Journal of Universal Computer Science, vol. 21, no. 7 (2015), 891-911
submitted: 24/2/14, accepted: 19/5/15, appeared: 1/7/15 © J.UCS

Naive Infinite Enumeration of Context-free Languages in Incremental Polynomial Time

Christophe Costa Florêncio
(KU Leuven, Belgium
chris.costaflorencio@cs.kuleuven.be)

Jonny Daenen
(Hasselt University and Transnational University of Limburg, Belgium
jonny.daenen@uhasselt.be)

Jan Ramon
(KU Leuven, Belgium
jan.ramon@cs.kuleuven.be)

Jan Van den Bussche
(Hasselt University and Transnational University of Limburg, Belgium
jan.vandenbussche@uhasselt.be)

Dries Van Dyck
(Belgian Nuclear Research Centre (SCK-CEN)
Boeretang 200, BE-2400 Mol, Belgium
vandyck.dries@gmail.com)

Abstract: We consider the naive bottom-up concatenation scheme for a context-free language and show that this scheme has the incremental polynomial time property. This means that all members of the language can be enumerated without duplicates so that the time between two consecutive outputs is bounded by a polynomial in the number of strings already generated.

Key Words: context-free grammar, systematic generation, incremental polynomial time, polynomial delay

Category: F.4.2, F.2

1 Introduction

Let G be a context-free grammar that is arbitrary but fixed, i.e., G is not considered as part of the input. Hence, we may suppose G is in a convenient normal form, in particular Chomsky Normal Form. We can define two basic enumeration problems concerning the language $L(G)$ generated by G:

Given-length enumeration with polynomial delay: Given a natural number n, output all strings of length n belonging to $L(G)$, without duplicates, with *polynomial delay.* By "polynomial delay" we mean that the first output, and every next output, is produced within $p(n)$ time, for some fixed polynomial p. Technically, the output is ended by an "end of output" (EOO) message, and the time spent between the last output string and EOO should also be bounded by $p(n)$. Moreover, if there are no strings of length n in $L(G)$, then the algorithm should output an EOO right away, again in time bounded by $p(n)$.

Infinite enumeration in incremental polynomial time: Output *all* of the strings in $L(G)$, without duplicates, in incremental polynomial time (IPT), meaning that the time spent between the mth and the $(m + 1)$th output is bounded by $p(m)$ for some fixed polynomial p. Here, m is not directly related to string length, but is simply a count of the number of strings that have been output so far. Since all but the most trivial context-free languages are infinite, we refer to this problem as *infinite enumeration.* In principle, an algorithm for infinite enumeration runs forever, but the incremental polynomial-time bound guarantees that the time for every next output grows only polynomially.

The notions of polynomial delay and incremental polynomial time were originally introduced (in a setting unrelated to context-free languages) by [Johnson et al. 1988]. The set of strings of some length n belonging to a language is also known as a "cross-section" of that language [see Ackerman and Mäkinen 2009].

Basic as the above two problems are, the literature on them is relatively scarce. Given-length enumeration was first discussed by [Mäkinen 1997], but not solved completely; then [Dömösi 2000] presented a polynomial-delay solution to the same problem by a modification of the well-known CYK parsing algorithm. Notably, for the special case of regular languages, very efficient algorithms are available [see Ackerman and Mäkinen 2009]. Their solution has the additional benefit of enumerating the strings in lexicographic order. Later, [Dong 2009] reported linear-time improvements to Dömösi's algorithm. A related problem which has received quite some attention in the literature is the efficient generation of a true random sample of a context-free language [see Gore et al. 1997, Flajolet et al. 1994, Arnold and Sleep 1980].

So, efficient algorithms for given-length enumeration are already available. In the present paper, we consider infinite enumeration. We will show, perhaps unsurprisingly, that any algorithm for given-length enumeration with polynomial delay can be adapted to do infinite enumeration in incremental polynomial time.

The main topic of this paper, however, is the naive, bottom-up concatenation scheme that enumerates strings not by length, but by depth of their parse tree.

While this scheme is not as efficient as the above algorithms, it is still important because it is so basic and natural. Indeed it is a natural question to ask: does the naive bottom-up concatenation scheme already have the IPT property? In this paper we answer this question affirmatively. We believe this result is interesting mainly from a theoretical perspective as it adds to our fundamental understanding of enumerating context-free languages. The proof of our main result is elementary and is based on detailed pumping-lemma-like arguments. An important property is that the gap in lengths between two consecutive strings in a context-free language is bounded by a constant (which depends on the grammar). Our proofs bound important parameters that govern the amount of work done in one iteration of the concatenation scheme in terms of the number of unique strings generated up to the previous iteration. In particular, ambiguous grammars do not pose a problem.

Infinite enumeration may have practical applications in software testing [see Somerville 1998], where a language of test-inputs is described by a context-free grammar [see Arnold and Sleep 1980, Duncan and Hutchinson 1981, Maurer 1990]. In this situation, exhaustive testing of the software on all inputs of the language (e.g., up to a certain length, or until the time budget for testing is exhausted) can be driven by infinite enumeration of a tailor-made context-free language.

Conversely, infinite enumeration may also have applications in verification of context-free languages. While this task is decidable for some properties [see Baeten et al. 1993], it is undecidable for many other properties, e.g., containment of one context-free language in another is undecidable [see Hopcroft and Ullman 1979]. In such cases, infinite enumeration may be useful to detect counterexamples to conjectured properties, or, when no counterexample is found after a sufficiently long time, it may provide confidence in the conjecture, after which the verifier may start an attempt to find a proof by other methods.

Also, there has been interest in tools for testing and debugging the grammars themselves [see Lämmel 2001, Purdom 1972, Xu et al. 2011], where again infinite enumeration may be helpful.

The paper is outlined as follows: in [Section 2] we first give the necessary definitions and in [Section 3] we give a formal specification of the naive algorithm. In [Section 4], four important results are obtained, which are used in [Section 5] to show the IPT property of the naive algorithm. In [Section 6] a general method is given for transforming a given-length enumeration algorithm with polynomial delay to an algorithm for infinite enumeration in incremental polynomial time. We conclude in [Section 7].

2 Preliminaries

A *context-free grammar* G is a tuple $(\mathcal{N}, \Sigma, \mathcal{P}, S)$, where

– $\mathcal{N}$ is a finite set of *non-terminals*;

– Σ is a finite set of *terminals*, disjoint from $\mathcal{N}$;

– $\mathcal{P}$ is a set of *productions* of the form $X \to \alpha$ with $X \in \mathcal{N}$ and $\alpha \in (\Sigma \cup \mathcal{N})^*$;

– $S \in \mathcal{N}$ is the *start symbol*.

