Grammars for Document Spanners

Liat Peterfreund ✉
DI ENS, ENS, CNRS, PSL University, Paris, France
Inria, Paris, France

Abstract

We propose a new grammar-based language for defining information-extractors from documents (text) that is built upon the well-studied framework of document spanners for extracting structured data from text. While previously studied formalisms for document spanners are mainly based on regular expressions, we use an extension of context-free grammars, called extraction grammars, to define the new class of context-free spanners. Extraction grammars are simply context-free grammars extended with variables that capture interval positions of the document, namely spans. While regular expressions are efficient for tokenizing and tagging, context-free grammars are also efficient for capturing structural properties. Indeed, we show that context-free spanners are strictly more expressive than their regular counterparts. We reason about the expressive power of our new class and present a pushdown-automata model that captures it. We show that extraction grammars can be evaluated with polynomial data complexity. Nevertheless, as the degree of the polynomial depends on the query, we present an enumeration algorithm for unambiguous extraction grammars that, after quintic preprocessing, outputs the results sequentially, without repetitions, with a constant delay between every two consecutive ones.

2012 ACM Subject Classification Information systems → Information extraction; Information systems → Relational database model; Information systems → Data model extensions

Keywords and phrases Information Extraction, Document Spanners, Context-Free Grammars, Constant-Delay Enumeration, Regular Expressions, Pushdown Automata

Digital Object Identifier 10.4230/LIPIcs.ICDT.2021.7

Related Version *Full Version*: `https://arxiv.org/abs/2003.06880`

Funding *Liat Peterfreund*: A part of this work was done while affiliated with IRIF – CNRS & Université de Paris, Paris, France, and with University of Edinburgh, Edinburgh, UK. This work is supported by the Fondation des Sciences Mathématiques de Paris (FSMP).

Acknowledgements I would like to thank the anonymous reviewers for their extremely valuable comments, and in particular those that enabled me to extend the enumeration algorithm to a broader class of extraction grammars. I am grateful to Arnaud Durand, Michael Kaminski, Benny Kimelfeld, and Leonid Libkin for useful discussions, and to Dominik D. Freydenberger for references.

1 Introduction

The abundance and availability of valuable textual resources in the last decades position text analytics as a standard component in data-driven workflows. One of the core operations that aims to facilitate the analysis and integration of textual content is Information Extraction (IE), the extraction of structured data from text. IE arises in a large variety of domains, including social media analysis [4], health-care analysis [43], customer relationship management [1], information retrieval [45], and more.

Rules have always been a key component in various paradigms for IE, and their roles have varied and evolved over the time. Systems such as Xlog [38] and IBM's SystemT [27, 6] use rules to extract relations from text (e.g., tokenizer, dictionary lookup, and part-of-speech tagger) that are further manipulated with relational query languages. Other systems use rules to generate features for machine-learning classifiers [26, 35].

x	y
$[1,2\rangle$	$[2,3\rangle$
$[3,4\rangle$	$[4,5\rangle$
$[3,4\rangle$	$[4,6\rangle$

Figure 1 Extracted relation.

$$S \to B \vdash_x \mathtt{a}A\mathtt{b} \dashv_y B$$
$$A \to \mathtt{a}A\mathtt{b} \mid \dashv_x\vdash_y$$
$$B \to \mathtt{a}B \mid \mathtt{b}B \mid \epsilon$$

Figure 2 Production rules.

$$\vdash_x \mathtt{aa} \dashv_x\vdash_y \mathtt{bb} \dashv_y \mathtt{b}$$
$$\mathtt{aa} \vdash_x \mathtt{aa} \dashv_x\vdash_y \mathtt{bb} \dashv_y \mathtt{b}$$
$$\mathtt{aa} \vdash_x \mathtt{a} \dashv_x\vdash_y \mathtt{b} \dashv_y \mathtt{b}$$

Figure 3 Ref-words.

Document Spanners. The framework of document spanners, presented by Fagin et al., provides a theoretical basis for investigating the principles of relational rule systems for IE [11]. The research on document spanners has focused on their expressive power [11, 14, 34, 17, 32, 19] their computational complexity [2, 13, 18, 33], incompleteness [29, 33], and other system aspects such as cleaning [12], dynamic complexity [20], distributivity [7] and an annotated variant [8].

In the documents spanners framework, a *document* $\mathbf{d}$ is a string over a fixed finite alphabet, and a *spanner* is a function that extracts from a document a relation over the spans of $\mathbf{d}$. A *span* x is a half-open interval of positions of $\mathbf{d}$ and it represents a substring $\mathbf{d}_x$ of $\mathbf{d}$ that is identified by these positions. A natural way to specify a spanner is by a *regex formula*: a regular expression with embedded *capture variables* that are viewed as relational attributes. For instance, the spanner that is given by the regex formula $(\mathtt{a} \vee \mathtt{b})^* \vdash_x \mathtt{aa}^* \dashv_x\vdash_y \mathtt{bb}^* \dashv_y (\mathtt{a} \vee \mathtt{b})^*$ extracts from documents spans x and y that correspond, respectively, with a non-empty substring of `a`'s followed by a non-empty substring of `b`'s. In particular, it extracts from the document `ababb` the relation depicted in Figure 1.

The class of *regular spanners* is the class of spanners definable as the closure of regex formulas under positive relational algebra operations: projection, natural join and union. The class of regular spanners can be represented alternatively by finite state machines, namely *variable-set automata (vset-automata)*, which are nondeterministic finite-state automata that can open and close variables (that, as in the case of regex formulas, play the role of the attributes of the extracted relation). *Core* spanners [11] are obtained by extending the class of regular spanners with string-equality selection on span variables. Although core spanners can express strictly more than regular spanners, they are still quite limited as, e.g., there is no core spanner that extracts all pairs x and y of spans having the same *length* [11].

To date, most research on spanners has been focused on the regular representation, that is, regular expressions and finite state automata. While regular expressions are useful for segmentation and tokenization, they are not useful in describing complex nested structures (e.g., syntactic structure of a natural language sentence) and relations between different parts of the text. Regular languages also fall short in dealing with tasks such as syntax highlighting [30] and finding patterns in source code [39]. For all of the above mentioned tasks we have context-free grammars. It is well known that context-free languages are strictly more expressive than regular languages. Büchi [5] has showed that regular languages are equivalent to monadic second order logic (over strings), and Lautemann et al. [25] have showed that adding an existential quantification over a binary relation interpreted as a matching is enough to express all context-free languages. This quantification, intuitively, is what makes it possible to also express structural properties.

Contribution. In this work we propose a new grammar-based approach for defining the class of *context-free spanners*. Context-free spanners are defined via *extraction grammars* which, like regex formulas, incorporate *capture variables* that are viewed as relational attributes.

Extraction grammars produce *ref-words* which are words over an extended alphabet that consists of standard terminal symbols along with *variable operations* that denote opening and closing of variables. The result of evaluating an extraction grammar on a document $\mathbf{d}$ is defined via the ref-words that are produced by the grammar and equal to $\mathbf{d}$ after erasing the variable operations. For example, the extraction grammar from Figure 2 produces also the ref-words $\vdash_x \mathtt{a} \dashv_x \vdash_y \mathtt{b} \dashv_y \mathtt{abb}$ and $\mathtt{ab} \vdash_x \mathtt{a} \dashv_x \vdash_y \mathtt{b} \dashv_y \mathtt{b}$. Hence, it extracts from $\mathbf{d} := \mathtt{ababb}$ the two first tuples from the relation in Figure 1. In Figure 3 there are additional examples of ref-words produced by this grammar. In general, the given grammar extracts from documents the spans x and y that correspond, respectively, with a non-empty substring of $\mathtt{a}$'s followed by an equal-length substring of $\mathtt{b}$'s. With a slight adaptation of Fagin et al. inexpressibility proof [11, Theorem 4.21], it can show that this spanner is inexpressible by core spanners.

Indeed, we show that context-free spanners are strictly more expressive than regular spanners and that the restricted class of regular extraction grammars captures the regular spanners. We compare the expressiveness of context-free spanners against core and generalized core spanners and show that context-free spanners are incomparable to any of these classes. In addition to extraction grammars, we present a pushdown automata model that captures the context-free spanners.

In term of evaluation of context-free spanners we can evaluate extraction grammars in polynomial time in *data complexity*, where the spanner is regarded as fixed and the document as input. However, as the degree of this polynomial depends on the query (in particular, on the number of variables in the relation it extracts), we propose an enumeration algorithm for unambiguous extraction grammars. Our algorithm outputs the results consecutively, after quintic preprocessing, with constant delay between every two answers. In the first step of the preprocessing stage we manipulate the extraction grammar so that it will be adjusted to the input document. In the second step of the preprocessing we change it in a way that its non-terminals include extra information on the variable operations. This extra information enables us to skip sequences of productions that do not affect the output, hence obtaining a delay that is independent of the input document and linear in the number of variables associated with the spanner.

