilon \cdot A \cdot \varepsilon$ and $u = \varepsilon \in L_{G'}(\varepsilon)$.

Induction step. $A()$, called on uv, begins with determining a rule (13) using $T_k(A, First_k(uv))$ and then proceeds with calling the subroutines corresponding to the symbols in the right hand side of (13). 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 $A \to \gamma C \delta$ ($\gamma C \delta \in \{\alpha_i, \beta_j\}$) 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 $\gamma = s_1 \ldots s_\ell$. The call to $C()$ is preceded by the calls to $s_1(); \ldots; s_\ell()$, where each $s_t()$ is called on $x_t \ldots x_\ell yv$ and returns, consuming x_t ($x = x_1 \ldots x_\ell$). By part I of this lemma, this implies $s_t \overset{G'}{\Longrightarrow}{}^* x_t$, and hence $\gamma \overset{G'}{\Longrightarrow}{}^* x$.

By the existence of the unsigned conjunct $A \to \gamma C \delta$, $\varepsilon \cdot A \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \gamma \cdot C \cdot \delta$. 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 to apply the induction hypothesis to conclude that $\varepsilon \cdot C \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \mu \cdot B \cdot \eta$, such that $\mu \overset{G'}{\Longrightarrow}^* y$.

Combining these two derivations according to Definition 6, we get $\varepsilon \cdot A \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \gamma\mu \cdot B \cdot \eta\delta$, while the two conjunctive derivations in G' can be merged to obtain $\gamma\mu \overset{G'}{\Longrightarrow}^* u$. $\square$

Lemma 3. *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar. Let $k \geqslant 1$; let $T : N \times \Sigma^{\leqslant k} \to P$ be an arbitrary function, let the set of procedures be constructed with respect to G and T.*

Then, for any $s \in \Sigma \cup N$ 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.

By König's lemma, this tree should contain an infinite path

$$(A_1, u_1), (A_2, u_2), \ldots, (A_p, u_p), \ldots \tag{16}$$

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

Since w is of finite length, it has finitely many different suffices, and the decreasing second component in (16) 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 (16). Consider any two instances of (A, u); then, by Lemma 2, $\varepsilon \cdot A \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \gamma \cdot A \cdot \delta$, such that $\gamma \overset{G'}{\Longrightarrow}^* \varepsilon$. This contradicts the assumption that G is strongly non-left-recursive. $\square$

Lemma 4. *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar that complies to the semantics of naturally reachable solution. Let $k \geqslant 1$. Let $T : N \times \Sigma^{\leqslant k} \to P$ be a deterministic LL(k) table for G, let the set of procedures be constructed with respect to G and T.*

For any $y, z \in \Sigma^*$ and $s_1 \ldots s_\ell \in \Sigma \cup N$ ($l \geqslant 0$), such that z follows $s_1 \ldots s_\ell$, 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. Generally, although the code $s_1(); \ldots; s_\ell()$ does not necessarily *return*, it always terminates in this or that way (by returning or by raising an exception) according to Lemma 3. Consider the tree of recursive calls of the code $s_1(); \ldots; s_\ell()$ executed on the input yz, as in Lemma 3. This tree is finite; let h be its height.

The proof is an induction on the pair (h, ℓ). The natural basis case would be $h = 0, \ell = 1$; to improve 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: $(0, 1)$ **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: $(h-1, \ldots) \to (h, 1)$**, or** $(0, 1)$ **and** $s \in N$**.** Let $\ell = 1$ and $s_1 = A \in N$, let z follow A, and let h be the height of the tree of recursive calls made by $A()$ executed on yz.

$\textcircled{\Rightarrow}$ If $A()$ returns on yz, consuming y, then $T(A, First_k(yz))$ gives some rule

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \quad (m \geqslant 1,\ n \geqslant 0), \tag{17}$$

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 $h-1$. z follows $s_1 \ldots s_\ell$ just because z follows A and $A \to s_1 \ldots s_\ell \in uconjuncts(P)$. Hence, by the induction hypothesis, $y \in L_G(\alpha_i)$.
2. For every negative conjunct $A \to \neg\beta_j$ ($\beta_j = t_1 \ldots t_\ell$), the code $t_1(); \ldots; t_\ell()$ 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, z follows $t_1 \ldots t_\ell$. This allows 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 $y \in L_G(A)$.

$\textcircled{\Leftarrow}$ If $y \in L_G(A)$, then there exists a rule (17), such that

$$y \in L_G(\alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n) \tag{18}$$

Since z follows A, $\varepsilon \cdot S \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \delta \cdot A \cdot \eta$, where $z \in L_G(\eta)$. Combining (18) with this, we obtain that $yz \in L_G(\varphi\eta)$, where $\varphi = \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n$. Then, by the definition of a deterministic LL(k) table, $T(A, First_k(yz))$ equals (17).

Hence the computation of $A()$ on yz starts from choosing the alternative (17). Consider all the conjuncts of the rule (17) in the order of the corresponding code fragments, and let us prove that each of these fragments is successively passed:

1. For every positive conjunct $A \to \alpha_i$ ($\alpha_i = s_1 \ldots s_\ell$), $y \in L_G(s_1 \ldots s_\ell)$ by (18) and z follows $s_1 \ldots s_\ell$ (since z follows A and $A \to s_1 \ldots s_\ell \in uconjuncts(P)$). Therefore, by the induction hypothesis, the code $s_1(); \ldots; s_\ell()$ returns on yz, consuming z.