For the rest of the paper, we assume that all grammars are in *Chomsky Normal Form* (CNF) [see Hopcroft and Ullman 1979] without ε-productions, i.e., all productions are of the following form:

– $A \to BC$, a *non-terminal production* or

– $A \to a$, with $a \in \Sigma$, a *terminal production*

where A, B and C are non-terminals. As we mentioned, the empty string ε cannot be used. Importantly, this implies that we will only deal with nonempty strings.

We say a non-terminal A *derives* a string s, written as $A \Rightarrow^* s$, if one of the following holds:

– $s \in \Sigma$ and $A \to s \in \mathcal{P}$ (one-step derivation); or

– $\exists B, C \in \mathcal{N} : \exists u, v \in \Sigma^* : A \to BC \in \mathcal{P} \wedge B \Rightarrow^* u \wedge C \Rightarrow^* v \wedge s = uv$.

The *language of a non-terminal* A is defined by $L(G_A) = \{s \mid A \Rightarrow^* s\}$. The language of the start symbol S is also called the *language of* G and is defined by $L(G) = L(G_S)$.

We will also use the *dependency graph* of a context-free grammar (in CNF). This is a directed graph having $\mathcal{N}$ as set of nodes. There is an edge from A to B if there exists a production of the form $A \to BC$ or $A \to CB$, for some non-terminal C. Note that it is possible for the dependency graph to contain self-loops. When we speak of *reachability* in a directed graph, we always mean reachability by a *directed* path. The length of a path π is equal to the number of edges it contains and is denoted by $l(\pi)$.

We classify the nodes in the dependency graph as follows: a node is *recursive* when it belongs to a directed cycle. It is *leeching* when it can reach a recursive node, but is not recursive itself. Finally, it is *restricted* when it is neither recursive nor leeching. The start symbol must be either recursive or leeching, in order to obtain an infinite language (see [Example 1] for the intuition). We denote the set of recursive non-terminals by $\mathcal{N}_{rec}$, the set of leeching non-terminals by $\mathcal{N}_{leech}$ and the set of restricted non-terminals by $\mathcal{N}_{res}$.

Furthermore, without loss of generality, we consider *proper* grammars only, i.e., we assume that all nodes in the dependency graph are reachable from S, and that all non-terminals are productive [Hopcroft and Ullman 1979].

$$\begin{tikzcd}
 & & S \arrow[rr, bend left] \arrow[d, bend left] & & G \\
 & & A \arrow[rr, bend left] \arrow[dr, bend left] & & D \\
F & C \arrow[l, bend left] \arrow[ur, bend left] & & B \arrow[ll, bend left] \arrow[d, bend left] & \\
 & & & E &
\end{tikzcd}$$

Figure 1: The dependency graph of context-free grammar G_1.

For each string $s \in L(G_A)$, there exists at least one parse tree that yields s and in which the root of the parse tree is labelled with A. The *depth* of a parse tree τ is the length of a longest path (number of edges) from the root to a leaf and is denoted by $d(\tau)$. Note that we do not restrict the grammar in terms of ambiguity: ambiguous grammars are allowed, hence each string may have multiple parse trees. This leads to the notion of a *minimal parse tree* of a string: a parse tree of minimum depth. Note that a string may have more than one minimal parse tree rooted at a non-terminal A. A parse tree is called *non-recursive* if no path contains two nodes labeled with the same non-terminal.

Example 1. Consider the following context-free grammar G_1:

$$\begin{array}{ll}
S \to AG & C \to c \\
A \to BD & D \to d \\
B \to CE & E \to e \\
C \to AF & F \to f \\
 & G \to g.
\end{array}$$

The corresponding dependency graph is shown in [Fig. 1]. We observe the following classification of the non-terminals:

- $\mathcal{N}_{rec} = \{A, B, C\}$;
- $\mathcal{N}_{leech} = \{S\}$;
- $\mathcal{N}_{res} = \{D, E, F, G\}$.

$$\begin{aligned}
A^0 &= \{a \in \Sigma \mid A \to a \in \mathcal{P}\};\\
A^{i+1} &= A^i \cup \{u \cdot v \mid \exists B, C \in \mathcal{N} : A \to BC \in \mathcal{P} \wedge u \in B^i \wedge v \in C^i\};\\
\Delta A^0 &= A^0;\\
\Delta A^{i+1} &= A^{i+1} \setminus A^i.
\end{aligned}$$

Figure 2: The naive concatenation scheme for context-free grammars.

3 Algorithm

We now present an iterative algorithm that generates the language described by a given, fixed grammar G.

During the execution of the algorithm, every non-terminal is associated with a set of terminal strings. By ΔA^i we denote the set of all terminal strings generated in iteration i for non-terminal A. By A^i we denote the set of all terminal strings generated in iterations 0 to i for non-terminal A.

The iterations are computed according to the standard inductive *concatenation scheme* shown in [Fig. 2]. It is easy to see that $A^i \subseteq A^{i+1}$ for each $A \in \mathcal{N}$ and all $i \in \mathbb{N}$.

Remark. In the second rule of the scheme, the use of A^0 instead of A^i would yield equivalent definitions. □

The strings in S^i are called *output strings*, these are the strings in $L(G)$. Note that the terminal productions are only used in iteration 0 and the non-terminal productions are only used in the subsequent iterations.

The set ΔA^i contains all strings that can be obtained by combining previously generated strings, according to the associated production(s) of A, except for those that have already been generated. Note that we are working with sets: duplicates are removed, but it is still possible that in the same iteration, or in two different iterations, two identical strings are generated for a non-terminal A (see [Example 2]).

We denote the length of a string s by $|s|$ and the maximal length of a string in A^i by ω_A^i. Note that it is possible for A^i to be empty when $i < |\mathcal{N}| - 1$ (this will be shown in [Section 4.2]), in which case ω_A^i is undefined. Clearly, $\omega_A^{i+1} \geq \omega_A^i$ holds for $i \geq |\mathcal{N}| - 1$.