Related Work. Grammar-based parsers are widely used in IE systems [44, 37]. There are, as well, several theoretical frameworks that use grammars for IE, one of which is Knuth's framework of attribute grammars [23, 24]. In this framework, the non-terminals of a grammar are attached with attributes[1] that pass semantic information up and down a parse-tree. While both extraction grammars and attribute grammars extract information via grammars, it seems as if the expressiveness of these formalisms is incomparable to extraction grammars.

The problem of enumerating words of context-free grammars arises in different contexts [41, 31]. Providing complexity guarantees on the enumeration is usually tricky and requires assumptions either on the grammar or on the output. Mäkinen [28] has presented an enumeration algorithm for regular grammars and for unambiguous context-free grammars with additional restrictions (strongly prefix-free and length complete). Later, Dömösi [9] has presented an enumeration algorithm for unambiguous context-free grammars that outputs, with quadratic delay, only the words of a fixed length.

Organization. In Section 2, we present extraction grammars and extraction pushdown automata. In Section 3, we discuss the expressive power of context-free spanners and their evaluation. In Sections 4 and 5, we present our enumeration algorithm, and in Section 6 we conclude.

[1] The term "attributes" was previously used in the relational context; Here the meaning is different.

2 Context-Free Spanners

In this section we present the class of context-free spanners by presenting two formalisms for expressing them: extraction grammars and extraction pushdown automata.

2.1 Preliminaries

We start by presenting the formal setup based on notations and definitions used in previous works on document spanners (e.g., [11, 18]).

Strings and Spans. We set an infinite set Vars of variables and fix a finite alphabet Σ that is disjoint of Vars. In what follows we assume that our alphabet Σ consists of at least two letters. A *document* $\mathbf{d}$ is a finite sequence over Σ whose length is denoted by $|\mathbf{d}|$. A *span* identifies a substring of $\mathbf{d}$ by specifying its bounding indices. Formally, if $\mathbf{d} = \sigma_1 \cdots \sigma_n$ where $\sigma_i \in \Sigma$ then a span of $\mathbf{d}$ has the form $[i, j\rangle$ where $1 \leq i \leq j \leq n+1$ and $\mathbf{d}_{[i,j\rangle}$ denotes the substring $\sigma_i \cdots \sigma_{j-1}$. When $i = j$ it holds that $\mathbf{d}_{[i,j\rangle}$ equals the empty string, which we denote by ϵ. We denote by $\mathsf{Spans}(\mathbf{d})$ the set of all possible spans of a document $\mathbf{d}$.

Document Spanners. Let $X \subseteq \mathsf{Vars}$ be a finite set of variables and let $\mathbf{d}$ be a document. An $(X, \mathbf{d})$-mapping assigns spans of $\mathbf{d}$ to variables in X. An $(X, \mathbf{d})$-relation is a finite set of $(X, \mathbf{d})$-mappings. A *document spanner* (or *spanner*, for short) is a function associated with a finite set X of variables that maps documents $\mathbf{d}$ into $(X, \mathbf{d})$-relations.

2.2 Extraction Grammars

The *variable operations* of a variable $x \in \mathsf{Vars}$ are $\vdash_x$ and $\dashv_x$ where, intuitively, $\vdash_x$ denotes the opening of x, and $\dashv_x$ its closing. For a finite subset $X \subseteq \mathsf{Vars}$, we define the set $\Gamma_X := \{\vdash_x, \dashv_x \mid x \in X\}$. That is, Γ_X is the set that consists of all the variable operations of all variables in X. We assume that Σ and Γ_X are disjoint. We extend the classical definition of context-free grammars [22] by treating the variable operations as special terminal symbols. Formally, a *context-free extraction grammar*, or *extraction grammar* for short, is a tuple $G := (X, V, \Sigma, P, S)$ where

- $X \subseteq \mathsf{Vars}$ is a finite set of variables,
- V is a finite set of *non-terminal* symbols[2],
- Σ is a finite set of *terminal* symbols;
- P is a finite set of *production rules* of the form $A \to \alpha$ where A is a non-terminal and $\alpha \in (V \cup \Sigma \cup \Gamma_X)^*$, and
- S is a designated non-terminal symbol referred to as the *start symbol*.

We say that the extraction grammar G is *associated* with X.

▶ **Example 1.** In this and in the following examples we often denote the elements in V by upper case alphabet letters from the beginning of the English alphabet $(A, B, C, \ldots)$. Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and let us consider the grammar DISJEQLEN associated with the variables $\{x, y\}$ that is given by the following production rules:

- $S \to B \vdash_x A \dashv_y B \mid B \vdash_y A \dashv_x B$
- $A \to \mathtt{a}A\mathtt{a} \mid \mathtt{a}A\mathtt{b} \mid \mathtt{b}A\mathtt{b} \mid \mathtt{b}A\mathtt{a}$

[2] Note that these are often referred to as variables, however, here we use the term "non-terminals" to distinguish between these symbols and elements in Vars.

- $A \to \dashv_x B \vdash_y \mid \dashv_y B \vdash_x$
- $B \to \epsilon \mid \mathtt{a}B \mid \mathtt{b}B$

Here and in what follows, we use the compact notation for production rules by writing $A \to \alpha_1 \mid \cdots \mid \alpha_n$ instead of the productions $A \to \alpha_1, \cdots, A \to \alpha_n$. As we shall later see, this grammar extracts pairs of disjoint spans with the same length. ⌟

While classical context-free grammars generate words, extraction grammars generate words over the extended alphabet $\Sigma \cup \Gamma_X$. These words are referred to as *ref-words* [36]. Similarly to (classical) context-free grammars, the process of deriving ref-words is defined via the notations $\Rightarrow, \Rightarrow^n, \Rightarrow^*$ that stand for one, n, and several (possibly zero) derivation steps, respectively. To emphasize the grammar being discussed, we sometime use the grammar as a subscript (e.g., $\Rightarrow^*_G$). For the full definitions we refer the reader to Hopcroft et al. [22]. A non-terminal A is called *useful* if there is some derivation of the form $S \Rightarrow^* \alpha A \beta \Rightarrow^* w$ where $w \in (\Sigma \cup \Gamma_X)^*$. If A is not useful then it is called *useless*. For complexity analysis, we define the *size* $|G|$ of an extraction grammar G as the sum of the number of symbols at the right-hand sides (i.e., to the right of $\to$) of its rules.

2.3 Semantics of Extraction Grammars

Following Freydenberger [15] we define the semantics of extraction grammars using ref-words. A ref-word $\mathbf{r} \in (\Sigma \cup \Gamma_X)^*$ is *valid (for X)* if each variable of X is opened and then closed exactly once, or more formally, for each $x \in X$ the string $\mathbf{r}$ has precisely one occurrence of $\vdash_x$, precisely one occurrence of $\dashv_x$, and the former is before (i.e., to the left of) the latter.

▶ **Example 2.** The ref-word $\mathbf{r}_1 := \vdash_x \mathtt{aa} \dashv_y \vdash_x \mathtt{ab} \dashv_y$ is not valid for $\{x, y\}$ whereas the ref-words $\mathbf{r}_2 := \vdash_x \mathtt{aa} \dashv_x \vdash_y \mathtt{ab} \dashv_y$ and $\mathbf{r}_3 := \vdash_y \mathtt{a} \dashv_y \vdash_x \mathtt{a} \dashv_x \mathtt{ab}$ are valid for $\{x, y\}$. ⌟

To connect ref-words to terminal strings and later to spanners, we define a morphism $\mathsf{clr}\colon (\Sigma \cup \Gamma_X)^* \to \Sigma^*$ by $\mathsf{clr}(\sigma) := \sigma$ for $\sigma \in \Sigma$, and $\mathsf{clr}(\tau) := \epsilon$ for $\tau \in \Gamma_X$. For $\mathbf{d} \in \Sigma^*$, let $\mathsf{Ref}(\mathbf{d})$ be the set of all valid ref-words $\mathbf{r} \in (\Sigma \cup \Gamma_X)^*$ with $\mathsf{clr}(\mathbf{r}) = \mathbf{d}$. By definition, every $\mathbf{r} \in \mathsf{Ref}(\mathbf{d})$ has a unique factorization $\mathbf{r} = \mathbf{r}'_x \cdot \vdash_x \cdot \mathbf{r}_x \cdot \dashv_x \cdot \mathbf{r}''_x$ for each $x \in X$. With these factorizations, we interpret $\mathbf{r}$ as a $(X, \mathbf{d})$-mapping $\mu^{\mathbf{r}}$ by defining $\mu^{\mathbf{r}}(x) := [i, j\rangle$, where $i := |\mathsf{clr}(\mathbf{r}'_x)| + 1$ and $j := i + |\mathsf{clr}(\mathbf{r}_x)|$. An alternative way of understanding $\mu^{\mathbf{r}} = [i, j\rangle$ is that i is chosen such that $\vdash_x$ occurs between the positions in $\mathbf{r}$ that are mapped to σ_{i-1} and σ_i, and $\dashv_x$ occurs between the positions that are mapped to σ_{j-1} and σ_j (assuming that $\mathbf{d} = \sigma_1 \cdots \sigma_{|\mathbf{d}|}$, and slightly abusing the notation to avoid a special distinction for the non-existing positions σ_0 and $\sigma_{|\mathbf{d}|+1}$).

