Attribute Grammars for Scalable Query Processing on XML Streams

Christoph Koch[1]* and Stefanie Scherzinger[2]

[1] LFCS, University of Edinburgh, Edinburgh EH9 3JZ, UK
`koch@dbai.tuwien.ac.at`
[2] Lehrstuhl für Dialogorientierte Systeme
FMI, Universität Passau, D-94030 Passau, Germany
`scherzin@fmi.uni-passau.de`

Abstract. We introduce the new notion of XML Stream Attribute Grammars (XSAGs). XSAGs are the first scalable query language for XML streams (running strictly in linear time with bounded memory consumption independent of the size of the stream) that allows for actual data transformations rather than just document filtering. XSAGs are also relatively easy to use for humans. Moreover, the XSAG formalism provides a strong intuition for which queries can or cannot be processed scalably on streams. We introduce XSAGs together with the necessary language-theoretic machinery, study their theoretical properties such as their expressiveness and complexity, and discuss their implementation.

1 Introduction

In recent years, XML has become a standard format for document exchange and now seems to develop into a preeminent representation language for streaming data as well. This development calls for flexible query languages for processing streams which support data transformations.

In [12, 15, 7], fragments of the standard XML Query language [16] are evaluated on XML streams. These fragments tend to support powerful data transformations, with the consequence that query processing neither scales in terms of runtime nor memory consumption. Indeed, in these works, memory buffers are required that can grow arbitrarily large, depending on the amount of data communicated via the stream.

This problem is due to the nature of XML Query, which renders it ill-suited for stream processing: Features such as nested for-loops with transitive paths (e.g., using the descendant axis), which may lead to a nonlinearly-sized output, and nonlocal computations such as joins and the reordering and sorting of data cannot be handled scalably on streams. In addition, the syntax of XML Query makes it difficult to tell for a user whether a query can – at least in principle – be evaluated scalably, in linear time using little memory.

* Work sponsored by Erwin Schrödinger Grant J2169 of the Austrian Research Fund (FWF).

Query languages that require unbounded memory buffers constitute a scalability issue on streams and are not in the spirit of the database community's quest for tailored formalisms that provide the appropriate tradeoffs between expressiveness and complexity for the data management challenge at hand.

XML streams by definition may be *very* long, or should even be assumed to be infinite. For query processing to be feasible on streams, there is a need for special-purpose query languages and evaluation algorithms which *scale to streams*, i.e.,

(a) which can be evaluated strictly in linear time in the size of the input,
(b) which work in streaming fashion, by one linear forward scan of the data, and
(c) for which, at any time during query evaluation, memory consumption is bounded[3].

Among the models of computation that allow for better control of complexity than languages such as XML Query, there are various forms of automata/transducers and certain attribute grammars. The former are, however, unsuitable as query languages used by humans because their specifications tend to be large, technical, and hard to read. The latter approach is developed in the present paper.

Attribute Grammars for Stream Processing

In this work, we develop and investigate a formalism for processing XML streams called *XML Stream Attribute Grammars* (XSAGs), a new class of attribute grammars specifically designed for scalable XML stream processing. XSAGs can be evaluated strictly in linear time in streaming fashion, consuming only a stack of memory bounded by the depth of the XML tree being streamed. Thus, XSAGs satisfy our desiderata (a) through (c).

XSAGs are based on *extended regular tree grammars*, i.e. regular tree grammars in which the right-hand sides of productions may contain regular expressions, allowing to specify nodes in the parse tree that have an unbounded number of children. Extended regular tree grammars are thus well-suited for specifying classes of unranked trees denoting XML documents. We assume that extended regular tree grammars are often available for XML streams in the dialect of *Document Type Definitions* (DTDs). This adds to the relevance of the present formalism.

An XSAG is obtained by annotating a given extended regular tree grammar with attribution functions that describe the output to be produced from the input stream. In the tradition of L-attributed grammars [1], XSAGs are assumed

[3] Note that a stack of memory proportional to the maximum depth of the XML tree is necessary for even the most basic sequential navigation and parsing tasks (see e.g. [8, 10]). To be precise, in (c) we thus call for memory consumption that is bounded w.r.t. the length of the stream but not the depth of the XML tree (an indication of its structural complexity). XML trees tend to be very shallow but wide, therefore such a stack is not considered a bottleneck to scalability.

to perform a single scan of the XML stream, which amounts to effecting a single depth-first left-to-right traversal of the document tree. Also in the tradition of L-attributed grammars, right-hand sides of productions can be annotated with *two* attribution functions, one – placed at the beginning of the right-hand side – that is executed upon reaching the opening tag of a node in the XML document (or equivalently, when descending into a subtree), and the second – placed at the end of the right-hand side – which is executed when the corresponding closing tag is reached (or, equivalently, when returning from the depth-first-traversal of the subtree).

Example 1. Consider the extended regular tree grammar $G = (Nt, T, P, bib)$ with nonterminals

$$Nt = \{bib, book, article, title, author\},$$

start nonterminal *bib*, terminals

$$T = \{\mathrm{bib}, \mathrm{book}, \mathrm{article}, \mathrm{title}, \mathrm{author}, PCDATA\},$$

and the productions P

$$\begin{aligned} bib &::= \mathrm{bib}((book \cup article)^*) \\ book &::= \mathrm{book}(title.author.author^*) \\ article &::= \mathrm{article}(title.author.author^*) \\ title &::= \mathrm{title}(PCDATA) \\ author &::= \mathrm{author}(PCDATA) \end{aligned}$$

which defines an XML bibliography database.

By changing the first production to

$$bib ::= \{ECHO\}\ \mathrm{bib}((book \cup article)^*)$$

we obtain an XSAG that simply echoes the input stream. Indeed, the start production matches the root node of the document, and *ECHO* writes the entire subtree of the current node to the output as XML.

If we are instead only interested in books arriving on the stream, we can use the XSAG obtained by changing the *bib* and *book* productions to

$$\begin{aligned} bib &::= \{\text{print } \langle\text{books}\rangle\}\ \mathrm{bib}\big((book \cup article)^*\big)\ \{\text{print } \langle/\text{books}\rangle\} \\ book &::= \{ECHO\}\ \mathrm{book}\big(title.author.author^*\big) \end{aligned}$$

Here we apply *ECHO* to the "book" subtrees, but not to articles. We output the opening and closing tags of the root node explicitly, and label the root node of the output produced by this XSAG "books", rather than "bib". □

Based on the basic notion of XSAGs (bXSAGs) exemplified so far, we introduce the *easy XSAGs* (yXSAGs). These allow to annotate the regular expressions inside productions with attribution functions as well, which adds to the flexibility of the formalism.