We define

$$\begin{aligned}
\mathcal{T}^i &= \bigcup_{A \in \mathcal{N}} A^i,\\
\Delta\mathcal{T}^i &= \bigcup_{A \in \mathcal{N}} \Delta A^i.
\end{aligned}$$

In addition to the output strings, these sets also contain the strings that are only used as building blocks for the output strings, and are not output strings

Non-terminal N	ΔN^0	ΔN^1	ΔN^2	ΔN^3	...
S	{}	{ab}	{abb}	{abbb}	...
A	{a}	{ab}	{abb}	{abbb}	...
B	{b}	{}	{}	{}	...
C	{a}	{}	{}	{}	...

Table 1: Generated output and intermediate strings in the first iterations of applying the naive concatenation scheme to the context-free grammar G_2.

themselves. These are called the *intermediate strings*. The maximal length of a string in $\mathcal{T}^i$ is denoted by $\omega^i_{\mathcal{T}}$. Note that the same string might be generated for multiple non-terminals, i.e., the union $\cup_{A\in\mathcal{N}} A^i$ that defines $\mathcal{T}^i$ is generally not a disjoint union.

Example 2. Consider the following grammar G_2:

$$\begin{array}{ll} S \rightarrow AB & A \rightarrow a \\ S \rightarrow CB & B \rightarrow b \\ A \rightarrow AB & C \rightarrow a. \end{array}$$

[Table 1] shows the results of the concatenation scheme applied on G_2 for the first few iterations. Observe that the string ab is generated both by both $S \rightarrow CB$ and $S \rightarrow AB$ in two different iterations. In iteration 2 the string is already present in S^1, hence it is not in ΔS^2, even though it is in S^2. The output strings shown in the table (S^3) are ab, abb and abbb. The intermediate strings shown in the table are a, b, ab, abb and abbb (this set equals $\mathcal{T}^3$).

Remark. An equivalent but more efficient inductive concatenation scheme, which avoids duplicate concatenations is the well-known "semi-naive" scheme [see Ceri et al. 1990], which is shown in [Fig. 3]. Although this semi-naive scheme can give practical improvements in performance, e.g., in applications to dabatases [see Bancilhon and Ramakrishnan 1986], the theoretical worst-case complexity is of the same order as that of the standard scheme. In this paper we will prove that the standard scheme runs in polynomial incremental time. □

$$\begin{aligned} A^0 &= \{a \in \Sigma \mid A \rightarrow a \in \mathcal{P}\}; \\ A^{i+1} &= A^i \cup \{u \cdot v \mid \exists B, C \in \mathcal{N} : (A \rightarrow BC) \in \mathcal{P} \\ &\qquad \wedge \big((u \in B^i \wedge v \in \Delta C^i) \vee (u \in \Delta B^i \wedge v \in C^{i-1})\big)\}; \\ \Delta A^{i+1} &= A^{i+1} \setminus A^i. \end{aligned}$$

Figure 3: The semi-naive concatenation for context-free grammars.

4 Upper and Lower Bounds on the Number and Length of Generated Strings

Our main result is that the naive algorithm satisfies the IPT property.In order to prove this, four important results are obtained in this section:

- A formalization of the start-up phase in [Section 4.2].
- A relation between the iteration number and the number of intermediate strings in [Section 4.3].
- A relation between the maximum string length and the number of strings in [Section 4.4].
- A relation between the number of intermediate strings and the number of output strings in [Section 4.5].

4.1 String properties

Lemma 1. *For any non-terminal A, the set ΔA^i consists precisely of the strings that can be derived from A and have a minimal parse tree depth of $i + 1$.*

Proof. We prove the lemma by induction on i.

Base For $i = 0$, ΔA^0 contains all strings that can be derived from A in one step.

Induction For $i > 0$, suppose the lemma holds for all values smaller than i. Consider a string $s = u \cdot v \in \Delta A^i$ with $u \in B^{i-1}$, $v \in C^{i-1}$ and $A \to BC \in \mathcal{P}$ for some $B, C \in \mathcal{N}$. By induction u and v have minimal parse trees τ_u and τ_v of depth at most i.

It remains to show that all strings with a minimal parse tree of depth $i + 1$, that can be derived from A, belong to ΔA^i. Thereto, consider such a string $s \in L(G_A)$ that has a minimal parse tree τ of depth $i + 1$.

We first show that $s \in A^i$. Since $i > 0$, τ has the form of an A-root with two children τ_B and τ_C and $A \to BC \in \mathcal{P}$ for some $B, C \in \mathcal{N}$. Let u and v be the strings yielded by τ_B and τ_C respectively, so $s = u \cdot v$. Since τ has depth $i + 1$, the trees τ_B and τ_C both have a depth $\leq i$. By induction, $u \in \Delta B^j$ and $v \in \Delta C^k$, for some $j, k < i$. In particular, $u \in B^{i-1}$ and $v \in C^{i-1}$. It is now obvious from the definition of A^i that $s = u \cdot v \in A^i$.

Finally, we show that $s \in \Delta A^i = A^i \backslash A^{i-1}$ by proving that $s \notin A^{i-1}$. Suppose that $s \in A^{i-1}$. By induction, s has a minimal parse tree of depth $\leq i$, which contradicts our assumption. □

Knowledge about the iteration in which a string is generated gives us information about the length of the string. Because the yield of a parse tree of depth $i+1$ has length at least $i+1$ and at most 2^i, the lemma above implies the following:

Lemma 2. $\forall i \in \mathbb{N} : \forall s \in \Delta A^i : i+1 \leq |s| \leq 2^i$.

4.2 Start-up Phase

We look at the first $|\mathcal{N}|$ iterations of the algorithm: the *start-up phase*. After this initial start-up, all non-terminals will have generated at least one (intermediate) string:

Lemma 3 (Start-up phase ending). $\forall A \in \mathcal{N} : A^{|\mathcal{N}|-1} \neq \emptyset$.

Proof. Consider the following definitions:

$$\begin{aligned}
\mathcal{N}^0 &= \{A \in \mathcal{N} \mid \exists a : (A \to a) \in \mathcal{P}\};\\
\mathcal{N}^{i+1} &= \mathcal{N}^i \cup \{A \in \mathcal{N} \mid \exists B, C \in \mathcal{N}^i : (A \to BC) \in \mathcal{P}\};\\
\Delta\mathcal{N}^0 &= \mathcal{N}^0;\\
\Delta\mathcal{N}^{i+1} &= \mathcal{N}^{i+1} \setminus \mathcal{N}^i.
\end{aligned}$$

Note that $\mathcal{N}^i \subseteq \mathcal{N}^{i+1}$ for all i. Intuitively, a non-terminal A is in $\Delta\mathcal{N}^i$ iff it generates its first string in iteration i. By induction, we readily verify that

$$\forall A \in \mathcal{N} : A \in \mathcal{N}^i \Leftrightarrow A^i \neq \emptyset.$$

Since for each non-terminal N there exists a non-recursive parse tree rooted at N, the above implies that $\mathcal{N}^{|\mathcal{N}|-1} = \mathcal{N}$. Hence, $\forall A \in \mathcal{N} : A^{|\mathcal{N}|-1} \neq \emptyset$. □

Remark. After iteration $|\mathcal{N}| - 1$, restricted non-terminals do not generate any new strings:

$$A \in \mathcal{N}_{res}, j \geq |\mathcal{N}| : \Delta A^j = \emptyset.$$

□

4.3 Generation pace

In this section we discuss the “speed” at which strings are generated: the *generation pace*.