▶ **Example 3.** Let $\mathbf{d} = \mathtt{aaab}$. The ref-word $\mathbf{r}_2$ from Example 2 is interpreted as the $(\{x, y\}, \mathbf{d})$-mapping $\mu^{\mathbf{r}_2}$ defined by $\mu^{\mathbf{r}_2}(x) := [1, 3\rangle$ and $\mu^{\mathbf{r}_2}(y) := [3, 5\rangle$. ⌟

Extraction grammars define ref-languages which are sets of ref-words. The ref-language $\mathcal{R}(G)$ of an extraction grammar $G := (X, V, \Sigma, P, S)$ is defined by $\mathcal{R}(G) := \{\mathbf{r} \in (\Sigma \cup \Gamma_X)^* \mid S \Rightarrow^* \mathbf{r}\}$. Note that we use $\mathcal{R}(G)$ instead of $\mathcal{L}(G)$ being used for standard grammars, to emphasize that the produced language is a ref-language. (We also use $\mathcal{L}(G)$ when G is a standard grammar.) To illustrate the definition let us consider the following example.

▶ **Example 4.** Both ref-words $\mathbf{r}_1$ and $\mathbf{r}_2$ from Example 2 are in $\mathcal{R}(\textsc{disjEqLen})$ where $\textsc{disjEqLen}$ is the grammar described in Example 1. Producing both $\mathbf{r}_1$ and $\mathbf{r}_2$ starts similarly with the sequence: $S \Rightarrow B \vdash_x A \dashv_y B \Rightarrow^2 \vdash_x A \dashv_y \Rightarrow \vdash_x \mathtt{a}A\mathtt{b} \dashv_y \Rightarrow \vdash_x \mathtt{aa}A\mathtt{ab} \dashv_y$. The derivation of $\mathbf{r}_1$ continues with $\Rightarrow \vdash_x \mathtt{aa} \dashv_y B \vdash_x \mathtt{ab} \dashv_y \Rightarrow \vdash_x \mathtt{aa} \dashv_y \vdash_x \mathtt{ab} \dashv_y$ whereas that of $\mathbf{r}_2$ continues with $\Rightarrow \vdash_x \mathtt{aa} \dashv_x B \vdash_y \mathtt{ab} \dashv_y \Rightarrow \vdash_x \mathtt{aa} \dashv_x \vdash_y \mathtt{ab} \dashv_y$. ⌟

We denote by $\mathsf{Ref}(G)$ the set of all ref-words in $\mathcal{R}(G)$ that are valid for X. Finally, we define the set $\mathsf{Ref}(G, \mathbf{d})$ of ref-words in $\mathsf{Ref}(G)$ that clr maps to $\mathbf{d}$. That is, $\mathsf{Ref}(G, \mathbf{d}) := \mathsf{Ref}(G) \cap \mathsf{Ref}(\mathbf{d})$. The result of evaluating the spanner $[\![G]\!]$ on a document $\mathbf{d}$ is then defined as

$$[\![G]\!](\mathbf{d}) := \{\mu^{\mathbf{r}} \mid \mathbf{r} \in \mathsf{Ref}(G, \mathbf{d})\}.$$

▶ **Example 5.** Let us consider the document $\mathbf{d} := \texttt{aaba}$. The grammar DISJEQLEN maps $\mathbf{d}$ into a set of $(\{x, y\}, \mathbf{d})$-mappings, amongst are $\mu^{\mathbf{r}_2}$ that is defined by $\mu^{\mathbf{r}_2}(x) := [1, 3\rangle$ and $\mu^{\mathbf{r}_2}(y) := [3, 5\rangle$ and $\mu^{\mathbf{r}_3}$ that is defined by $\mu^{\mathbf{r}_3}(x) := [2, 3\rangle$ and $\mu^{\mathbf{r}_3}(y) := [1, 2\rangle$. It can be shown that the grammar DISJEQLEN maps every document $\mathbf{d}$ into all possible $(\{x, y\}, \mathbf{d})$-mappings μ such that $\mu(x)$ and $\mu(y)$ are disjoint (i.e., do not overlap) and have the same length (i.e., $|\mathbf{d}_{\mu(x)}| = |\mathbf{d}_{\mu(y)}|$). ⌟

A spanner S is said to be *definable* by an extraction grammar G if $S(\mathbf{d}) = [\![G]\!](\mathbf{d})$ for every document $\mathbf{d}$.

▶ **Definition 6.** *A* context-free spanner *is a spanner definable by an extraction grammar.*

2.4 Extraction Pushdown Automata

An *extraction pushdown automaton*, or *extraction* PDA, is associated with a finite set $X \subseteq \mathsf{Vars}$ of variables and can be viewed as a standard pushdown automata over the extended alphabet $\Sigma \cup \Gamma_X$. Formally, an *extraction* PDA is a tuple $A := (X, Q, \Sigma, \Delta, \delta, q_0, Z, F)$ where X is a finite set of variables; Q is a finite set of states; Σ is the input alphabet; Δ is a finite set which is called *the stack alphabet*; δ is a mapping $Q \times \left(\Sigma \cup \{\epsilon\} \cup \Gamma_X\right) \times \Delta \to 2^{Q \times \Delta^*}$ which is called the *transition function*; $q_0 \in Q$ is the *initial state*; $Z \in \Delta$ is the *initial stack symbol*; and $F \subseteq Q$ is the set of *accepting states*. Indeed, extraction PDAs run on ref-words (i.e., finite sequences over $\Sigma \cup \Gamma_X$), as opposed to classical PDAs whose input are words (i.e., finite sequences over Σ). Similarly to classical PDAs, the computation of extraction PDAs can be described using sequences of configurations: a *configuration* of A is a triple (q, w, γ) where q is the state, w is the remaining input, and γ is the stack content such that the top of the stack is the left end of γ and its bottom is the right end. We use the notation $\vdash^*$ similarly to how it is used in the context of PDAs [21] and define the ref-language $\mathcal{R}(A)$:

$$\mathcal{R}(A) := \{\mathbf{r} \in (\Sigma \cup \Gamma_X)^* \mid \exists \alpha \in \Delta^*, q_f \in F : (q_0, \mathbf{r}, Z) \vdash^* (q_f, \epsilon, \alpha)\}.$$

We denote the language of A by $\mathcal{R}(A)$ to emphasize that it is a ref-language, and denote by $\mathsf{Ref}(A)$ the set of all ref-words in $\mathcal{R}(A)$ that are valid for X. The result of evaluating the spanner $[\![A]\!]$ on a document $\mathbf{d}$ is then defined as

$$[\![A]\!](\mathbf{d}) := \{\mu^{\mathbf{r}} \mid \mathbf{r} \in \mathsf{Ref}(A) \cap \mathsf{Ref}(\mathbf{d})\}.$$

▶ **Example 7.** We define the extraction PDA that maps a document $\mathbf{d}$ into the set of $(\{x, y\}, \mathbf{d})$-mappings μ where $\mu(x)$ ends before $\mu(y)$ starts and their lengths are the same. The stack alphabet consists of the bottom symbol $\perp$ and C, and the transition function δ is described in Figure 4 where a transition from state q to state q' that is labeled with $\tau, A/\ \gamma$ denotes that the automaton moves from state q to state q' upon reading τ with A at the top of the stack, while replacing A with γ. We can extend the automaton in a symmetric way such that it will represent the same spanner as that represented by the grammar DISJEQLEN from Example 1. ⌟

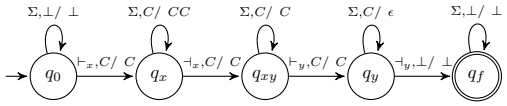

Figure 4 Transition function of Example 7.

$$S \to B \vdash_x A_1 \dashv_y B \mid B \vdash_y A_2 \dashv_x B$$
$$A_i \to \mathtt{a} A_i \mathtt{a} \mid \mathtt{a} A_i \mathtt{b} \mid \mathtt{b} A_i \mathtt{b} \mid \mathtt{b} A_i \mathtt{a}, \ i = 1, 2$$
$$A_1 \to \dashv_x B \vdash_y , \quad A_2 \to \dashv_y B \vdash_x$$
$$B \to \epsilon \mid \mathtt{a} B \mid \mathtt{b} B$$

Figure 5 Productions of Example 9.

We say that a spanner S is *definable* by an extraction PDA A if for every document $\mathbf{d}$ it holds that $[\![A]\!](\mathbf{d}) = S(\mathbf{d})$. Treating the variable operations as terminal symbols enables us to use the equivalence of PDAs and context-free grammars and conclude the following straightforward observation.