Example 2. The yXSAG with production

$$\begin{aligned} \mathit{article} ::= {} & \{\text{print } \langle\text{article}\rangle\} \\ & \text{article}\Big((\{\mathit{ECHO}\}\,\mathit{title}). \\ & \qquad (\{\text{print } \langle\text{authors}\rangle;\ \mathit{ECHO}\} \\ & \qquad\quad (\mathit{author}.\mathit{author}^*) \\ & \qquad \{\text{print } \langle/\text{authors}\rangle\})\Big) \\ & \{\text{print } \langle/\text{article}\rangle\} \end{aligned}$$

outputs articles basically as they arrive on the stream, but groups the authors of each article under a common *authors* node. Here, the second appearance of *ECHO* in the production applies to the tree region matched by the regular expression $\mathit{author}.\mathit{author}^*$, i.e., to the subtrees below *article* nodes that are rooted by *author* nodes. □

Being attribute grammars, XSAGs of course support attributes. In order to assure scalability in the strictest sense, we require that attributes range over a finite domain fixed with the XSAG.

Example 3. Assume that our grammar assures that books in addition have a *year* of publication as a first child:

$$\begin{aligned} \mathit{book} &::= \text{book}\big(\mathit{year}.\mathit{title}.\mathit{author}.\mathit{author}^*\big) \\ \mathit{year} &::= \text{year}(\mathit{PCDATA}) \end{aligned}$$

Then, for instance, the yXSAG

$$\begin{aligned} \mathit{bib} &::= \{\text{print } \langle\text{books}\rangle\}\ \text{bib}\big((\mathit{book} \cup \mathit{article})^*\big)\ \{\text{print } \langle/\text{books}\rangle\} \\ \mathit{book} &::= \text{book}\Big((\{\mathit{MATCH\_CHILDREN}(\text{“2003”}, c)\}\ \mathit{year}). \\ &\qquad (\{\text{if } c = \mathit{true} \text{ then} \\ &\qquad\quad \text{begin} \\ &\qquad\qquad \text{print } \langle\text{book}\rangle; \mathit{ECHO} \\ &\qquad\quad \text{end}\} \\ &\qquad\quad (\mathit{title}.\mathit{author}.\mathit{author}^*) \\ &\qquad \{\text{if } c = \mathit{true} \text{ then print } \langle/\text{book}\rangle\})\Big) \end{aligned}$$

outputs books whose year of publication is 2003 with their title and author children, but without the years.

For a given *year* node, *MATCH_CHILDREN* sets boolean-valued condition attribute[4] c to true if the character text encountered while scanning its children from left to right matches “2003”. Otherwise, c is set to false.

This condition attribute is passed on through the document tree during its traversal. The regular expression $\mathit{title}.\mathit{author}.\mathit{author}^*$ describes a tree region

[4] Note that in the technical sections of this paper, we will use a somewhat more explicit syntax when employing attributes (see e.g. Example 14).

among the children of a book node. Just before we first enter this tree region, we examine the value of c. If c is true (and the current book has been published in 2003), we print opening tag $\langle$book$\rangle$ and echo the tree region to the output. On leaving the tree region, we further output closing tag $\langle$/book$\rangle$ if c is set to true.

Note that this yXSAG is equivalent to XML Query

```
⟨books⟩
{ for $x in //book where $x/year = 2003
  return ⟨book⟩ {$x/title} {$x/author} ⟨/book⟩ }
⟨/books⟩
```

on documents conforming to our grammar. □

Attribute grammars are well known in the field of compilers and have recently been revisited in the context of XML, for instance for grammar-directed XML publishing [4, 3]. Some of their theory relevant in the context of structured documents has been studied in [14, 13].

Our emphasis is on designing a *practical* formalism for query processing that is *relatively easy to use*. Attribute grammars are widely agreed to carry a strong intuition for specifying syntax-directed translations. In our setting, they provide a metaphor for strictly linear-time one-pass XML transformations that can be grasped very intuitively. This renders it relatively easy for a user to recognize or design queries which can be executed (scalably) on a stream, even if this intuition is paid for by our formalism being more operational than languages such as XML Query. While ease of use cannot be conclusively asserted based only on our own observations and the examples we provide, alternative formalisms such as deterministic pushdown transducers (DPDTs) are unsuitable as query languages to be used by humans; Query processors for languages such as XML Query, on the other hand, do not scale to streams. We can therefore argue that XSAGs achieve our goal of relative ease of use. Already bXSAGs are much more practical than DPDTs. yXSAGs permit elegant nested attributions, which, as can be seen in Example 2 and others throughout the paper, allow to specify many interesting data transformations conveniently.

Contributions

The technical contributions of this paper are as follows.

– We examine the framework of extended regular tree grammars and craft grammar classes appropriate for attribution and stream processing.
– In order to be able to characterize yXSAGs properly, we develop the new notion of *strongly one-unambiguous regular expressions*, as well as some of its theory. These expressions are precisely those for which the *parse tree* of a word (analogously to the derivation tree of a grammar) can be unambiguously constructed *online*, with just a one-symbol lookahead, while processing the stream.

yXSAGs allow for attributions to be nested inside regular expressions by only permitting strongly one-unambiguous regular expressions in the right-hand sides of productions.

– We introduce and formally define our two notions of attribute grammars, bXSAGs and yXSAGs, and compare them with respect to usability.
– We introduce *XML-DPDT*s, deterministic push-down transducers with a natural stack discipline that assures that the size of the stack remains strictly proportional to the depth of the XML tree and which can only accept well-formed XML documents. *XML-DPDT*s in a sense capture the intuition of scalable XML stream processing and serve as an expressiveness yardstick for XSAGs.
– We show that both bXSAGs and yXSAGs are precisely as expressive as *XML-DPDT*s. XSAGs provide the same quasi-optimal trade-off between expressiveness and evaluation cost as do *XML-DPDT*s.
– Finally, we study the complexity of evaluating XSAGs, and their implementation.

The *structure* of this paper basically follows the order of contributions described above.

2 Regular Tree Grammars

Throughout this paper, we assume that regular expressions are constructed from atomic symbols using concatenation ., union $\cup$, and the Kleene star * (but not ϵ, $^+$, or ?).

Let *Tag* be a set of node labels ("tags") and let *Char* be a set of characters distinct from the tags. An *extended regular tree grammar* is a grammar $G = (Nt, T, P, s)$ where

1. Nt is a set of nonterminals,
2. $T = Tag \cup Char$ is a set of terminals,
3. P is a set of productions $nt ::= t(\rho)$ where $nt \in Nt$, $t \in T$, ρ is either ϵ or a regular expression over alphabet Nt, and if $t \in Char$, then $\rho = \epsilon$, and
4. $s \in Nt$ is the start symbol.

We assume the standard meaning of grammars and their derivations for which we refer to [9] for basic and to [11] for extended grammars.

Extended regular tree grammars (and DTDs, which are a dialect of extended regular tree grammars) are a convenient way to specify a class of unranked labeled trees and thus XML documents.

Let $Char = \{c_1, \ldots, c_n\}$. As a shortcut, we define the regular expression macro

$$PCDATA := (c_1' \cup \cdots \cup c_n')^*$$

which, using new nonterminals $c_1', \ldots, c_n'$ and productions $c_i' ::= c_i(\epsilon)$ for each $1 \leq i \leq n$, can be used just like a terminal in right-hand sides of grammar productions. PCDATA accepts all character strings. As a further notational convenience, we will allow ourselves to be somewhat imprecise below whenever we only use PCDATA, but no characters, in our grammar definitions. Then we will list "PCDATA" among the terminals and will keep the nonterminals $c'_1, \ldots, c'_n$ unmentioned.