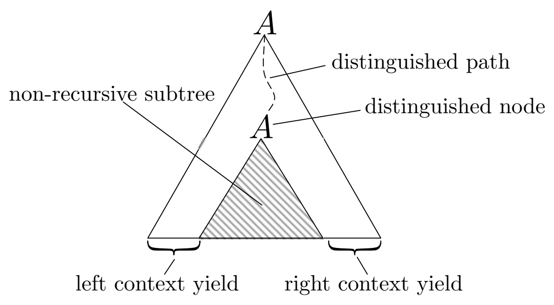

Figure 4: A filled context for a recursive non-terminal A.

4.3.1 Recursive Non-terminals

The following definition is illustrated in [Fig. 4].

Definition 4 (Filled context). Let $A \in \mathcal{N}_{rec}$. A *filled context* τ for A is a parse tree rooted at A with the following properties:

1. A occurs at least twice (note that the root node is already labelled with A);
2. some non-root A-node is called the *distinguished node*. The path from the root to the distinguished node is called the *distinguished path*. No non-terminal occurs more than once on the distinguished path, except for A, which appears exactly twice on the distinguished path;
3. for any non-terminal node x not lying on the distinguished path, the subtree rooted at x is non-recursive; and
4. the subtree rooted at the distinguished node is non-recursive.

The *context yield* of τ is the yield of τ without the yield of the distinguished node. The *left context yield* (resp. *right context yield*) is the yield of τ before (resp. after) the yield of the distinguished node. The *context length* is the length of the context yield.

By a standard pumping lemma argument [see Hopcroft and Ullman 1979], we can construct the following.

Lemma 5. *For each $A \in \mathcal{N}_{rec}$ there exists a filled context. Moreover, let τ be a filled context for A of depth δ and distinguished path length ζ. For all $i \geq 0$, there exists a parse tree τ_i for A, such that*

$$d(\tau_i) = \delta + i \cdot \zeta.$$

Furthermore,

$$|s_i| = \gamma + i \cdot \rho,$$

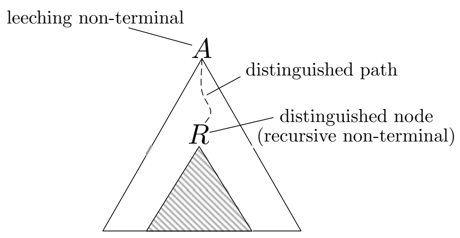

Figure 5: An R-filled context for a leeching non-terminal A.

where s_i is the yield of τ_i, γ is the length of the yield of τ and ρ is the context length of τ.

We conclude with the following lower bound on the number of strings generated by recursive non-terminals.

Lemma 6. *There exists a constant c, such that:*

$$\forall A \in \mathcal{N}_{rec}, \forall i \geq 0 : |A^{c+|\mathcal{N}|\cdot i}| > i.$$

Proof. Let τ be a filled context for A of depth δ with a distinguished path length of ζ. By [Lemma 5], for each $i \geq 0$, we have a different string s_i with a parse tree τ_i of depth $\delta + i \cdot \zeta$, where $\delta \leq 2|\mathcal{N}|$ and $\zeta \leq |\mathcal{N}|$. Hence, each τ_i yields a new string lastly in iteration $2|\mathcal{N}| + i \cdot |\mathcal{N}|$. Therefore, $|A^{2|\mathcal{N}|+i\cdot|\mathcal{N}|}| - 1 \geq i$, or $|A^{2|\mathcal{N}|+i\cdot|\mathcal{N}|}| > i$, for $i \geq 0$. □

4.3.2 Leeching Non-terminals

In [Definition 4], we defined the notion of filled context for a recursive non-terminal. We now define the analogous notion for a leeching non-terminal; this is illustrated in [Fig. 5].

Definition 7 (Filled context). Let $A \in \mathcal{N}_{leech}$. A *filled context* for A is a parse tree for A with the following properties:

1. it contains at least one recursive non-terminal;
2. it is non-recursive; and
3. some recursive node is called the *distinguished node*. The path from the root to the distinguished node is called the *distinguished path*.

When the distinguished node is labeled with a recursive non-terminal R, we call the tree a *R-filled context* for A. The notions of (left and right) context yield, and of context length are defined in the same way as in [Definition 4].

In analogy to [Lemma 5] and [Lemma 6], we have the following.

Lemma 8. *For each $A \in \mathcal{N}_{leech}$ there exists a filled context. Moreover, let τ be a B-filled context for A having depth δ and distinguished path length ζ. Let τ_B be a filled context for B having depth δ_B and distinguished path length ζ_B. For all $i \geq \eta$ there exists a parse tree τ_i for A, such that*

$$d(\tau_i) = \zeta + \delta_B + i \cdot \zeta_B.$$

Furthermore

$$|s_i| = \rho + \gamma_B + i \cdot \rho_B,$$

where s_i is the yield of τ_i, ρ is the context length of τ, γ_B is the length of the yield of τ_B and ρ_B is the context length of τ_B.

The next lemmas show a relationship between the iteration number and the number of generated strings up to that iteration.