▶ **Proposition 8.** *The class of spanners definable by extraction grammars is equal to the class of spanners definable by extraction PDAs.*

Thus, we have also an automata formalism for defining context-free spanners.

2.5 Functional Extraction Grammars

Freydenberger and Holldack [16] have presented the notion of *functionality* in the context of regular spanners. We now extend it to extraction grammars. The intuition is that interpreting an extraction grammar as a spanner disregards ref-words that are not valid. We call an extraction grammar G *functional* if every ref-word in $\mathcal{R}(G)$ is valid.

▶ **Example 9.** The grammar DISJEQLEN in our running example is not functional. Indeed, we saw in Example 4 that the ref-word $\mathbf{r}_1$, although it is not valid, is in $\mathcal{R}(\text{DISJEQLEN})$. We can, however, simply modify the grammar to obtain an equivalent functional one. Notice that the problem arises due to the production rules $S \to B \vdash_x A \dashv_y B$ and $S \to B \vdash_y A \dashv_x B$. For the non-terminal A we have $A \Rightarrow^* \mathbf{r}_1$ where $\mathbf{r}_1$ contains both $\dashv_x$ and $\vdash_y$, and we also have $A \Rightarrow^* \mathbf{r}_2$ where $\mathbf{r}_2$ contains both $\dashv_y$ and $\vdash_x$. To fix that, we can replace the non-terminal A with two non-terminals, namely A_1 and A_2, and change the production rules so that for every ref-word $\mathbf{r}$, if $A_1 \Rightarrow^* \mathbf{r}$ then $\mathbf{r}$ contains both $\dashv_x$ and $\vdash_y$, and if $A_2 \Rightarrow^* \mathbf{r}$ then $\mathbf{r}$ contains both $\dashv_y$ and $\vdash_x$. It can be shown that the grammar G whose production rules appear in Figure 5 is functional and that $[\![G]\!] = [\![\text{DISJEQLEN}]\!]$. ⌟

▶ **Proposition 10.** *Every extraction grammar G can be converted into an equivalent functional extraction grammar G' in $O(|G|^2 + 3^{2k}|G|)$ time where k is the number of variables G is associated with.*

Inspired by Chomsky's hierarchy, we say that an extraction grammar is in *Chomsky Normal Form (CNF)* if it is in CNF when viewed as a grammar over the extended alphabet $\Sigma \cup \Gamma_X$. We remark that, in Proposition 10, G' is in CNF.

2.6 Unambiguous Extraction Grammars

A grammar G is said to be unambiguous if every word it produces has a unique parse-tree. We extend this definition to extraction grammars. An extraction grammar G is said to be *unambiguous* if for every document $\mathbf{d}$ and every $(X, \mathbf{d})$-mapping $\mu \in [\![G]\!](\mathbf{d})$ it holds that there is a unique ref-word $\mathbf{r}$ for which $\mu^{\mathbf{r}} = \mu$ and this ref-word has a unique parse-tree. Unambiguous extraction grammars are less expressive than their ambiguous counterparts as the Boolean case shows – unambiguous context-free grammars are less expressive than ambiguous context-free grammars [22].

▶ **Example 11.** The extraction grammar given in Example 9 is not unambiguous since it produces the ref-words $\vdash_x \dashv_x \vdash_y \dashv_y$ and $\vdash_y \dashv_y \vdash_x \dashv_x$ that correspond to the same mapping. It can be shown that replacing the derivation $B \to \epsilon$ with $B \to \mathsf{a} \mid \mathsf{b}$ results in an unambiguous extraction grammar which is equivalent to DISJEQLEN on any document different than ϵ. (Note however that this does not imply that the ref-languages both grammars produce are equal.) ⌟

Our main enumeration algorithm for extraction grammars relies on unambiguity and the following observation.

▶ **Proposition 12.** *In Proposition 10, if G is unambiguous then so is G'.*

3 Expressive Power and Evaluation

In this section we compare the expressiveness of context-free spanners compared to other studied classes of spanners and discuss its evaluation shortly.

3.1 Regular Spanners

A *variable-set automaton* A (or *vset-automaton*, for short) is a tuple $A := (X, Q, q_0, q_f, \delta)$ where $X \subseteq \mathsf{Vars}$ is a finite set of variables also referred to as $\mathsf{Vars}(A)$, Q is the set of *states*, $q_0, q_f \in Q$ are the *initial* and the *final* states, respectively, and $\delta \colon Q \times (\Sigma \cup \{\epsilon\} \cup \Gamma_X) \to 2^Q$ is the *transition function*. To define the semantics of A, we interpret A as a non-deterministic finite state automaton over the alphabet $\Sigma \cup \Gamma_X$, and define $\mathcal{R}(A)$ as the set of all ref-words $\mathbf{r} \in (\Sigma \cup \Gamma_X)^*$ such that some path from q_0 to q_f is labeled with $\mathbf{r}$. Like for regex formulas, we define $\mathsf{Ref}(A, \mathbf{d}) = \mathcal{R}(A) \cap \mathsf{Ref}(\mathbf{d})$ and finally we define for every document $\mathbf{d} \in \Sigma^*$: $[\![A]\!](\mathbf{d}) := \{\mu^{\mathbf{r}} \mid \mathbf{r} \in \mathsf{Ref}(A, \mathbf{d})\}$. The class of *regular spanners* equals the class of spanners that are expressible as a vset-automaton [11].

Inspired by Chomsky's hierarchy, we say that an extraction grammar G is *regular* if its productions are of the form $A \to \sigma B$ and $A \to \sigma$ where A, B are non-terminals and $\sigma \in (\Sigma \cup \Gamma_X)$. We then have the following equivalence that is strongly based on the equivalence of regular grammars and finite state automata.

▶ **Proposition 13.** *The class of spanners definable by regular extraction grammars is equal to the class of regular spanners.*

3.2 (Generalized) Core Spanners

An alternative way to define regular spanners is based on the notion of regex formulas: Formally, a *regex formula* is defined recursively by $\alpha := \emptyset \mid \epsilon \mid \sigma \mid \alpha \vee \alpha \mid \alpha \cdot \alpha \mid \alpha^* \mid \vdash_x \alpha \dashv_x$ where $\sigma \in \Sigma$ and $x \in \mathsf{Vars}$. We denote the set of variables whose variable operations occur in α by $\mathsf{Vars}(\alpha)$, and interpret each regex formula α as a generator of a ref-word language $\mathcal{R}(\alpha)$ over the extended alphabet $\Sigma \cup \Gamma_{\mathsf{Vars}(\alpha)}$. For every document $\mathbf{d} \in \Sigma^*$, we define $\mathsf{Ref}(\alpha, \mathbf{d}) = \mathcal{R}(\alpha) \cap \mathsf{Ref}(\mathbf{d})$, and the spanner $[\![\alpha]\!]$ by $[\![\alpha]\!](\mathbf{d}) := \{\mu^{\mathbf{r}} \mid \mathbf{r} \in \mathsf{Ref}(\alpha, \mathbf{d})\}$. The class of regular spanners is then defined as the closure of regex formulas under the relational algebra operators: union, projection and natural join. (See full definitions in [11].)

In their efforts to capture the core of AQL which is IBM's SystemT query language, Fagin et al. [11] have presented the class of core spanners which is the closure of regex formulas under the positive operators, i.e., union, natural join and projection, along with the string equality selection that is defined as follows Let S be a spanner and let $x, y \in \mathsf{Vars}(S)$, the *string equality selection* $\zeta^=_{x,y} S$ is defined by $\mathsf{Vars}(\zeta^=_{x,y} S) = \mathsf{Vars}(S)$ and, for all $\mathbf{d} \in \Sigma^*$,

$\zeta^{=}_{x,y}S(\mathbf{d})$ is the set of all $\mu \in S(\mathbf{d})$ where $\mathbf{d}_{\mu(x)} = \mathbf{d}_{\mu(y)}$. Note that unlike the join operator that joins mappings that have identical spans in their shared variables, the selection operator compares the substrings of $\mathbf{d}$ that are described by the spans, and does not distinguish between different spans that span the same substrings.

The class of *generalized core spanners* is obtained by adding the difference operator. That is, it is defined as the closure of regex formulas under union, natural join, projection, string equality, and difference. We say that two classes $\mathcal{S}, \mathcal{S}'$ of spanners are *incomparable* if both $\mathcal{S} \setminus \mathcal{S}'$ and $\mathcal{S}' \setminus \mathcal{S}$ are not empty.

▶ **Proposition 14.** *The classes of core spanners and generalized core spanners are each incomparable with the class of context-free spanners.*

We conclude the discussion by a straightforward result on closure properties.

▶ **Proposition 15.** *The class of context-free spanners is closed under union and projection, and not closed under natural join and difference.*

3.3 Evaluating Context-Free Spanners

The *evaluation* problem of extraction grammars is that of computing $[\![G]\!](\mathbf{d})$ where $\mathbf{d}$ is a document and G is an extraction grammar. Our first observation is the following.