Given a nonterminal *nt*, let $\theta(nt)$ denote the set of terminals t such that the grammar contains a production $nt ::= t(\rho_{nt,t})$. Given a regular expression ρ, let $\tau(\rho)$ denote the regular expression in which each nonterminal *nt* in ρ is replaced by the union of terminals $\bigcup \theta(nt)$.

Each extended regular tree grammar can be alternatively considered as an extended context-free word grammar (CFG), which is obtained by simply rewriting each right-hand side $tag(\rho)$ into $\langle tag\rangle\rho\langle/tag\rangle$. *Deterministic* context-free languages are precisely those recognizable by the deterministic pushdown automata (DPDA, see e.g. [9]). DPDAs run comfortably on streams requiring only a stack of memory bounded by the depth of the input tree. Using automata, we can thus scalably recognize the deterministic context-free languages.

The problem of processing an (extended) attribute grammar on a document requires an additional, different restriction on the grammar, besides determinism, to allow for deterministic computation. We need to be able to unambiguously refer to the atomic *symbols* in the regular expressions to be able to access or assign attributes. In attribute grammars, a straightforward solution [13] is to require regular expressions to be *unambiguous*.

Example 4. Consider the grammar

$$\begin{aligned} bib &::= \mathrm{bib}((book_1 \cup book_2)^*) \\ book_1 &::= \mathrm{book}(\rho) \\ book_2 &::= \mathrm{book}(\rho) \end{aligned}$$

where ρ is some regular expression. The regular expression $(book_1 \cup book_2)^*$ is unambiguous, but

$$\tau(book_1 \cup book_2)^* = (\mathrm{book} \cup \mathrm{book})^*$$

is not. Therefore, when considering the tags of the children of the root "bib" node, we cannot determine where to apply which of the two "book" productions with their possibly different attributions. □

On streams, we cannot look ahead beyond a nonterminal (which may stand for a large subtree that we do not want to buffer) when parsing the input. Thus, we will assume the stronger notion of *one-unambiguity* for regular expressions $\tau(\rho)$. That is, we will require that $\tau(\rho)$ can be unambiguously parsed with just one symbol of lookahead.

2.1 One-unambiguity and TDLL(1)

By a *marking* of a regular expression ρ over alphabet Σ, we denote a regular expression ρ' such that each occurrence of an atomic symbol in ρ is replaced by the symbol with its position among the atomic symbols of ρ added as subscript. That is, the i-th occurrence of a symbol $a \in \Sigma$ in ρ is replaced by a_i. For instance, the marking of $(a \cup b)^*.a.a^*$ is $(a_1 \cup b_2)^*.a_3.a_4^*$. The reverse of a marking (indicated by $^\#$) is obtained by dropping the subscripts.

Let ρ be a regular expression, ρ' its marking, and Σ' the marked alphabet used by ρ'. Then, ρ is called *one-ambiguous* iff there are words u, v, w over Σ' and symbols $x, y \in \Sigma'$ such that $uxv, uyw \in L(\rho')$, $x \neq y$, and $x^\# = y^\#$. A regular expression is called *one-unambiguous* if it is not one-ambiguous.

Example 5. Consider the regular expression $\rho = a^*.a$ and its marking $\rho' = a_1^*.a_2$. Let $u = a_1$, $x = a_2$, $y = a_1$, $v = \epsilon$, and $w = a_2$. Clearly, $uxv = a_1.a_2$ and $uyw = a_1.a_1.a_2$ are both words of $L(\rho')$, thus ρ is one-ambiguous. On the other hand, the equivalent regular expression $a.a^*$ is one-unambiguous. □

Definition 1 ([11]). A *TDLL(1)-Grammar* is an extended regular tree grammar where $\tau(s)$ is unambiguous[5] and in which for each regular expression ρ in the right-hand side of a production, $\tau(\rho)$ is one-unambiguous. □

Example 6. The grammar of Example 1 is TDLL(1). On the other hand, since the grammar of Example 4 contains a regular expression ρ such that $\tau(\rho)$ is not even unambiguous, that grammar is not TDLL(1). □

Remark 1 (One-unambiguity in DTDs). For XML elements that exclusively have elements as children (but no character data), the W3C recommendation [6] explicitly requires a one-unambiguous *content model* (that is, right-hand side regular expression) in order to assure compatibility with SGML.

Productions defining elements also containing character data must be constructed according to either the pattern

$$nt_0 ::= (\text{PCDATA} \cup nt_1 \cup \cdots \cup nt_m)^*$$

or $nt_0 ::= \text{PCDATA}$ (where $nt_0, \ldots, nt_m$ are DTD element names, i.e., nonterminals). Clearly, all regular expressions such constructed are also one-unambiguous.

Thus, since DTDs also contain at most one production $nt ::= t(\rho)$ for each "element" t (and thus if ρ is one-unambiguous, $\tau(\rho)$ is as well), DTDs are TDLL(1) grammars (see also [11]). □

2.2 Strong One-unambiguity and STDLL(1)

One-unambiguity and TDLL(1) grammars allow us to use attributed regular tree grammars on XML streams. However, as we show below, the ability to use attribution functions inside the regular expressions at the right-hand sides of productions will allow us to write many practical queries in a much more user-friendly fashion. Our machinery for achieving this is the notion of *strongly one-unambiguous* regular expressions.

[5] Note that $\tau(s)$ is guaranteed to be a very simple form of regular expression, a disjunction of atomic symbols (so unambiguity implies one-unambiguity).

[6] Sections 3.2.1, 3.2.2, and Appendix E in [5]

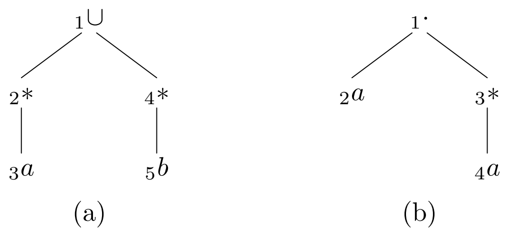

Fig. 1. Parse trees of regular expressions $a^* \cup b^*$ (a) and $a.a^*$ (b) with nodes annotated with markings.

Intuitively, by a *bracketing* of a regular expression ρ, we refer to a *marking of the nodes in the parse tree* of ρ using distinct indexes. We realize this by assigning the indexes by a depth-first left-to-right traversal of the parse tree, that is, in document order (see Figure 1 for two examples). The bracketing $\rho^{[]}$ is then obtained by inductively mapping each subexpression π of ρ with index i to $[_i.\pi.]_i$. Thus, a bracketing is a regular expression over the alphabet $\Sigma \cup \Gamma$, where $\Gamma = \{[_i,]_i \mid i \in \{1, 2, 3, \dots\}\}$. (We assume Σ and Γ disjoint.) For example, $[_1.\big(([_2.([_3.a.]_3)^*.]_2) \cup ([_4.([_5.b.]_5)^*.]_4)\big).]_1$ is the bracketing of $a^* \cup b^*$ and $[_1.\big(([_2.a.]_2).([_3.([_4.a.]_4)^*.]_3)\big).]_1$ is the bracketing of $a.a^*$.