Lemma 9. *There exists a constant c, such that:*

$$\forall A \in \mathcal{N}_{leech}, \forall i \geq 0 : |A^{c+|\mathcal{N}|\cdot i}| > i.$$

Proof. Consider a recursive non-terminal R, reachable by A, having a context of depth δ_R with a distinguished path length of ζ_R. By [Lemma 8], for each $i \geq |\mathcal{N}|$, we have a different string with a parse tree of depth $\zeta + \delta_R + i \cdot \zeta_R$, where $\zeta \leq |\mathcal{N}| - 1$, $\delta_R \leq 2|\mathcal{N}|$ and $\zeta_R \leq |\mathcal{N}|$. Hence, each τ_i yields a new string lastly in iteration $|\mathcal{N}| - 1 + 2|\mathcal{N}| + i \cdot |\mathcal{N}| = 3|\mathcal{N}| + i \cdot |\mathcal{N}| - 1$. Therefore, $|A^{3|\mathcal{N}|+i\cdot|\mathcal{N}|-1}| - 1 \geq i - |\mathcal{N}|$, or $|A^{3|\mathcal{N}|+i\cdot|\mathcal{N}|-1}| > i - |\mathcal{N}|$, for $i \geq |\mathcal{N}|$. By substituting i' for $i - |\mathcal{N}|$ we obtain that, for $i' \geq 0$, $|A^{3|\mathcal{N}|+(i'+|\mathcal{N}|)\cdot|\mathcal{N}|-1}| > i'$, or $|A^{3|\mathcal{N}|+i'\cdot|\mathcal{N}|+|\mathcal{N}|^2-1}| > i'$. □

Corollary 10 (Non-terminal lower bound). *There exists a constant c, such that:*

$$\forall A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}, \forall i \geq 0 : |\mathcal{N}| \cdot |A^{i+c}| > i.$$

Proof. Consider the constants c_1 and c_2 from [Lemma 6] and [Lemma 9], respectively. We can choose c_3 as the maximum of c_1 and c_2 and get:

$$\forall A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}, \forall k \geq 0 : |A^{c_3+|\mathcal{N}|\cdot k}| > k.$$

Now let $A \in \mathcal{N}_{rec} \cup \mathcal{N}_{leech}$ and i be an arbitrary natural number. We distinguish the following two cases to show that the corollary holds:

(a) i is a multiple of $|\mathcal{N}|$.
Hence, $i = |\mathcal{N}| \cdot k$. We can now argue as follows:

$$\begin{aligned} |A^{c_3+|\mathcal{N}|\cdot k}| > k &\Leftrightarrow |A^{c_3+|\mathcal{N}|\cdot \frac{i}{|\mathcal{N}|}}| > \frac{i}{|\mathcal{N}|} \\ &\Leftrightarrow |\mathcal{N}| \cdot |A^{c_3+i}| > i. \end{aligned}$$

(b) i is not a multiple of $|\mathcal{N}|$.
Let i' be the next multiple of $|\mathcal{N}|$ larger than i. We have $i < i' < i + |\mathcal{N}|$ and by (a) we know that $i' < |\mathcal{N}| \cdot |A^{c_3+i'}|$. Also $|A^j| \leq |A^{j'}|$ when $j \leq j'$, since $A^j \subseteq A^{j'}$. Combining these observations we get:

$$i < i' < |\mathcal{N}| \cdot |A^{c_3+i'}| < |\mathcal{N}| \cdot |A^{c_3+i+|\mathcal{N}|}|.$$

When we choose $c = c_3 + |\mathcal{N}|$, we have $|\mathcal{N}| \cdot |A^{i+c}| > i$, for $i \geq 0$. □

This readily implies the following.

Corollary 11 (Intermediate lower bound). *There exists a constant c such that*

$$\forall i \geq 0 : |\mathcal{N}| \cdot |\mathcal{T}^{i+c}| \geq i.$$

4.4 Length bound

In this section, we establish a relation between the length of the longest string and the total number of generated strings up to an iteration. We begin by stating the following lemma, which can be proven by a pumping argument.

Lemma 12. *Let $s \in A^i$ with $|s| \geq 2^{|\mathcal{N}|}$. Then A^i also contains a shorter string s' with*

$$|s| - 2^{|\mathcal{N}|} < |s'| < |s|.$$

Lemma 13. $\forall A \in \mathcal{N}, \forall i \geq |\mathcal{N}| - 1 : \omega^i_A < 2^{|\mathcal{N}|} \cdot |A^i|.$

Proof. Since $i \geq |\mathcal{N}| - 1$, we know by [Lemma 3] that there exists some $s \in A^i$. If $\omega^i_A < 2^{|\mathcal{N}|}$ then the lemma is trivial. Else, let $j = \lfloor \omega^i_A / 2^{|\mathcal{N}|} \rfloor$. [Lemma 12] can be repeatedly applied at least j times, starting from $s_0 = s$, yielding j additional distinct strings $s_1, s_2, \ldots, s_j \in A^i$. Hence, $|A^i| \geq j + 1$, and therefore $|A^i| > \omega^i_A / 2^{|\mathcal{N}|}$. □

[Lemma 13] readily implies the relation announced at the beginning of this subsection.

Corollary 14 (Length bound). $\forall i \geq |\mathcal{N}| - 1 : \omega^i_{\mathcal{T}} < 2^{|\mathcal{N}|} \cdot |\mathcal{T}^i|.$

4.5 Intermediate string bound

In this section we bound the number of intermediate strings by the number of output strings.

Lemma 15 (Past bound). $\forall i \geq 0 : \forall 0 \leq k \leq i : |\mathcal{T}^i| \leq 2^{2^k-1} \cdot |\mathcal{T}^{i-k}|^{2^k}$.

Proof. We prove the lemma by induction on i.

Basis For $i = 0$, the only possible value for k is 0 and the inequality $|\mathcal{T}^0| \leq 2^{2^0-1} \cdot |\mathcal{T}^0|^{2^0}$ becomes trivial.

Induction For $i > 0$, assume the lemma holds for $i - 1$:

$$|\mathcal{T}^{i-1}| \leq 2^{2^k-1} \cdot |\mathcal{T}^{i-1-k}|^{2^k} \qquad (0 \leq k \leq i-1).$$

By the remark made after introducing the concatenation scheme in Section 3, we have for all $j \geq 0 : \mathcal{T}^{j+1} \subseteq (\mathcal{T}^j \cdot \mathcal{T}^j) \cup \mathcal{T}^0$. We get:

$$\begin{aligned}
|\mathcal{T}^i| &\leq |(\mathcal{T}^{i-1} \cdot \mathcal{T}^{i-1}) \cup \mathcal{T}^0| \\
&\leq |\mathcal{T}^{i-1}| \cdot |\mathcal{T}^{i-1}| + |\mathcal{T}^0| \\
&= |\mathcal{T}^{i-1}|^2 + |\mathcal{T}^0| \\
&\leq 2 \cdot |\mathcal{T}^{i-1}|^2 && (\mathcal{T}^0 \subseteq \mathcal{T}^{i-1}) \\
&\leq 2 \cdot \left(2^{2^k-1} \cdot |\mathcal{T}^{i-1-k}|^{2^k}\right)^2 && \text{(by induction hypothesis, } 0 \leq k \leq i-1) \\
&= 2^{2^{k+1}-1} \cdot |\mathcal{T}^{i-(k+1)}|^{2^{k+1}} && (0 \leq k \leq i-1) \\
&= 2^{2^k-1} \cdot |\mathcal{T}^{i-k}|^{2^k} && (1 \leq k \leq i) \\
&= 2^{2^k-1} \cdot |\mathcal{T}^{i-k}|^{2^k} && (0 \leq k \leq i, k = 0 \text{ is immediate}).
\end{aligned}$$

□

The following lemma is obvious from [Lemma 3].