▶ **Proposition 16.** *For every extraction grammar G and every document $\mathbf{d}$ it holds that $[\![G]\!](\mathbf{d})$ can be computed in $O(|G|^2 + |\mathbf{d}|^{2k+3}\, k^3\, |G|)$ time where k is the number of variables G is associated with.*

The proof of this proposition is obtained by iterating through all valid ref-words and using the Cocke-Younger-Kasami (CYK) parsing algorithm [22] to check whether the current valid ref-word is produced by G. We can, alternatively, use Valiant's parser [40] and obtain $O(|G|^2 + |\mathbf{d}|^{2k+\omega}\, k^\omega\, |G|)$ where $\omega < 2.373$ is the matrix multiplication exponent [42].

While the evaluation can be done in polynomial time in data complexity (where G is regarded as fixed and $\mathbf{d}$ as input), the output size might be quite big. To be more precise, for an extraction grammar G associated with k variables, the output might consist of up to $|\mathbf{d}|^{2k}$ mappings. Instead of outputting these mappings altogether, we can output them sequentially (without repetitions) after some preprocessing.

Our main enumeration result is the following.

▶ **Theorem 17.** *For every unambiguous extraction grammar G and every document $\mathbf{d}$ there is an algorithm that outputs the mappings in $[\![G]\!](\mathbf{d})$ with delay $O(k)$ after $O(|\mathbf{d}|^5|G|^2 3^{4k})$ preprocessing where k is the number of variables G is associated with.*

Our algorithm consists of two main stages: preprocessing and enumeration. In the preprocessing stage, we manipulate the extraction grammar and do some precomputations which are later exploited in the enumeration stage in which we output the results sequentially. We remark that unambiguity is crucial for the enumeration stage as it allows to output the mappings without repetition.

Through the lens of data complexity, our enumeration algorithm outputs the results with constant delay after quintic preprocessing. That should be contrasted with regular spanners for which there exists a constant delay enumeration algorithm whose preprocessing is linear [2, 13]. In the following sections, we present the enumeration algorithm and discuss its correctness but before we deal with the special case $\mathbf{d} := \epsilon$. In this case, $[\![G]\!](\mathbf{d})$ is either empty or contains exactly one mapping (since, by definition, the document ϵ has exactly

one span, namely $[1, 1\rangle$). Notice that $[\![G]\!](\mathbf{d})$ is empty if and only if G does not produce a ref-word that consists only of variable operations. To check this, it suffices to change the production rules of G by replacing every occurrence of $\tau \in \Gamma_X$ with ϵ, and checking whether the new grammar produces ϵ. This can be done in linear time [21], which completes the proof of this case. From now on it is assumed that $\mathbf{d} \neq \epsilon$.

4 Preprocessing of the Enumeration Algorithm

Due to Propositions 10 and 12, we can assume that our unambiguous extraction grammar is functional and in CNF. As this conversion requires $O(3^{2k}|G|^2)$, it can be counted as part of our preprocessing.

The preprocessing stage consists of two steps: in the first we adjust the extraction grammar to a given document and add subscripts to non-terminals to track this connection, and in the second we use superscripts to capture extra information regarding the variable operations.

4.1 Adjusting the Extraction Grammar to $\mathbf{d}$

Let $G := (X, V, \Sigma, P, S)$ be an extraction grammar in CNF, and let $\mathbf{d} := \sigma_1 \cdots, \sigma_n, n \geq 1$ be a document. The goal of this step is to restrict G so that it will produce only the ref-words which clr maps to $\mathbf{d}$. To this end, we define the grammar $G_\mathbf{d}$ that is associated with the same set X of variables as G, and is defined as follows:

- The non-terminals are $\{A_{i,j} \mid A \in V, 1 \leq i \leq j \leq n\} \cup \{A_\epsilon \mid A \in V\}$,
- the terminals are Σ,
- the initial non-terminal is $S_{1,n}$, and
- the production rules are defined as follows:
  - $A_{i,i} \to \sigma_i$ for any $A \to \sigma_i \in P$,
  - $A_\epsilon \to \sigma$ for any $A \to \sigma \in P$ with $\sigma \in \Gamma_X$,
  - $A_\epsilon \to B_\epsilon C_\epsilon$ for any $A \to BC \in P$,
  - $A_{i,j} \to B_{i,j} C_\epsilon$ for any $1 \leq i \leq j \leq n$ and any $A \to BC \in P$,
  - $A_{i,j} \to B_\epsilon C_{i,j}$ for any $1 \leq i \leq j \leq n$ and any $A \to BC \in P$,
  - $A_{i,j} \to B_{i,i'} C_{i'+1,j}$ for any $1 \leq i \leq i' < j \leq n$ and $A \to BC \in P$.

We eliminate useless non-terminals from $G_\mathbf{d}$ and by a slight abuse of notation refer to the result as $G_\mathbf{d}$ from now on. The intuition behind this construction is that if the subscript of a non-terminal is i, j then this non-terminal produces a ref-word that clr maps to $\sigma_i \cdots \sigma_j$, and if it is ϵ then it produces a ref-word that consists only of variable operations.

▶ **Example 18.** Figure 6 presents a possible parse-tree of a grammar $G_\mathbf{d}$. ⌟

We establish the following connection between G and $G_\mathbf{d}$.

▶ **Lemma 19.** *For every extraction grammar G in CNF, every document $\mathbf{d} := \sigma_1 \cdots \sigma_n$, every non-terminal A of G, and every ref-word $\mathbf{r} \in (\Sigma \cup \Gamma_X)^*$ with $\mathsf{clr}(\mathbf{r}) = \sigma_i \cdots \sigma_j$ the following holds: $A \Rightarrow_G^* \mathbf{r}$ if and only if $A_{i,j} \Rightarrow_{G_\mathbf{d}}^* \mathbf{r}$*

This allows us to conclude the following straightforward corollary.

▶ **Corollary 20.** *For every extraction grammar G in CNF and for every document $\mathbf{d}$, it holds that $\mathsf{Ref}(G, \mathbf{d}) = \mathcal{L}(G_\mathbf{d})$.*

We note that adjusting our extraction grammar to $\mathbf{d}$ is somewhat similar to the CYK algorithm [22] and therefore it is valid on extraction grammars G in CNF. For a similar reason, we obtain the following complexity which is cubic in $|\mathbf{d}|$.

▶ **Proposition 21.** *For every extraction grammar G in CNF and for every document $\mathbf{d}$, it holds that $G_{\mathbf{d}}$ can be constructed in $O(|\mathbf{d}|^3|G|)$.*

Can the complexity of the adjustment be improved? We leave this as an open question. We note, however, that it might be possible to use similar ideas used by Earley's algorithm [10] to decrease the complexity of this step.

4.2 Constructing the Decorated Grammar

The goal of this step of the preprocessing is to encode the information on the produced variable operations within the terminals and non-terminals. We obtain from $G_{\mathbf{d}}$, constructed in the previous step, a new grammar, namely $\textsc{decorGrmr}(G_{\mathbf{d}})$, that produces *decorated words* over the alphabet $\{(\mathbf{x}, i, \mathbf{y}) \mid \mathbf{x}, \mathbf{y} \subseteq \Gamma_X, 1 \leq i \leq n\}$. A terminal $(\mathbf{x}, i, \mathbf{y})$ indicates that $\mathbf{x}$ and $\mathbf{y}$ are variable operations that occur right before and right after σ_i, respectively. (Notice that $\mathbf{x}, \mathbf{y}$ does not necessarily contain all of these variable operations as some of the variable operations that appear, e.g., after i, can be contained in $\mathbf{x}'$ in case $(\mathbf{x}', i+1, \mathbf{y}')$ is the terminal that appears right after $(\mathbf{x}, i, \mathbf{y})$.) This information is propagated also to the non-terminals such that a non-terminal with a superscript $\mathbf{x}, \mathbf{y}$ indicates that $\mathbf{x}$ and $\mathbf{y}$ are variable operations at the beginning and end, respectively, of the sub decorated word produced by this non-terminal. Non-terminals with subscript ϵ are those that produce sequences of variable operations.

To define $\textsc{decorGrmr}(G_{\mathbf{d}})$, we need G to be functional. The following key observation is used in the formal definition of $\textsc{decorGrmr}(G_{\mathbf{d}})$ and is based on the functionality of G.

▶ **Proposition 22.** *For every functional extraction grammar G and every non-terminal A of G there is a set $\mathbf{x}_A \subseteq \Gamma_X$ of variable operations such that for every ref-word $\mathbf{r}$ where $A \Rightarrow^* \mathbf{r}$ the variable operations that appear in $\mathbf{r}$ are exactly those in $\mathbf{x}_A$. Computing all sets $\mathbf{x}_A$ can be done in $O(|G|)$.*

In other words, for functional extraction grammars, the information on the variable operations is stored implicitly in the non-terminals. The grammar $\textsc{decorGrmr}(G_{\mathbf{d}})$ is defined in three steps as we now describe.