Definition 2. Let ρ be a regular expression and $\rho^{[]}$ be its bracketing. A regular expression ρ is called *strongly one-unambiguous* iff there do not exist words u, v, w over $\Sigma \cup \Gamma$, words $\alpha \neq \beta$ over Γ, and a symbol $x \in \Sigma$ such that $u\alpha xv, u\beta xw \in L(\rho^{[]})$ or $u\alpha, u\beta \in L(\rho^{[]})$. $\square$

Example 7. The regular expression $a^* \cup b^*$ (see Figure 1 (a)) is one-unambiguous but not strongly one-unambiguous. Consider the bracketing $[_1.\big(([_2.([_3.a.]_3)^*.]_2) \cup ([_4.([_5.b.]_5)^*.]_4)\big).]_1$. The empty word can be matched in two alternative ways, namely as $u.\alpha = [_1.[_2.]_2.]_1$ and as $u.\beta = [_1.[_4.]_4.]_1$. The equivalent regular expression $a.a^* \cup b^*$ is strongly one-unambiguous. $\square$

Example 8. The regular expression $(a^*)^*$ with bracketing $[_1.([_2.([_3.a.]_3)^*.]_2)^*.]_1$ is not strongly one-unambiguous. For instance, for the word $a.a$, there are the bracketings $[_1.[_2.[_3.a.]_3.[_3.a.]_3.]_2.]_1$ and $[_1.[_2.[_3.a.]_3.]_2.[_2.[_3.a.]_3.]_2.]_1$ ($u = [_1.[_2.[_3.a$, $\alpha =]_3.[_3$, $\beta =]_3.]_2.[_2.[_3$, $x = a$, $v = w =]_3.]_2.]_1$). $\square$

Definition 3. An *STDLL(1) Grammar* is an extended regular tree grammar where $\tau(s)$ is unambiguous[7] and in which for each regular expression ρ in the right-hand side of a production, $\tau(\rho)$ is strongly one-unambiguous. $\square$

Obviously, all STDLL(1) grammars are also TDLL(1) grammars.

Remark 2. We suspect that most practical DTDs actually use only strongly one-unambiguous regular expressions in productions and are thus STDLL(1) grammars. Strong one-unambiguity is only a short way from one-unambiguity, and

[7] Obviously, the unambiguity of $\tau(s)$ implies its strong one-unambiguity.

many of the most widely used forms of regular expressions are actually strongly one-unambiguous (e.g., regular expressions of the form $(e_1 \cup \cdots \cup e_m)^*$, where $e_1, \ldots, e_m$ are element names). In particular, the syntactic restriction on mixed-content models mentioned in Remark 1 ensures that such regular expressions are guaranteed to be strongly one-unambiguous. $\square$

2.3 Parse Trees

Given a regular expression π over atomic symbols T, we obtain an equivalent (extended) *regular grammar* $G = (V, T, P, \underline{\pi})$ by recursively decomposing π into productions P,

$$\underline{\rho_1.\rho_2} ::= \underline{\rho_1}\ \underline{\rho_2} \qquad \underline{\rho_1 \cup \rho_2} ::= \underline{\rho_1} \mid \underline{\rho_2} \qquad \underline{\rho^*} ::= \underline{\rho}^*$$

for regular expressions ρ, ρ_1, ρ_2. Here, by $\underline{\rho}$, we refer to a symbol of $V \cup T$ rather than a regular expression. The nonterminals V consist precisely of the symbols $\underline{\rho}$ such that ρ is nonatomic. (For the special case that π is atomic, P is empty and the start symbol $\underline{\pi}$ is allowed to be a terminal.[8])

Regular grammars provide us with a natural way of assigning parse trees to words. For simplicity, we will use interior nodes of the forms "$\cup$", ".", and "$*$", rather than nonterminals $\underline{\rho_1 \cup \rho_2}$, $\underline{\rho_1.\rho_2}$, and $\underline{\rho^*}$, as illustrated in Figure 2 (b).

For TDLL(1) grammars, the parse trees are simply the usual document trees associated to XML documents. However, for STDLL(1) grammars, the parse trees incorporate the parse trees of the regular expressions occurring in productions on the input document. (If a regular expression contains a nonterminal *nt*, we assign it the terminal t as its unique child in the parse tree if the production used to rewrite *nt* is of the form $nt ::= t(\rho)$, for some ρ.)

We illustrate the two forms of parse trees by an example.

Example 9. The extended regular tree grammar G of Example 1 is an STDLL(1) grammar. Consider the XML document

⟨bib⟩ ⟨article⟩ ⟨title/⟩ ⟨author/⟩ ⟨author/⟩ ⟨author/⟩ ⟨/article⟩ ⟨/bib⟩

Using G as a TDLL(1) grammar, the document parses into the tree depicted in Figure 2 (a). To be precise, we assume here that the operation "." associates to the right and that the production

$$\mathit{article} ::= \mathrm{article}\big(\mathit{title}.\mathit{author}.\mathit{author}^*\big)$$

of G is thus equivalent to

$$\mathit{article} ::= \mathrm{article}\big(\mathit{title}.(\mathit{author}.\mathit{author}^*)\big).$$

The parse tree for G viewed as an STDLL(1) grammar is shown in Figure 2 (b). $\square$

[8] This is somewhat nonstandard but fits ours needs.

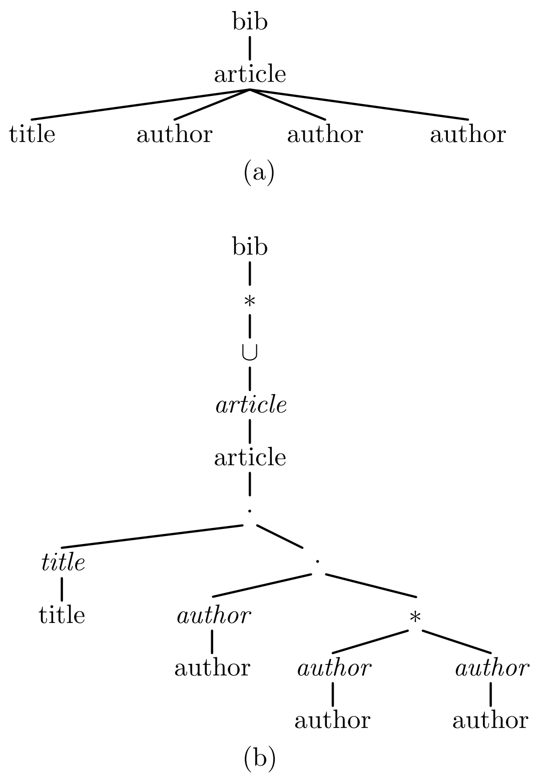

Fig. 2. Parse trees of Example 9.

3 XML Stream Attribute Grammars

We are now in the position to define our main attribute grammar formalism, XML Stream Attribute Grammars (XSAGs).

3.1 XSAGs in the Abstract

Definition 4 (Syntax). Let $Att = \{a_1, \ldots, a_k\}$ be a set of attributes and Dom be a finite set of *domain values* (or, to invoke an alternative intuition, of *states*).