Lemma 16 (String Growth). *For each edge $A \to B$ in the dependency graph the following holds:*

$$\forall i \geq |\mathcal{N}| - 1 : \forall s \in B^i : \exists s' \in A^{i+1} : s \textit{ is a strict substring of } s'.$$

We are now in a position to formulate the Future and Present bounds, which will be important in the proof of our main theorem.

Lemma 17 (Future bound). $\forall i \geq |\mathcal{N}| - 1 : |\mathcal{T}^i| \leq (\omega_S^{i+|\mathcal{N}|})^2 \cdot |S^{i+|\mathcal{N}|}|$.

Proof. We first prove that every intermediate string will appear as a substring of an output string, several iterations later. Next, we bound the number of substrings of these output strings to obtain the desired bound.

Any string $s \in \mathcal{T}^i$ appears in A^i for some A. Consider a simple path π from S to A in the dependency graph; π has length at most $|\mathcal{N}|$. By repeatedly applying [Lemma 16] we know that $S^{i+l(\pi)}$ contains a string s' that is a superstring of s. Because $l(\pi) \leq |\mathcal{N}|$ it holds that $s' \in S^{i+l(\pi)} \subseteq S^{i+|\mathcal{N}|}$. Hence, each string in $\mathcal{T}^i$ has a superstring in $S^{i+|\mathcal{N}|}$.

The number of substrings of a string s' is bounded by $|s'|^2$. Consequently, the number of substrings we can create using strings in $S^{i+|\mathcal{N}|}$ is bounded by $(\omega_S^{i+|\mathcal{N}|})^2 \cdot |S^{i+|\mathcal{N}|}|$. Together with the first observation, this gives $|\mathcal{T}^i| \leq (\omega_S^{i+|\mathcal{N}|})^2 \cdot |S^{i+|\mathcal{N}|}|$, as desired. □

Corollary 18 (Present bound). $\exists c \in \mathbb{N} : \forall i \geq |\mathcal{N}| - 1 : |\mathcal{T}^i| \leq c \cdot |S^i|^{2^{|\mathcal{N}|+2}}$.

Proof. We combine the Future and Past bounds to bound $|\mathcal{T}^i|$:

$$\begin{aligned}
|\mathcal{T}^i| &\leq 2^{2^{|\mathcal{N}|-1}} \cdot |\mathcal{T}^{i-|\mathcal{N}|}|^{2^{|\mathcal{N}|}} && \text{(Past bound)}\\
&\leq 2^{2^{|\mathcal{N}|-1}} \cdot \left((\omega_S^i)^2 \cdot |S^i|\right)^{2^{|\mathcal{N}|}} && \text{(Future bound)}\\
&= 2^{2^{|\mathcal{N}|-1}} \cdot (\omega_S^i)^{2^{|\mathcal{N}|+1}} \cdot |S^i|^{2^{|\mathcal{N}|}}\\
&\leq 2^{2^{|\mathcal{N}|-1}} \cdot (2^{|\mathcal{N}|} \cdot |S^i|)^{2^{|\mathcal{N}|+1}} \cdot |S^i|^{2^{|\mathcal{N}|}} && \text{(Lemma 13)}\\
&\leq c \cdot |S^i|^{2^{|\mathcal{N}|+2}}.
\end{aligned}$$

□

5 The Naive Algorithm Runs in Incremental Polynomial Time

In order to prove that our naive generation algorithm yields an enumeration in *incremental polynomial time* in the sense of [Johnson et al. 1988], we only require the following proposition.

Proposition 19. *There exists a fixed polynomial p such that after each iteration i, the total time spent by the naive algorithm [Fig. 2] so far is bounded by $p(|S^{i-1}|)$.*

Proof. We will first look at the time necessary to generate one string, then at the time necessary to generate one iteration and finally at the time needed to generate strings up to an iteration i.

Consider an intermediate string $s \in A^i$. When $i = 0$, the only thing that needs to happen is to store s, given that there are no duplicate productions. When $i > 0$, the following steps need to be performed:

1. concatenate two strings to form s;
2. check if the string has already been generated for A (duplicate check);
3. save the string in order to check for duplicates later.

The concatenation of two strings, resulting in s, can be done in time $\mathcal{O}\left(|s|\right)$. A lookup and insertion, to keep track of the set of generated strings, can both be done in time $\mathcal{O}\left(|A^i| \cdot |s|\right)$.

Next, we construct a bound for the total number of intermediate strings calculated in iteration $i > 0$. In the worst case, all strings in $\mathcal{T}^{i-1}$ will be pairwise combined, for each production. Hence, the total number of candidates in iteration i is bounded by $|\mathcal{P}| \cdot |\mathcal{T}^{i-1}|^2$, where $|\mathcal{P}|$ is equal to the number of productions in the grammar.

Combining the two observations above gives us an upper bound on the total work in iteration i: $\mathcal{O}\left(\omega_{\mathcal{T}}^{i} \cdot |\mathcal{T}^{i}| \cdot |\mathcal{T}^{i-1}|^2\right)$. From the Past bound we know that $|\mathcal{T}^i| = \mathcal{O}\left(|\mathcal{T}^{i-1}|^2\right)$. The total work done up to and including iteration i is therefore bounded by

$$\mathcal{O}\left(\sum_{j=1}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4\right).$$

Note that the work in iteration 0 is constant, since it requires storing just one string for each terminal production. The work in the first $|\mathcal{N}|$ iterations is also bounded by a constant:

$$\sum_{j=1}^{|\mathcal{N}|-1} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4 \leq |\mathcal{N}| \cdot \omega_{\mathcal{T}}^{|}\mathcal{N}| \cdot |\mathcal{T}^{|\mathcal{N}|-1}|^4 = \mathcal{O}\left(1\right).$$

Hence, the total time spent up to and including iteration $|\mathcal{N}| - 1$ is considered constant.