Step 1. We set the following production rules for all subsets $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w} \subseteq \Gamma_X$ that are pairwise disjoint:

- $A_{i,i}^{\emptyset,\emptyset} \to \sigma_i$ for every rule $A_{i,i} \to \sigma_i$ in $G_{\mathbf{d}}$,
- $A_\epsilon \to \epsilon$ for every rule $A_\epsilon \to \tau$ in $G_{\mathbf{d}}$ (with $\tau \in \Gamma_X$),
- $A_\epsilon \to B_\epsilon C_\epsilon$ for every rule $A_\epsilon \to B_\epsilon C_\epsilon$ in $G_{\mathbf{d}}$,
- $A_{i,j}^{\mathbf{x},\mathbf{y}\cup\mathbf{x}_C} \to B_{i,j}^{\mathbf{x},\mathbf{y}} C_\epsilon$ for every rule $A_{i,j} \to B_{i,j} C_\epsilon$ in $G_{\mathbf{d}}$ and $\mathbf{x} \cap \mathbf{x}_C = \mathbf{y} \cap \mathbf{x}_C = \emptyset$,
- $A_{i,j}^{\mathbf{x}\cup\mathbf{x}_B,\mathbf{y}} \to B_\epsilon C_{i,j}^{\mathbf{x},\mathbf{y}}$ for every rule $A_{i,j} \to B_\epsilon C_{i,j}$ in $G_{\mathbf{d}}$ and $\mathbf{x} \cap \mathbf{x}_B = \mathbf{y} \cap \mathbf{x}_B = \emptyset$,
- $A_{i,j}^{\mathbf{x},\mathbf{w}} \to B_{i,i'}^{\mathbf{x},\mathbf{y}} C_{i'+1,j}^{\mathbf{z},\mathbf{w}}$ for every rule $A_{i,j} \to B_{i,i'} C_{i'+1,j}$ in $G_{\mathbf{d}}$ and pairwise disjoint $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{w}$,

with $\mathbf{x}_B$ and $\mathbf{x}_C$ defined as in Proposition 22.

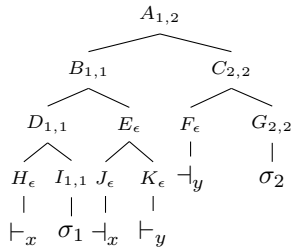

Figure 6 After the adjustment to $\mathbf{d}$.

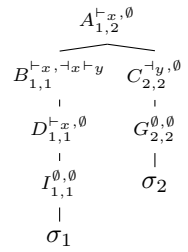

Figure 7 Before step 2 (*iii*).

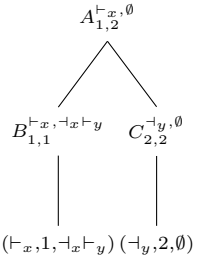

Figure 8 After step 3.

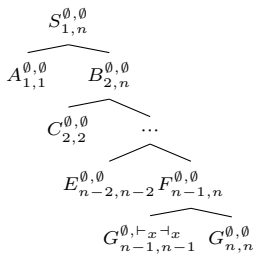

Figure 9 Non-stable non-terminals.

Step 2. We process the resulting grammar by three standard operations [22] in the following order: (*i*) we eliminate useless non-terminals (i.e., those that do not produce a terminal string or are not reachable from the initial non-terminal), (*ii*) we eliminate epsilon-productions, and (*iii*) we eliminate unit productions (i.e., rules of the form $A \to B$ were A, B are non-terminals). We elaborate on (*iii*) as it is important for the sequel. To eliminate unit productions we compute for each non-terminal the set of non-terminals that are reachable from it by unit productions only. That is, we say that a non-terminal B is *reachable* form non-terminal A if there is a sequence of unit productions of the form $A_1 \to A_2, \ldots, A_{n-1} \to A_n$ with $A_1 = A$ and $A_n = B$. We then replace every production $B \to \alpha$ which is not a unit production with $A \to \alpha$, and after that discard all unit productions.

Step 3. The last step of the construction is adding a fresh start symbol S and adding the production rules $S \to S_{1,n}^{\mathbf{x},\mathbf{y}}$ for every non-terminal of the form $S_{1,n}^{\mathbf{x},\mathbf{y}}$. We also replace each production of the form $A_{i,i}^{\mathbf{x},\mathbf{y}} \to \sigma_i$ with $A_{i,i}^{\mathbf{x},\mathbf{y}} \to (\mathbf{x}, i, \mathbf{y})$. This can be viewed as a "syntactic sugar" since it is only intended to help us formulate easily the connection between the grammar G and $\text{DECORGRMR}(G_{\mathbf{d}})$.

▶ **Example 23.** Figures 7 and 8 illustrate the different steps in the construction of the decorated grammar $\text{DECORGRMR}(G_{\mathbf{d}})$. For simplicity, we present the superscripts as pairs of sequences (each represent elements in the set) separated by commas ",". ⌟

Note that by a simple induction it can be shown that the resulting grammar does no longer contain non-terminals of the form A_ϵ. We denote the resulting grammar and its set of non-terminals by $\text{DECORGRMR}(G_{\mathbf{d}})$ and V^{DEC}, respectively.

The (X, d)-mapping μ^w that corresponds with $w := (\mathbf{x}_1, 1, \mathbf{y}_1) \cdots (\mathbf{x}_n, n, \mathbf{y}_n)$ (which is a decorated word produced by $\text{DECORGRMR}(G_{\mathbf{d}})$) is defined by $\mu^w(x) = [i, j\rangle$ where $\vdash_x \in \mathbf{x}_i \cup \mathbf{y}_{i-1}$ and $\dashv_x \in \mathbf{x}_j \cup \mathbf{y}_{j-1}$ with $\mathbf{y}_0 = \mathbf{x}_{n+1} = \emptyset$. We say that a decorated word w is *valid* if $\mu^w(x)$ is well-defined for every $x \in X$.

▶ **Proposition 24.** *For every functional extraction grammar G in CNF and for every document $\mathbf{d}$, if G is unambiguous then $\text{DECORGRMR}(G_{\mathbf{d}})$ is unambiguous.*

This allows us to establish the following connection between $\text{DECORGRMR}(G_{\mathbf{d}})$ and $[\![G]\!](\mathbf{d})$.

▶ **Lemma 25.** *For every functional unambiguous extraction grammar G in CNF and for every document $\mathbf{d}$, every decorated word produced by $\text{DECORGRMR}(G_{\mathbf{d}})$ is valid and*

$$[\![G]\!](\mathbf{d}) = \{\mu^w \mid S \Rightarrow^*_{\text{DECORGRMR}(G_{\mathbf{d}})} w\}.$$

Finally, combining Proposition 24 and Lemma 25 leads to the following direct corollary.

▶ **Corollary 26.** *For every functional unambiguous extraction grammar G in CNF and for every document $\mathbf{d}$, enumerating mappings in $[\![G]\!](\mathbf{d})$ can be done by enumerating parse-trees of decorated words in $\{w \mid S \Rightarrow^*_{\text{DECORGRMR}(G_\mathbf{d})} w\}$.*

To summarize the complexity of constructing $\text{DECORGRMR}(G_\mathbf{d})$ we have:

▶ **Proposition 27.** *For every functional unambiguous extraction grammar G in CNF and for every document $\mathbf{d}$, $\text{DECORGRMR}(G_\mathbf{d})$ can be constructed in $O(|G_\mathbf{d}|\, 5^{2k}) = O(|\mathbf{d}|^3 |G|\, 5^{2k})$ where k is the number of variables associated with G.*

5 Enumeration Algorithm

Our enumeration algorithm builds recursively the parse-trees of the decorated grammar $\text{DECORGRMR}(G_\mathbf{d})$. Before presenting it, we discuss some of the main ideas that allow us to obtain a constant delay between every two consecutive outputs.

5.1 Stable non-terminals

The non-terminals of $\text{DECORGRMR}(G_\mathbf{d})$ are decorated with superscripts and subscripts that give extra information that can be exploited in the process of the derivation.

▶ **Example 28.** Figure 10 presents a partial parse-tree (without the leaves and the first production) for a decorated word in $\text{DECORGRMR}(G_\mathbf{d})$. Notice that the variable operations that appear in the subtrees rooted in the non-terminal $C_{1,3}^{\vdash_x \vdash_y, \dashv_y}$ are only those indicated in its superscript. That is, there are no variable operations that occur between positions $1, 2$, and no such between positions $2, 3$. ⌟

Motivated by this, we say that a non-terminal $A_{i,j}^{\mathbf{x},\mathbf{y}}$ of $\text{DECORGRMR}(G_\mathbf{d})$ is *stable* if $\mathbf{x}_A = \mathbf{x} \cup \mathbf{y}$.

▶ **Lemma 29.** *For every functional extraction grammar G in CNF and for every document $\mathbf{d}$, the set of stable non-terminals of $\text{DECORGRMR}(G_\mathbf{d})$ is computable in $O(|G_\mathbf{d}| 5^{2k})$ where k is the number of variables G is associated with.*

Therefore, while constructing the parse-trees of $\text{DECORGRMR}(G_\mathbf{d})$ whenever we reach a stable non-terminal we can stop since its subtree does not affect the mapping.