Let $F_{\$[}$ denote the class of partial functions

$$f_{\$[} : Dom^k \to Dom^k \times string,$$

called first-visit attribution functions, and let $F_{\$]}$ denote the partial functions

$$f_{\$]} : Dom^{2k} \to Dom^k \times string,$$

called second-visit attribution functions. (We will introduce a language for implementing these partial functions in Section 3.2).

A basic XSAG (bXSAG) is an attributed extended regular tree grammar $G = (\mathit{Nt}, T, P, s)$ with nonterminals Nt, terminals $T = \mathit{Tag} \cup \mathit{Char}$, and productions in P that each are of one of the four forms

$$\mathit{nt} ::= t(\rho) \quad \mathit{nt} ::= \{f_{\$[}\}\ t(\rho)$$

$$\mathit{nt} ::= t(\rho)\ \{f_{\$]}\} \quad \mathit{nt} ::= \{f_{\$[}\}\ t(\rho)\ \{f_{\$]}\}$$

where $\mathit{nt} \in \mathit{Nt}$, $t \in T$, $f_{\$[} \in F_{\$[}$, $f_{\$]} \in F_{\$]}$, and ρ is either ϵ or a regular expression over Nt such that $\tau(\rho)$ is one-unambiguous.

The abstract syntax of an *attributed regular expression* over symbols Σ can be specified by the EBNF

$$\begin{aligned} \mathit{aregex} &::= (\text{``\{''}\ F_{\$[}\ \text{``\}''})?\ \ \mathit{aregex}_0\ \ (\text{``\{''}\ F_{\$]}\ \text{``\}''})? \\ \mathit{aregex}_0 &::= \Sigma \mid \mathit{aregex}\ \text{``.''}\ \mathit{aregex} \mid \\ &\qquad \mathit{aregex}\ \text{``}\cup\text{''}\ \mathit{aregex} \mid \mathit{aregex}\ \text{``*''} \end{aligned}$$

An easy XSAG (yXSAG) is an attributed extended regular tree grammar $G = (\mathit{Nt}, T, P, s)$ where each production in P is of one of the four forms

$$\mathit{nt} ::= t(\alpha) \quad \mathit{nt} ::= \{f_{\$[}\}\ t(\alpha)$$

$$\mathit{nt} ::= t(\alpha)\ \{f_{\$]}\} \quad \mathit{nt} ::= \{f_{\$[}\}\ t(\alpha)\ \{f_{\$]}\}$$

where α is either ϵ or an attributed regular expression over symbols Nt such that for the regular expression ρ obtained from α by removing the attributions (enclosed in curly braces), $\tau(\rho)$ is *strongly* one-unambiguous. □

The main purpose of the grammar component of an XSAG is to unambiguously map XML documents to parse trees.[9] The only differences between bXSAGs and yXSAGs are that the former use TDLL(1) grammars while the latter use STDLL(1) grammars, and that in yXSAGs, right-hand side regular expressions may be attributed[10]

It remains to specify how our attribute grammars are evaluated on such parse trees. Obviously, there is a natural method of assigning the attribution functions from $F_{\$[}$ and $F_{\$]}$ to nodes of the parse tree. For the sake of simplicity, we assume that each node v of the parse tree is assigned two functions $f^v_{\$[} \in F_{\$[}$ and $f^v_{\$]} \in F_{\$]}$ through the attribute grammar definition. Where this has not been the case, the defaults are

$$f^d_{\$[} : \langle x_1, \dots, x_k \rangle \mapsto \langle x_1, \dots, x_k, \epsilon \rangle$$

and

$$f^d_{\$]} : \langle x_1, \dots, x_k, x_{k+1}, \dots, x_{2k} \rangle \mapsto \langle x_{k+1}, \dots, x_{2k}, \epsilon \rangle.$$

[9] However, for evaluating XSAGs it will not at any time be necessary to maintain entire parse trees in memory.

[10] And indeed, precisely the restriction to STDLL(1) grammars makes it safe to attribute regular expressions in XSAGs.

bXSAGs and yXSAGs are (attributed) *extended* regular tree grammars. For such grammars, nodes of the parse tree may have an arbitrary number of children. When dealing with streams, we generally cannot store the attribute values of all these children in memory. We thus have to introduce special restrictions to be able to deal with streams on one hand and at the same time assure ease of use and expressiveness to cover practical queries on the other.

We define XSAGs as L-attributed grammars, i.e., attribute grammars whose attributes are evaluated by a single depth-first left-to-right traversal of the document tree. Each node v of the parse tree is visited twice (the visits are referred to by \$[and \$]), first from the previous sibling or the parent of v (if v has no previous sibling) and a second time on returning from the rightmost child of v. To provide a clear picture of the necessary computations, we distinguish the states of attribute values *before* (using the subscript "in") and *after* (using the subscript "out") the application of an attribution function.

Definition 5 (Semantics). Let $q_\perp \in Dom$ be a special "uninitialized" value. We evaluate an XSAG on a parse tree T by a depth-first traversal of T in which we compute, for each attribute $a_i \in Att$ and each node v of T, the four assignments $(a_i)^v_{\$[.in}$, $(a_i)^v_{\$[.out}$, $(a_i)^v_{\$].in}$, and $(a_i)^v_{\$].out}$ (inductively) as follows.

$$(a_i)^v_{\$[.in} := \begin{cases} q_\perp & \ldots\ v \text{ is the root node} \\ (a_i)^{v_0}_{\$[.out} & \ldots\ v \text{ is the first child of } v_0 \\ (a_i)^{v_0}_{\$].out} & \ldots\ v \text{ is the right sibling of } v_0 \end{cases}$$

$$(a_i)^v_{\$].in} := \begin{cases} (a_i)^v_{\$[.out} & \ldots\ v \text{ has no children} \\ (a_i)^w_{\$].out} & \ldots\ w \text{ is the rightmost child of } v \end{cases}$$

In the first visit of node v, we compute

$$\langle (a_1)^v_{\$[.out}, \ldots, (a_k)^v_{\$[.out}, \sigma\rangle := f^v_{\$[}\big((a_1)^v_{\$[.in}, \ldots, (a_k)^v_{\$[.in}\big)$$

and write σ to the output. In the second, we compute

$$\langle (a_1)^v_{\$].out}, \ldots, (a_k)^v_{\$].out}, \sigma\rangle := f^v_{\$]}\big((a_1)^v_{\$[.in}, \ldots, (a_k)^v_{\$[.in}, (a_1)^v_{\$].in}, \ldots, (a_k)^v_{\$].in}\big)$$

and write σ to the output. In case $f^v_{\$[}$ or $f^v_{\$]}$ is undefined on its input, the evaluation terminates and the input is rejected.

The result of an XSAG on an input tree is the *output* (rather than attribute values) it computes, if it accepts its input. □

Even though this semantics definition may seem involved, we believe that its application is natural.