In the remainder of the proof, we bound $\mathcal{O}\left(\sum_{j=|\mathcal{N}|}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4\right)$ by a polynomial in $|S^{i-1}|$. First, observe the following:

$$\begin{aligned}
\sum_{j=|\mathcal{N}|}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4 &\leq i \cdot \omega_{\mathcal{T}}^{i} \cdot |\mathcal{T}^{i-1}|^4 \\
&< c_1 \cdot i \cdot |\mathcal{T}^{i}| \cdot |\mathcal{T}^{i-1}|^4 && \text{(Corollary 14)} \\
&\leq c_2 \cdot |\mathcal{T}^{i+c_3}| \cdot |\mathcal{T}^{i}| \cdot |\mathcal{T}^{i-1}|^4 && \text{(Corollary 11)} \\
&\leq c_4 \cdot |\mathcal{T}^{i-1}|^{c_5} && \text{(Past bound)} \\
&\leq c_6 \cdot |S^{i-1}|^{c_7} && \text{(Present bound)}
\end{aligned}$$

for constants $c_1, \ldots, c_7$.

Note that the applied lemmas only hold from iteration $|\mathcal{N}| - 1$ on. This is not a problem as they are only applied for $j \geq |\mathcal{N}|$. From the above we can conclude:

$$\sum_{j=1}^{i} \omega_{\mathcal{T}}^{j} \cdot |\mathcal{T}^{j-1}|^4 = \mathcal{O}\left(|S^{i-1}|^c\right),$$

for some constant c.

The time needed by the algorithm to calculate all intermediate strings up to and including interation i is bounded by $\mathcal{O}\left(|S^{i-1}|^c\right)$, which is clearly polynomial in the size of S^{i-1}, as desired. □

Theorem 20. *There is a fixed polynomial p such that the entire language $L(G)$ can be enumerated without duplicates in such a way that the time needed to output the $(m+1)$th output string is bounded by $p(m)$.*

Proof. Consider the $(m + 1)$th output string s. We know that $s \in \Delta S^i$ for some i and we also know, by [Proposition 19] that the time needed to calculate all strings up to and including iteration i is bounded by $\mathcal{O}\left(|S^{i-1}|^c\right)$, for some constant c. Since $|S^{i-1}| \leq m$, we obtain a polynomial in m as desired. □

6 From Given-Length to Infinite Enumeration

The purpose of this section is to show that we can always use an algorithm for given-length enumeration with polynomial delay (GLEPD) to obtain an algorithm for infinite enumeration in incremental polynomial time (IEIPT).

For a fixed context-free grammar G, consider a GLEPD-algorithm that, given a natural number n, enumerates all strings $w \in L(G)$ with $|w| = n$. We treat the algorithm as a black box and denote it by $\texttt{Enumerate}_G(n)$. The polynomial delay property holds for the algorithm: there exists a fixed polynomial p_D such that, on input n, the time before the first output, the time between two outputs and the time after the last output until the algorithm terminates, is bounded by $p_D(n)$.

From this algorithm, we can derive the algorithm $\texttt{Enumerate}_{G,\infty}$ [Fig. 6]. We now prove that $\texttt{Enumerate}_{G,\infty}$ enumerates the entire language $L(G)$ in IPT.

The following lemma readily follows from [Lemma 5] and [Lemma 8].

Lemma 21. *For each infinite context-free language $\mathcal{L}$, there exist two constants $c \in \mathbb{N} \setminus \{0\}$ and $d \in \mathbb{N}$ such that for each $l \in \mathbb{N}$ the language $\mathcal{L}$ contains at least one string of length $c \cdot l + d$.*

Theorem 22 ($\texttt{Enumerate}_{G,\infty}$ runs in IPT). *Let G be a context-free grammar. There exists a fixed polynomial p such that in the algorithm $\texttt{Enumerate}_{G,\infty}$ the time spent between the mth output and the $(m+1)$th output is bounded by $p(m)$, where $m > 0$.*

```
Input: None
Output: all strings in $L(G)$
for $i \leftarrow 1$ to $\infty$ do
    Enumerate$_G(i)$
end
```

Figure 6: The algorithm $\texttt{Enumerate}_{G,\infty}$.

Proof. Consider the mth and the $(m+1)$th output strings that are generated consecutively by the algorithm and denote them by s_m and s_{m+1}, respectively. For algorithm $\texttt{Enumerate}_G(n)$ we have the polynomial $p_D(n)$, guaranteed by the polynomial delay property. We may assume p_D is monotonically increasing over the natural numbers. This can be achieved by converting all negative coefficients to positive.

There are two cases to consider:

- $|s_m| = |s_{m+1}|$.
  This means that the strings are generated in the same iteration i. Let c and d be the constants given by [Lemma 21]. We consider two further cases:
  (a) $i \leq d$.
  Let c_0 be the total time performed by algorithm $\texttt{Enumerate}_{G,\infty}$ in the iterations up to and including iteration d. Then clearly the time between the outputs s_m and s_{m+1} is bounded by c_0.
  (b) $i > d$.
  Let $l = \lfloor \frac{i-1-d}{c} \rfloor$. By [Lemma 21], at least $l+1$ strings have already been generated before iteration i. Hence $l+1 < m$. As $l = \lfloor \frac{i-1-d}{c} \rfloor < m$, we obtain $i \leq m \cdot c + d$. As the time between s_m and s_{m+1} is bounded by $p_D(i)$, it is also bounded by $p_D(m \cdot c + d)$, because p_D is monotonically increasing. This is clearly a polynomial in m.
- $|s_m| < |s_{m+1}|$.
  This means that the strings are generated in different iterations. Let i be the iteration in which s_m was generated and j be the iteration in which s_{m+1} was generated. Clearly, $1 \leq i < j$. The total time spent between outputs s_m and s_{m+1} consists of three parts:
  - the time spent in iteration i after the generation of s_m;
  - the time spent in iteration j before the generation of s_{m+1};
  - the time spent in iterations $i+1, \ldots, j-1$.