5.2 The Jump Function

If G is associated with k variables, there are exactly $2k$ variable operations in each ref-word produced by G. Hence, we can bound the number of non-stable non-terminals in a parse-tree of $\text{DECORGRMR}(G_\mathbf{d})$. Nevertheless, the depth of a non-stable non-terminal can be linear in $|\mathbf{d}|$ as the following example suggests.

▶ **Example 30.** Consider the non-stable non-terminal $F_{n-1,n}^{\emptyset,\emptyset}$ in the partial parse-tree in Figure 9 of the decorated word $(\emptyset, 1, \emptyset) \cdots (\emptyset, n-1, \vdash_x \dashv_x)(\emptyset, n, \emptyset)$. Observe that the depth of this non-terminal is linear in n. ⌟

Since we want the delay of our algorithm to be independent of $|\mathbf{d}|$, we skip parts of the parse-tree in which no variable operation occurs. This idea somewhat resembles an idea that was implemented by Amarilli et al. [2] in their constant delay enumeration algorithm for regular spanners represented as vset-automata. There, they defined a function that "jumps"

from one state to the other if the path from the former to the latter does not contain any variable operation. We extend this idea to extraction grammars by defining the notion of skippable productions. Intuitively, when we focus on a non-terminal in a parse-tree, the corresponding mapping is affected by either the left subtree of this non-terminal, or by its right subtree, or by the production applied on the non-terminal itself (or by any combination of the above). If the mapping is affected exclusively by the left (right, respectively) subtree then we can skip the production and move to check the left (right, respectively) subtree, and do so recursively until we reach a production for which this is no longer the case.

Formally, a *skippable* production rule is of the form $A^{\mathbf{x},\mathbf{y}}_{i,j} \to B^{\mathbf{x},\mathbf{z}}_{i,i'} C^{\mathbf{z}',\mathbf{y}}_{i'+1,j}$ where (a) $A^{\mathbf{x},\mathbf{y}}_{i,j}$ is non-stable, (b) $\mathbf{z} = \mathbf{z}' = \emptyset$, and (c) exactly one of $B^{\mathbf{x},\mathbf{z}}_{i,i'}, C^{\mathbf{z}',\mathbf{y}}_{i'+1,j}$ is stable. Intuitively, (a) assures that the parse-tree rooted in $A^{\mathbf{x},\mathbf{y}}_{i,j}$ affects the mapping, (b) assures that the production applied on $A^{\mathbf{x},\mathbf{y}}_{i,j}$ does not affect the mapping and (c) assures that exactly one subtree of $A^{\mathbf{x},\mathbf{y}}_{i,j}$ (either the one rooted at $B^{\mathbf{x},\mathbf{z}}_{i,i'}$ if $C^{\mathbf{z}',\mathbf{y}}_{i'+1,j}$ is stable, or the one rooted at $C^{\mathbf{z}',\mathbf{y}}_{i'+1,j}$ if $B^{\mathbf{x},\mathbf{z}}_{i,i'}$ is stable) affects the mapping. We then say that a skippable production rule ρ *follows* a skippable production rule ρ' if the non-stable non-terminal in the right-hand side of ρ' is the non-terminal in the left-hand side of ρ. The function $\textsc{jump}\colon V^{\text{DEC}} \to 2^{V^{\text{DEC}}}$ is defined by $B \in \textsc{jump}(A^{\mathbf{x},\mathbf{y}}_{i,j})$ if there is a sequence of skippable production rules $\rho_1, \cdots, \rho_m$ such that:

- ρ_ι follows $\rho_{\iota-1}$ for every ι,
- the left-hand side of ρ_1 is $A^{\mathbf{x},\mathbf{y}}_{i,j}$,
- the non-stable non-terminal in the right-hand side of ρ_m is B,
- there is a production rule that is not skippable whose left-hand side is B.

▶ **Example 31.** In the decorated grammar whose (one of its) parse-tree appears in Figure 9 it holds that $F^{\emptyset,\emptyset}_{n-1,n} \in \textsc{jump}(S^{\emptyset,\emptyset}_{1,n})$. ⌟

The acyclic nature of the decorated grammar (that is, the fact that a non-terminal cannot be produced from itself) enables us to obtain the following upper bound for the computation of the JUMP function.

▶ **Lemma 32.** *For every functional unambiguous extraction grammar G in CNF and for every document $\mathbf{d}$, the* JUMP *function is computable in $O(|\mathbf{d}|^5 3^{4k} |G|^2)$ where k is the number of variables G is associated with.*

Lemmas 29 and 32 imply that we can find the non-stable non-terminals as well as compute the JUMP function as part of the quintic preprocessing. It is important to note that if we can reduce the complexity of computing the JUMP function to cubic then we can reduce the whole preprocessing time to cubic.

5.3 The Algorithm

Our main enumeration algorithm is presented in Algorithm 1 and outputs $(X, \mathbf{d})$-mappings μ represented as sets of pairs $(\vdash_x, i), (\dashv_x, j)$ whenever $\mu(x) = [i, j\rangle$. The procedure APPLYPROD is called with a non-terminal $A^{\mathbf{x},\mathbf{y}}_{i,j}$ that is (a) non-stable and (b) appears at the left-hand side of at least one rule that is not skippable. The iterator APPLYPROD outputs with constant delay all those pairs (β, map) for which there exists a not skippable rule of the form $A^{\mathbf{x},\mathbf{y}}_{i,j} \to B^{\mathbf{x},\mathbf{z}}_{i,i'} C^{\mathbf{z}',\mathbf{y}}_{i'+1,j}$ such that the following hold:

- $\mathsf{map} = \{(\tau, i'+1) \mid \tau \in \mathbf{z} \cup \mathbf{z}'\}$, and
- β is the concatenation of the non-stable terminals amongst $B^{\mathbf{x},\mathbf{z}}_{i,i'}$ and $C^{\mathbf{z}',\mathbf{y}}_{i'+1,j}$.

Notice that since $A^{\mathbf{x},\mathbf{y}}_{i,j}$ is non-stable and since $A^{\mathbf{x},\mathbf{y}}_{i,j} \to B^{\mathbf{x},\mathbf{z}}_{i,i'} C^{\mathbf{z}',\mathbf{y}}_{i'+1,j}$ is not skippable, it holds that either $\mathbf{z} \neq \emptyset$ or $\mathbf{z}' \neq \emptyset$ (or both). Thus, the returned map is not empty which implies

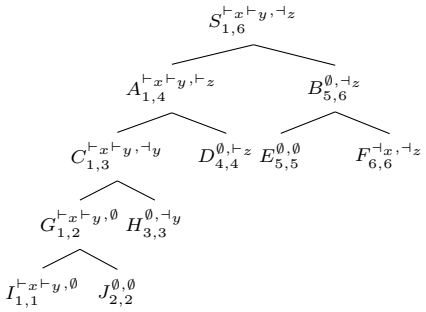

Figure 10 DECORGRMR($G_{\mathbf{d}}$) Parse tree.

Algorithm 1 Main enumeration algorithm.

procedure ENUMERATE(α, map)
if $\alpha = \epsilon$ **then**
  output map
denote α *by* $A \cdot \alpha'$;
foreach $B \in$ JUMP(A) **do**
  foreach $(\beta, \mathsf{map}') \in$ APPLYPROD(B) **do**
    ENUMERATE($\beta \cdot \alpha', \mathsf{map} \cup \mathsf{map}'$);

that every call to this procedure adds information on the mapping, and thus the number of calls is bounded. Notice also that β is the concatenation of the non-terminals among $B^{\mathbf{x},\mathbf{z}}_{i,i'}$ and $C^{\mathbf{z}',\mathbf{y}}_{i'+1,j}$ that affect the mapping.

▶ **Example 33.** The procedure APPLYPROD applied on $S^{\vdash_x \vdash_y, \dashv_z}_{1,6}$ from Figure 10 adds the pair $(\vdash_z, 5)$ to map; When applied on $A^{\vdash_x \vdash_y, \vdash_z}_{1,4}$, it adds the pair $(\dashv_y, 4)$ to map; When applied on $B^{\emptyset, \dashv_z}_{5,6}$, it adds the pair $(\dashv_x, 6)$ to map. ⌟

The recursive procedure ENUMERATE outputs the mapping as a set of pairs of the form (γ, i) with $\gamma \in \Gamma_X$ a variable operation and $1 \leq i \leq n$. The main enumeration algorithm calls the recursive procedure ENUMERATE with pairs $(S^{\mathbf{x},\mathbf{y}}_{1,n}, \mathsf{map})$ where $S^{\mathbf{x},\mathbf{y}}_{1,n}$ is a non-terminal in DECORGRMR($G_{\mathbf{d}}$), and map is the set containing pairs $(\tau, 1)$ for any $\tau \in \mathbf{x}$, and $(\tau, n+1)$ for any $\tau \in \mathbf{y}$. The recursive procedure ENUMERATE gets a pair (α, map) as input where α is a (possibly empty) sequence of non-stable non-terminals and map is a set of pairs of the above form. It recursively constructs an output mapping by applying derivations on the non-stable non-terminals (by calling APPLYPROD) while skipping the skippable productions (by using JUMP). We assume that ENUMERATE has $O(1)$ access to everything computed in the preprocessing stage, that is, the grammar DECORGRMR($G_{\mathbf{d}}$), the JUMP function, and the sets of stable and non-stable non-terminals.