Example 10. Consider the bXSAG G with $Dom = \{q_\perp, q_{book}, q_{article}\}$, $Att = \{prev\}$, productions

$$\begin{aligned} bib &::= \{f^{bib}_{\$[}\}\ \text{bib}((book \cup article)^*)\ \{f^{bib}_{\$]}\} \\ book &::= \{f^{book}_{\$[}\}\ \text{book}(\epsilon) \\ article &::= \{f^{article}_{\$[}\}\ \text{article}(\epsilon) \end{aligned}$$

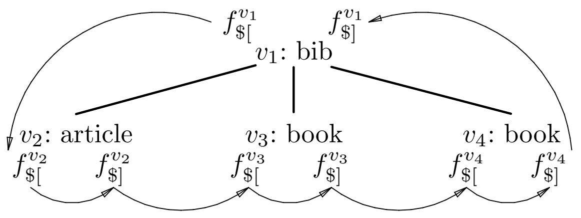

Fig. 3. bXSAG parse tree and traversal of Example 10.

and the attribution functions

$$\begin{aligned} f^{bib}_{\$[} &: x \mapsto (x, \langle \text{bib} \rangle) \\ f^{article}_{\$[} &: x \mapsto (q_{article}, \langle \text{article/} \rangle) \\ f^{book}_{\$[} &: (x_0, x) \mapsto \begin{cases} (q_{book}, \langle \text{book/} \rangle) & \ldots\ x = q_{article} \\ (q_{book}, \epsilon) & \ldots\ \text{otherwise} \end{cases} \\ f^{bib}_{\$]} &: (x_0, x) \mapsto (x, \langle \text{/bib} \rangle) \end{aligned}$$

The grammar requires the input to consist of a dummy bibliography database containing book and article nodes without children. As output, the XSAG writes a root node labeled “bib”, to which it assigns bachelor nodes labeled “book” and “article” as children, filtering out books which are not right neighbors of articles[11].

The parse tree of XML document

$$\langle \text{bib} \rangle\ \langle \text{article/} \rangle\ \langle \text{book/} \rangle\ \langle \text{book/} \rangle\ \langle \text{/bib} \rangle$$

is shown in Figure 3. Naturally, we assign $f^{bib}_{\$[}$ to $f^{v_1}_{\$[}$, $f^{bib}_{\$]}$ to $f^{v_1}_{\$]}$, $f^{article}_{\$[}$ to $f^{v_2}_{\$[}$, $f^{book}_{\$[}$ to $f^{v_3}_{\$[}$ and $f^{v_4}_{\$[}$, and have $f^{v_2}_{\$]}, f^{v_3}_{\$]}, f^{v_4}_{\$]} : (x_1, x_2) \mapsto (x_2, \epsilon)$, i.e., the default. (The attribute *prev* is initialized with $q_\perp$.)

G is evaluated on the parse tree as shown in Table 1. The four columns have the following meaning. The first column shows which attribution function is applied in the respective step. The second and third columns show the value of our attribute before and after the application of the attribution function, respectively, and the rightmost column shows which output is produced and written to the output stream. Clearly, G outputs

$$\langle \text{bib} \rangle\ \langle \text{article/} \rangle\ \langle \text{book/} \rangle\ \langle \text{/bib} \rangle$$

and accepts its input. $\square$

[11] This is a somewhat contrived example but illustrates a number of important points related to the evaluation of XSAGs.

	Attribute value		
F	input	output	Output
$f^{v_1}_{\$[}$	$q_\perp$	$q_\perp$	$\langle$bib$\rangle$
$f^{v_2}_{\$[}$	$q_\perp$	$q_{article}$	$\langle$article/$\rangle$
$f^{v_2}_{\$]}$	$q_{article}$	$q_{article}$	ϵ
$f^{v_3}_{\$[}$	$q_{article}$	q_{book}	$\langle$book/$\rangle$
$f^{v_3}_{\$]}$	q_{book}	q_{book}	ϵ
$f^{v_4}_{\$[}$	q_{book}	q_{book}	ϵ (!!)
$f^{v_4}_{\$]}$	q_{book}	q_{book}	ϵ
$f^{v_1}_{\$]}$	q_{book}	q_{book}	$\langle$/bib$\rangle$

Table 1. Run of bXSAG G of Example 10.

Example 11. Consider again the XSAG of the previous example. Alternatively, to reject the input[12] if two books arrive in sequence, we define $f^{book}_{\$[}$ as

$$f^{book}_{\$[} : x \mapsto \begin{cases} (q_{book}, \langle \text{book/} \rangle) \dots & x = q_{article} \\ \text{undefined} & \dots \text{ otherwise.} \end{cases}$$

This new XSAG rejects the input document of Example 10. □

3.2 Concrete XSAGs

We introduce a simple imperative programming language for the definition of attribution functions. This language is basically a fragment of Pascal, comprising the following constructs: (1) if-then-else statements, (2) blocks of multiple commands starting with the keyword "begin" and ending with "end", (3) boolean formulas – using "and", "or", and "not" – over equality conditions (used in if-statements), (4) assignments, (5) the keyword "reject" for terminating the computation and rejecting the input, and (6) "print" statements taking a constant string as argument.

An assignment is a statement of the form $x := y$, where x is an l-value and y is an r-value. [13] An equality condition is a statement of the form $x = y$, where x and y are r-values.

The semantics of such a program is defined using the usual notion of an environment, a function $\mathcal{E} : Att \rightarrow Dom$ that maps each attribute name to a domain value. For a program defining a function $f_{\$[} : Dom^k \rightarrow Dom^k \times string$, at the start of the execution of $f_{\$[}(\boldsymbol{x})$ (where $\boldsymbol{x} = \langle x_1, \dots, x_k \rangle$), $\mathcal{E}(a_i) = x_i$ for $1 \leq i \leq k$, $Att = \{a_1, \dots, a_k\}$, where the attributes a_i may be read as well as

[12] This could be alternatively achieved by modifying the grammar rather than the attributions as done in this example, but the goal here is to illustrate the use of partially undefined attribution functions.

[13] We call constructs of our language that may appear on the right side of an assignment *r-values* and and those that may appear on the left side of an assignment *l-values*.

written (by assignment ":="). $f_{\$[}(\boldsymbol{x})$ evaluates to $\langle \mathcal{E}^\omega(a_1), \ldots \mathcal{E}^\omega(a_k), o\rangle$, where $\mathcal{E}^\omega$ is the environment at the end of the execution and o is the concatenation of the symbols printed. Thus, for (partial) functions in $F_{\$[}$, the l-values consist of the set $\{\$[.a \mid a \in \mathit{Att}\}$ and the r-values consist of $\{\$[.a \mid a \in \mathit{Att}\} \cup \mathit{Dom}$.

For a program defining a function $f_{\$]} : \mathit{Dom}^{2k} \to \mathit{Dom}^k \times \mathit{string}$, at the start of the execution of $f_{\$]}(\$[.\boldsymbol{x}, \$].\boldsymbol{x})$, $\langle \$[.\boldsymbol{x}, \$].\boldsymbol{x}\rangle$ is copied into the environment, but the attributes of $\$[.\boldsymbol{x}$ are read-only (i.e., must not be used on the left-hand sides of assignments). For (partial) functions in $F_{\$]}$, the l-values are $\{\$].a \mid a \in \mathit{Att}\}$ and the r-values are $\{\$[.a, \$].a \mid a \in \mathit{Att}\} \cup \mathit{Dom}$.