2. For every negative conjunct $A \to \neg\beta_j$ ($\beta_j = t_1 \dots t_\ell$), since $y \notin L_G(t_1 \dots t_\ell)$ and z follows $t_1 \dots t_\ell$, by the induction hypothesis, it is not the case that the code $t_1(); \dots; t_\ell()$ returns on yz, consuming z. On the other hand, by Lemma 3, the code $t_1(); \dots; t_\ell()$ terminates on yz, either by returning or by raising an exception. This implies that $t_1(); \dots; t_\ell()$, 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 (17) 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: $(h, \ell - 1) \to (h, \ell)$**.** Let $\ell \geqslant 2$, let $s_1 \dots s_\ell \in (\Sigma \cup N)^*$ and let $First_k(z)$ follow $s_1 \dots s_\ell$.

$\textcircled{\Rightarrow}$ Let the code $s_1(); \dots; s_{\ell-1}(); s_\ell()$ 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 $s_1(); \dots; s_{\ell-1}()$ 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 z follows s_ℓ, the induction hypothesis is directly applicable to the computation of $s_\ell()$ on vz, yielding

$$v \in L_G(s_\ell) \tag{19}$$

In order to use the induction hypothesis for the former $\ell - 1$ calls, first it has to be established that vz follows $s_1 \dots s_\ell$. We know that z follows $s_1 \dots s_\ell$, which means that $\varepsilon \cdot S \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \delta \cdot s_1 \dots s_\ell \cdot \eta$, where $z \in L_G(\eta)$. Hence, $\varepsilon \cdot S \cdot \varepsilon \overset{\text{CF/d}}{\Longrightarrow} \delta \cdot s_1 \dots s_{\ell-1} \cdot s_\ell \eta$, while concatenating (19) with $z \in L_G(\eta)$ yields $vz \in L_G(s_\ell \eta)$.

Now the induction hypothesis can be used for the computation of $s_1(); \dots; s_\ell()$ on uvz (which returns, consuming u), giving

$$u \in L_G(s_1 \dots s_{\ell-1}) \tag{20}$$

By (20) and (19), $y = uv \in L_G(s_1 \dots s_{\ell-1} s_\ell)$.

$\textcircled{\Leftarrow}$ 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 z follows $s_1 \ldots s_{\ell-1}s_\ell$, vz follows $s_1 \ldots s_{\ell-1}$ (as proved in the previous part of the proof). This allows to apply the induction hypothesis to this subcomputation and obtain that $s_1(); \ldots; s_{\ell-1}()$ returns on uvz, consuming u.

Once the subcomputation $s_1(); \ldots; s_{\ell-1}()$ returns on uvz, the computation of $s_1(); \ldots; s_{\ell-1}(); s_\ell()$ proceeds with invoking $s_\ell()$ on vz. Hence, $s_\ell()$ on z is also a subcomputation, which has height no greater than that of the whole computation. Since 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, $s_1(); \ldots; s_{\ell-1}(); s_\ell()$, returns on yz, consuming $uv = y$. □

Now let $y = w$, $z = \varepsilon$, $\ell = 1$, $s_1 = S$, $\eta = \varepsilon$. Then the conditions of Lemma 4 are satisfied, and the following result is obtained:

Corollary 1. *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar that complies to the semantics of naturally reachable solution. Let $k \geqslant 1$. Let $T : N \times \Sigma^{\leqslant k} \to P$ 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 1. *Let $G = (\Sigma, N, P, S)$ be a strongly non-left-recursive Boolean grammar that complies to the semantics of naturally reachable solution. Let $k \geqslant 1$. Let $T : N \times \Sigma^{\leqslant k} \to P$ 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, accepts if $w \in L(G)$, rejects otherwise.

6 Conclusion

A generalization of the familiar context-free recursive descent method has been developed, which becomes the first truly practical parsing algorithm for Boolean grammars.

The notions behind the algorithm and the proof of its correctness are admittedly not very easy; however, the algorithm itself is intuitively clear and can easily be used in applications instead of the standard context-free recursive descent. This makes it possible to integrate support for Boolean grammars into practical parser generators that currently use context-free recursive descent.

References

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

[1981] A. J. T. Davie, R. Morrison, *Recursive descent compiling*, Chichester, Ellis-Horwood, 1981.

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

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

[1982] R. C. Holt, J. R. Cordy, D. B. Wortman, "An introduction to S/SL: Syntax/Semantic Language", *ACM TOPLAS*, 4:2 (1982), 149–178.

[2003] Q. T. Jackson, "Efficient formalism-only parsing of XML/HTML using the §-calculus", *ACM SIGPLAN Notices*, 38:2 (2003), 29–35.

[1975] S. Johnson, "Yacc: yet another compiler-compiler", *Computing Science Technical Report 32*, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[1967] D. E. Knuth, "On the translation of languages from left to right", *Information and Control*, 11 (1967), 269–289.

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

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

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

[1961] P. Lucas, "Die Strukturanalyse von Formelbersetzern", *Elektronische Rechenanlagen*, 3:4 (1961), 159–167.

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

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

[2003] A. Okhotin, "Boolean grammars", *Developments in Language Theory* (Proceedings of DLT 2003, Szeged, Hungary, July 7–11, 2003), LNCS 2710, 398–410; journal version submitted.

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

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

[1969–70] D. Wood, “The theory of left factored languages” (I, II), *Computer Journal*, 12:4 (1969), 349–356; 13:1 (1970) 55–62.