▶ **Theorem 34.** *For every functional unambiguous extraction grammar G in CNF and for every document $\mathbf{d}$, the main enumerating algorithm described above enumerates the mappings in $[\![G]\!](\mathbf{d})$ (without repetitions) with delay of $O(k)$ between each two consecutive mappings where k is the number of variables G is associated with.*

Had G been ambiguous, the complexity guarantees on the delay would not have held.

Finally, we remark that the proof of Theorem 17 follows from Corollary 25, Proposition 27, Lemma 29, Lemma 32, and Theorem 34.

6 Conclusion

In this paper we propose a new grammar-based language for document spanners, namely extraction grammars. We compare the expressiveness of context-free spanners with previously studied classes of spanners and present a pushdown model for these spanners. We present an enumeration algorithm for unambiguous grammars that outputs results with a constant delay after quintic preprocessing in data complexity. We conclude by suggesting several future research directions.

To reach a full understanding of the expressiveness of context-free spanners, one should characterize the string relations that can be expressed with context-free spanners. This can be done by understanding the expressiveness of context-free grammars enriched with string equality selection. We note that there are some similarities between recursive Datalog over regex formulas [34] and extraction grammars. Yet, with the former we reach the full expressiveness of polynomial time spanners (data complexity) whereas with the latter we cannot express string equality. Understanding the connection between these two formalisms better can be a step in understanding the expressive power of extraction grammars.

Regarding our enumeration complexity, it might be possible to decrease the preprocessing complexity by using other techniques to compute the jump function. Another direction is to find restricted classes of extraction grammars that are more expressive than regular spanners yet allow linear time preprocessing (similarly to [2]).

It can be interesting to examine more carefully whether the techniques used here for enumerating the derivations can be applied also for enumerating queries on trees, or enumerating queries beyond MSO on strings. This connects to a recent line of work on efficient enumeration algorithms for monadic-second-order queries on trees [3]. Can our techniques be used to obtain efficient evaluation for more expressive queries?

References

1 Jitendra Ajmera, Hyung-Il Ahn, Meena Nagarajan, Ashish Verma, Danish Contractor, Stephen Dill, and Matthew Denesuk. A CRM system for social media: challenges and experiences. In *WWW*, pages 49–58. ACM, 2013.

2 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, pages 22:1–22:19, 2019.

3 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *PODS*, pages 89–103, 2019.

4 Edward Benson, Aria Haghighi, and Regina Barzilay. Event discovery in social media feeds. In *ACL*, pages 389–398. The Association for Computer Linguistics, 2011.

5 J. Richard Büchi and Lawrence H. Landweber. Definability in the monadic second-order theory of successor. *J. Symb. Log.*, 34(2):166–170, 1969.

6 Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, and Shivakumar Vaithyanathan. SystemT: An algebraic approach to declarative information extraction. In *ACL*, pages 128–137, 2010.

7 Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. Split-correctness in information extraction. In *PODS*, pages 149–163, 2019.

8 Johannes Doleschal, Benny Kimelfeld, Wim Martens, and Liat Peterfreund. Weight annotation in information extraction. In *ICDT*, volume 155 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

9 Pál Dömösi. Unusual algorithms for lexicographical enumeration. *Acta Cybernetica*, 14(3):461–468, 2000.

10 Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

11 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12, 2015.

12 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Declarative cleaning of inconsistencies in information extraction. *ACM Trans. Database Syst.*, 41(1):6:1–6:44, 2016.

13 Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Constant delay algorithms for regular document spanners. In *PODS*, pages 165–177. ACM, 2018.

14 Dominik D. Freydenberger. A logic for document spanners. In *ICDT*, volume 68 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

15 Dominik D. Freydenberger. A logic for document spanners. *Theory Comput. Syst.*, 63(7):1679–1754, 2019.

16 Dominik D. Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. In *ICDT*, volume 48 of *LIPIcs*, pages 17:1–17:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

17 Dominik D. Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. *Theory Comput. Syst.*, 62(4):854–898, 2018.

18 Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining extractions of regular expressions. In *PODS*, pages 137–149, 2018.

19 Dominik D. Freydenberger and Liat Peterfreund. Finite models and the theory of concatenation. *CoRR*, abs/1912.06110, 2019. URL: `http://arxiv.org/abs/1912.06110`.

20 Dominik D. Freydenberger and Sam M. Thompson. Dynamic complexity of document spanners. In *ICDT*, volume 155, pages 11:1–11:21, 2020.

21 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

22 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

23 Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

24 Donald E. Knuth. Correction: Semantics of context-free languages. *Mathematical Systems Theory*, 5(1):95–96, 1971.

25 Clemens Lautemann, Thomas Schwentick, and Denis Thérien. Logics for context-free languages. In *CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 1994.

26 Yaoyong Li, Kalina Bontcheva, and Hamish Cunningham. SVM based learning system for information extraction. In *Deterministic and Statistical Methods in Machine Learning, First International Workshop, Sheffield, UK, September 7-10, 2004, Revised Lectures*, pages 319–339, 2004.

27 Yunyao Li, Frederick Reiss, and Laura Chiticariu. SystemT: A declarative information extraction system. In *ACL*, pages 109–114. ACL, 2011.

28 Erkki Mäkinen. On lexicographic enumeration of regular and context-free languages. *Acta Cybernetica*, 13(1):55–61, 1997.

29 Francisco Maturana, Cristian Riveros, and Domagoj Vrgoč. Document spanners for extracting incomplete information: Expressiveness and complexity. In *PODS*, pages 125–136, 2018.

30 Andrea Moro, Marco Tettamanti, Daniela Perani, Caterina Donati, Stefano F Cappa, and Ferruccio Fazio. Syntax and the brain: disentangling grammar by selective anomalies. *Neuroimage*, 13(1):110–118, 2001.

31 Takashi Nagashima. A formal deductive system for CFG. *Hitotsubashi journal of arts and sciences*, 28(1):39–43, 1987.

32 Yoav Nahshon, Liat Peterfreund, and Stijn Vansummeren. Incorporating information extraction in the relational database model. In *WebDB*, page 6. ACM, 2016.

33 Liat Peterfreund, Dominik D. Freydenberger, Benny Kimelfeld, and Markus Kröll. Complexity bounds for relational algebra over document spanners. In *PODS*, pages 320–334. ACM, 2019.

34 Liat Peterfreund, Balder ten Cate, Ronald Fagin, and Benny Kimelfeld. Recursive programs for document spanners. In *ICDT*, volume 127 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

35 Christopher De Sa, Alexander Ratner, Christopher Ré, Jaeho Shin, Feiran Wang, Sen Wu, and Ce Zhang. Deepdive: Declarative knowledge base construction. *SIGMOD Record*, 45(1):60–67, 2016.

36 Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inf. Comput.*, 249:1–17, 2016.

37 Rania A Abul Seoud, Abou-Bakr M Youssef, and Yasser M Kadah. Extraction of protein interaction information from unstructured text using a link grammar parser. In *2007 International Conference on Computer Engineering & Systems*, pages 70–75. IEEE, 2007.

38 Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using Datalog with embedded extraction predicates. In *VLDB*, pages 1033–1044, 2007.

39 Jason M Smith and David Stotts. SPQR: Flexible automated design pattern extraction from source code. In *18th IEEE International Conference on Automated Software Engineering*, pages 215–224. IEEE, 2003.

40 Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975. doi:10.1016/S0022-0000(75)80046-8.

41 Huang Wen-Ji. Enumerating sentences of context free language based on first one in order. *Journal of Computer Research and Development*, 41(1):9–14, 2004.

42 Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898. ACM, 2012.

43 Hua Xu, Shane P. Stenner, Son Doan, Kevin B. Johnson, Lemuel R. Waitman, and Joshua C. Denny. MedEx: a medication information extraction system for clinical narratives. *JAMIA*, 17(1):19–24, 2010. doi:10.1197/jamia.M3378.

44 Akane Yakushiji, Yuka Tateisi, Yusuke Miyao, and Jun-ichi Tsujii. Event extraction from biomedical papers using a full parser. In *Biocomputing 2001*, pages 408–419. World Scientific, 2000.

45 Huaiyu Zhu, Sriram Raghavan, Shivakumar Vaithyanathan, and Alexander Löser. Navigating the intranet with high precision. In *WWW*, pages 491–500. ACM, 2007.