Such a program defines functions in $F_{\$[}$ resp. $F_{\$]}$ in the obvious way, with the notable fact that the functions are assumed undefined for inputs for which the "reject" statement is called.

Example 12. Using our Pascal-like syntax, we define the attribution functions of Example 10 as

$$\begin{aligned}
f_{\$[}^{bib} &= \{\text{print } \langle\text{bib}\rangle\} \\
f_{\$]}^{bib} &= \{\text{print } \langle/\text{bib}\rangle\} \\
f_{\$[}^{article} &= \{\text{print } \langle\text{article}/\rangle; \$[.\mathit{prev} := q_{article}\} \\
f_{\$[}^{book} &= \{\text{if } \$[.\mathit{prev} = q_{article} \text{ then print } \langle\text{book}/\rangle; \\
&\qquad \$[.\mathit{prev} := q_{book}\}
\end{aligned}$$

Thus, we can write the bXSAG of Example 10 as

$$\begin{aligned}
\mathit{bib} &::= \{\text{print } \langle\text{bib}\rangle\} \text{ bib}((\mathit{book} \cup \mathit{article})^*) \; \{\text{print } \langle/\text{bib}\rangle\} \\
\mathit{book} &::= \{\text{if } \$[.\mathit{prev} = q_{article} \text{ then print } \langle\text{book}/\rangle; \\
&\qquad \$[.\mathit{prev} := q_{book}\} \text{ book}(\epsilon) \\
\mathit{article} &::= \{\text{print } \langle\text{article}/\rangle; \$[.\mathit{prev} := q_{article}\} \text{ article}(\epsilon)
\end{aligned}$$

To modify the XSAG to reject its input if two books arrive in sequence on the stream as in Example 11, we define $f_{\$[}^{book}$ as

$$\{\text{if } \$[.\mathit{prev} = q_{article} \text{ then begin print } \langle\text{book}/\rangle;\ \$[.\mathit{prev} := q_{book} \text{ end else reject}\}$$

□

3.3 Built-in Macros

For the convenient definition of queries using XSAGs, we introduce three standard built-in macros, *ECHO*, *ECHO_OFF*, and *MATCH_CHILDREN*. These are redundant with the formalism presented so far, but allow to define queries in a more concise way.

If macro *ECHO* is used in a first-visit attribution function, the subtree of the parse tree to which the attribution function applies is copied to the output. Correspondingly, macro *ECHO_OFF* can be used to override *ECHO* and to suppress the output of certain XML subtrees. Example 1 illustrates the use of the macro *ECHO*. Below, we show an example that combines *ECHO* and *ECHO_OFF*.

Example 13. The bXSAG

$$\begin{aligned} bib &::= \{ECHO\}\ \mathrm{bib}\big(book^*\big) \\ book &::= \mathrm{book}\big(title.author.(\{ECHO\_OFF\}\ author^*)\big) \\ title &::= \mathrm{title}(PCDATA) \\ author &::= \mathrm{author}(PCDATA) \end{aligned}$$

outputs each book with its title and first author (dropping further authors). □

To realize *ECHO*, we define a boolean attribute $echo \in Att$, initialized with *false*. Occurrences of *ECHO* are replaced by $\$[.echo := true$ and occurrences of *ECHO_OFF* are replaced by $\$[.echo := false$.

For every production $\{f_I\}t(\rho)\{f_{II}\}$,

- if $t \in Tag$, we append

$$\text{if } \$[.echo = true \text{ then print } \langle \mathrm{t} \rangle$$

  to $f_{\$[}$ and

$$\text{if } \$[.echo = true \text{ then print } \langle /\mathrm{t} \rangle$$

  to $f_{\$]}$.
- If $t \in Char$ we append "if $\$[.echo = true$ then print t" to $f_{\$[}$.

In both cases, we finally append "$\$].echo := \$[.echo$" to $f_{\$]}$. On leaving a node on which *ECHO* or *ECHO_OFF* is used, attribute *echo* is reset to its former value. Thus, even though *ECHO* may be overridden by *ECHO_OFF* within a subtree, we cannot accidentally create malformed documents.

The *MATCH_CHILDREN*(ρ, c) macro matches the string obtained by concatenating the character data encountered when traversing the children of a node from left to right against a regular expression ρ, yielding *true* or *false* as a value for the user-defined condition attribute c. *MATCH_CHILDREN* may only be used in first-visit attribution functions. The evaluation result becomes available in the corresponding second-visit attribution function.

Example 14. We slightly extend Example 3. The yXSAG production

```
book ::= book(({MATCH_CHILDREN(2003, $[.c)}year).
              ({if $[.c = true then
                 begin
                    print ⟨book⟩; ECHO
                 end}
               (title.author.author*)
               {if $].c = true then print ⟨year⟩2003⟨/year⟩⟨/book⟩}))
```

selects those books for which child "year" has string value "2003"; moreover, the year is output as the rightmost child of book, rather than as the leftmost as required for the input. (We omit productions defining bib, year, title, and author, which are as in Example 15.) □

MATCH_CHILDREN can be easily implemented by compiling ρ into a DFA and running it on the characters visited while traversing a node's children.

3.4 bXSAGs vs. yXSAGs

As we show in the next section, bXSAGs and yXSAGs have the same expressive power. However, yXSAGs are more convenient to use. In particular, it is often necessary to introduce more attributes and more complicated attribution functions to encode a given query as a bXSAG than to encode it as a yXSAG.

Example 15. Consider the yXSAG with productions

$$
\begin{aligned}
\mathit{bib} &::= \{\text{print } \langle\text{bib}\rangle\}\ \text{bib}(\mathit{article}^*)\ \{\text{print } \langle/\text{bib}\rangle\}\\
\mathit{article} &::= \text{article}\Big(\big(\{\text{print } \langle\text{article\_short}\rangle; \mathit{ECHO}\}\\
&\qquad\qquad (\mathit{title}.\mathit{author}.\mathit{author}^*)\\
&\qquad \{\text{print } \langle/\text{article\_short}\rangle\}\big)\\
&\qquad \cup\\
&\qquad \big(\{\text{print } \langle\text{article\_long}\rangle; \mathit{ECHO}\}\\
&\qquad\qquad (\mathit{year}.\mathit{title}.\mathit{author}.\mathit{author}^*.\mathit{pub})\\
&\qquad \{\text{print } \langle/\text{article\_long}\rangle\}\big)\Big)\\
\mathit{title} &::= \text{title}(\mathit{PCDATA})\\
\mathit{author} &::= \text{author}(\mathit{PCDATA})\\
\mathit{year} &::= \text{year}(\mathit{PCDATA})\\
\mathit{pub} &::= \text{pub}(\mathit{PCDATA})
\end{aligned}
$$

Article entries can appear either in a short version with title and authors only, or in a long version which also contains a year and a publisher. In the former case, article nodes are relabeled as *article_short*, and in the latter case, article nodes are relabeled *article_long*.