The first two parts are bounded by $p_D(i)$ and $p_D(j)$ respectively. In every iteration k between i and j, the time needed to verify that there is no string of length k in $L(G)$ is bounded by $p_D(k)$. Hence, the total time spent between s_m and s_{m+1} is bounded by

$$p_D(i) + p_D(j) + \sum_{k=i+1}^{j-1} p_D(k) \leq p_D(j) + p_D(j) + (j-2) \cdot p_D(j) = j \cdot p_D(j).$$

We know from [Lemma 21] that the maximal number of consecutive lengths for which no string exists is bounded by a constant. Hence, for some constant c_{wait} we have $j - i \leq c_{\text{wait}}$. The total time spent between s_m and s_{m+1} is therefore bounded by

$$p'(i) := (i + c_{\text{wait}}) \cdot p_D(i + c_{\text{wait}}),$$

which is clearly a polynomial in i. As in the previous case, $i \leq c \cdot m + d$, so we obtain $p'(c \cdot m + d)$ as a polynomial in m.

The proof is completed by taking for $p(m)$ the larger of the two polynomials from the two cases, increased by the constant c_0. □

7 Conclusion

The fact that the simple algorithm, based on the naive bottom-up concatenation scheme and described in [Section 3], already achieves the Incremental Polynomial Time criterion, is, we hope, an interesting theoretical (if not didactical) contribution of this paper, as we have not seen this mentioned elsewhere. An important caveat is that the context-free grammar G is considered fixed and not part of the input. An interesting question to investigate is what happens when G ís part of the input.

An elementary approach as presented here has the best chances of being generalizable. Indeed, we are currently investigating how the insights developed here can be extended to apply to the more general setting of context-free sets of arbitrary combinatorial objects as introduced by [Courcelle and Engelfriet 2012] and [Flajolet and Sedgewick 2009, Flajolet et al. 1991]. A major additional problem in this context is to keep the duplicate check (step 2 in the proof of [Proposition 19]) polynomial. Fortunately, in the HR approach to graph rewriting, every context-free graph language has bounded treewidth. In combination with imposing connectedness and a degree bound [see Matoušek and Thomas 1992] this may produce a polynomial duplicate check.

We also note that for unambiguous grammars, the methods of [Flajolet and Sedgewick 2009] can be used to count exactly the number of strings (or derivation trees, which coincides for unambiguous grammars) of a given size.

Acknowledgement

We thank Hendrik Blockeel and Frank Neven for their contributions in the initial phase of the project.

References

[Ackerman and Mäkinen 2009] Ackerman, M., Mäkinen, E.: "Three New Algorithms for Regular Language Enumeration"; Computing and Combinatorics, 15th Annual International Conference, COCOON 2009, Niagara Falls, NY, USA, July 13-15, 2009, Proceedings; (2009); 178–191.

[Arnold and Sleep 1980] Arnold, D. B., Sleep, M. R.: "Uniform Random Generation of Balanced Parenthesis Strings"; ACM Trans. Program. Lang. Syst.; 2, 1 (1980), 122–128.

[Baeten et al. 1993] Baeten, J. C. M., Bergstra, J. A., Klop, J. W.: "Decidability of Bisimulation Equivalence for Processes Generating Context-Free Languages"; J. ACM; 40, 3 (1993), 653–682.

[Bancilhon and Ramakrishnan 1986] Bancilhon, F., Ramakrishnan, R.: "An Amateur's Introduction to Recursive Query Processing Strategies"; SIGMOD Rec.; 15, 2 (1986), 16–52.

[Ceri et al. 1990] Ceri, S., Gottlob, G., Tanca, L.: "Logic Programming and Databases"; Springer (1990).

[Courcelle and Engelfriet 2012] Courcelle, B., Engelfriet, J.: "Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach"; volume 138 of Encyclopedia of mathematics and its applications; Cambridge University Press (2012).

[Dömösi 2000] Dömösi, P.: "Unusual Algorithms for Lexicographical Enumeration"; Acta Cybern.; 14, 3 (2000), 461–468.

[Dong 2009] Dong, Y.: "Linear algorithm for lexicographic enumeration of CFG parse trees."; Science in China Series F: Information Sciences; 52, 7 (2009), 1177–1202.

[Duncan and Hutchinson 1981] Duncan, A. G., Hutchinson, J.: "Using Attributed Grammars to Test Designs and Implementations"; Proceedings 5th International Conference on Software Engineering; IEEE Press (1981); 170–178.

[Flajolet et al. 1991] Flajolet, P., Salvy, B., Zimmermann, P.: "Automatic Average-Case Analysis of Algorithm"; Theor. Comput. Sci.; 79, 1 (1991), 37–109.

[Flajolet and Sedgewick 2009] Flajolet, P., Sedgewick, R.: "Analytic Combinatorics"; Cambridge University Press (2009).

[Flajolet et al. 1994] Flajolet, P., Zimmermann, P., Cutsem, B. V.: “A Calculus for the Random Generation of Labelled Combinatorial Structures”; Theor. Comput. Sci.; 132, 2 (1994), 1–35.

[Gore et al. 1997] Gore, V., Jerrum, M., Kannan, S., Sweedyk, Z., Mahaney, S. R.: “A Quasi-Polynomial-Time Algorithm for Sampling Words from a Context-Free Language”; Inf. Comput.; 134, 1 (1997), 59–74.

[Hopcroft and Ullman 1979] Hopcroft, J. E., Ullman, J. D.: “Introduction to Automata Theory, Languages, and Computation”; Addison-Wesley, Reading, Massachusetts (1979).

[Johnson et al. 1988] Johnson, D. S., Yannakakis, M., Papadimitriou, C. H.: “On Generating All Maximal Independent Sets”; Information Processing Letters; 27, 3 (1988), 119–123.

[Lämmel 2001] Lämmel, R.: “Grammar Testing”; FASE; (2001); 201–216.

[Mäkinen 1997] Mäkinen, E.: “On Lexicographic Enumeration of Regular and Context-Free Languages”; Acta Cybern.; 13, 1 (1997), 55–62.

[Matoušek and Thomas 1992] Matoušek, J., Thomas, R.: “On the complexity of finding iso- and other morphisms for partial k-trees”; Discrete Mathematics; 108, 1-3 (1992), 343–364.

[Maurer 1990] Maurer, P. M.: “Generating Test Data with Enhanced Context-Free Grammars.”; IEEE Software; 7, 4 (1990), 50–55.

[Purdom 1972] Purdom, P.: “A Sentence Generator for Testing Parsers”; j-BIT; 12, 3 (1972), 366–375.

[Somerville 1998] Somerville, I.: “Software Engineering”; Addison-Wesley (1998); 5th edition.

[Xu et al. 2011] Xu, Z., Zheng, L., Chen, H.: “A Toolkit for Generating Sentences from Context-Free Grammars”; Int. J. Software and Informatics; 5, 4 (2011), 659–676.