The following bXSAG is equivalent to the above yXSAG:

$$
\begin{aligned}
\mathit{bib} &::= \{\text{print } \langle\text{bib}\rangle\}\ \text{bib}(\mathit{article}^*)\ \{\text{print } \langle/\text{bib}\rangle\}\\
\mathit{article} &::= \{\$[.\mathit{state} := q_{unknown}\}\\
&\quad \text{article}\big((\mathit{title}.\mathit{author}.\mathit{author}^*) \cup\\
&\qquad\qquad (\mathit{year}.\mathit{title}.\mathit{author}.\mathit{author}^*.\mathit{pub})\big)\\
&\quad \{\text{if } \$].\mathit{state} = q_{short} \text{ then print } \langle/\text{article\_short}\rangle\\
&\quad\ \text{else if } \$].\mathit{state} = q_{long} \text{ then print } \langle/\text{article\_long}\rangle\}\\
\mathit{title} &::= \{\text{if } \$[.\mathit{state} = q_{unknown} \text{ then}\\
&\qquad \text{begin } \$[.\mathit{state} := q_{short};\ \text{print } \langle\text{article\_short}\rangle \text{ end}; \mathit{ECHO}\}\\
&\quad \text{title}(\mathit{PCDATA})\\
\mathit{author} &::= \{\mathit{ECHO}\}\ \text{author}(\mathit{PCDATA})\\
\mathit{year} &::= \{\$[.\mathit{state} := q_{long};\ \text{print } \langle\text{article\_long}\rangle;\ \mathit{ECHO}\}\\
&\quad \text{year}(\mathit{PCDATA})\\
\mathit{pub} &::= \{\mathit{ECHO}\}\ \text{pub}(\mathit{PCDATA})
\end{aligned}
$$

While there are other ways of encoding our query as a bXSAG, it does not seam to be possible to represent the query as a bXSAG without using attributes. □

It is easy to verify that bXSAGs equivalent to the yXSAGs of Examples 2 and 14 are also much more complicated.

4 Expressive Power of XSAGs

We introduce deterministic pushdown transducers (*DPDT*s) as deterministic pushdown automata with output which accept by empty stack. As stated in [2], the *DPDT*s accepting by empty stack are equivalent to the *DPDT*s accepting by final state.

Definition 6 (*DPDT*). A *deterministic pushdown transducer* is a tuple

$$\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$$

where Q is a finite set of states, Σ, Γ, and Δ are the finite alphabets for input tape, stack, and output tape respectively, δ is the partial transition function

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^* \times \Delta^*,$$

$q_0 \in Q$ denotes the initial state, and Z_0 the initial stack symbol. For each $q \in Q$ and $X \in \Gamma$ s.t. $\delta(q, \epsilon, X)$ is defined, $\delta(q, a, X)$ is undefined for all $a \in \Sigma$.

We define a run of $\mathcal{T}$ by means of *instantaneous descriptions* (IDs). An ID is a quadruple

$$(q, w, \alpha, o) \in Q \times \Sigma^* \times \Gamma^* \times \Delta^*,$$

where q is a state, w is the remaining input, α a string of stack symbols, and o the output generated so far. We make a transition

$$(q, aw, X\alpha, o) \vdash (q', w, \gamma\alpha, o\sigma)$$

if $\delta(q, a, X) = (q', \gamma, \sigma)$ where $a \in \Sigma \cup \{\epsilon\}$, $X \in \Gamma$, $q' \in Q$, and $\sigma \in \Delta^*$.

Here, $\gamma \in \Gamma^*$ is the string of stack symbols which replace X on top of the stack. For $\gamma = \epsilon$, the stack is popped, whereas for $\gamma = X$, the stack remains unchanged. If $\gamma = YX$, then stack symbol Y is pushed on top of X.

Let $\vdash^*$ be the reflexive and transitive closure of $\vdash$. $\mathcal{T}$ accepts by empty stack if

$$(q_0, w, Z_0, \epsilon) \vdash^* (q, \epsilon, \epsilon, o)$$

for $w \in \Sigma^*, q \in Q$, and $o \in \Delta^*$. We say o is the output for input word w.

The translation defined by an *XML-DPDT* $\mathcal{T}$, denoted $T(\mathcal{T})$, is

$$\{(w, o) \mid (q_0, w, Z_0, \epsilon) \vdash^* (q, \epsilon, \epsilon, o) \textit{ for some } q \in Q\},$$

We call two DPDTs equivalent if they define the same translation. □

Definition 7 (Well-formedness). An XML document is called *well-formed* iff it conforms to an extended regular tree grammar $G = (Nt, T, P, s)$ where for all productions of the form $s ::= t(\rho)$, $t \in \mathit{Tag}$. $\square$

A well-formed document contains at least one element (i.e. the root element) and has element start-tags and end-tags properly nested within each other. Furthermore, the first symbol in the document must be the opening tag for the root node. An XML document is *malformed* if it is not well-formed.

Definition 8 (*XML-DPDT*). Let input alphabet $\Sigma = \{\langle t\rangle, \langle /t\rangle \mid t \in \mathit{Tag}\} \cup \mathit{Char}$ consist of matching opening and closing tags and characters.

Throughout this paper, for a set S, we use $S^{\leq 2}$ as a shortcut for $\{\epsilon\} \cup S \cup S \times S$.

An *XML-DPDT* is a *DPDT*

$$\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$$

which rejects malformed XML documents and for which the transition function δ is restricted as follows:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^{\leq 2} \times \Delta^*$$

- In the very first transition the initial stack symbol Z_0 is replaced; i.e., we require $\delta(q_0, \langle t\rangle, Z_0) = (p, Y, \sigma)$ for $\langle t\rangle \in \Sigma$, $p \in Q$, $Y \in \Gamma$, and $\sigma \in \Delta^*$.
- For all other configurations of $q \in Q$ and $X \in \Gamma$, a symbol is only pushed on the stack when an opening tag is read from the input stream: $\delta(q, \langle t\rangle, X) = (p, YX, \sigma)$ for $\langle t\rangle \in \Sigma$, $p \in Q$, $Y \in \Gamma$, and $\sigma \in \Delta^*$.
- A symbol is only popped from the stack when a closing tag is encountered in the input stream, i.e. $\delta(q, \langle /t\rangle, X) = (p, \epsilon, \sigma)$ for $q, p \in Q$, $\langle /t\rangle \in \Sigma$, $X \in \Gamma$, and $\sigma \in \Delta^*$. $\square$

The conditions required in the definition of *XML-DPDT* are only natural in the context of XML stream processing: The size of the stack is bounded by the maximum depth of the incoming document tree. Moreover, the input has to start with the root element of the XML document being read. Due to acceptance by empty stack, only well-formed XML-documents are accepted.

Let $L(G)$ be the language accepted by XSAG $G = (NT, T, P, s)$. For *input* $w \in L(G)$, $G(w)$ is the *output* produced by G in accepting w. The *translation* defined by an XSAG G, written $T(G)$, is $\{(w, o) \mid w \in L(G) \textit{ and } o \in G(w)\}$. We call XSAG G and *XML-DPDT* $\mathcal{T}$ equivalent if they define the same translation, i.e. $T(G) = T(\mathcal{T})$.

Both bXSAGs and yXSAGs are precisely as expressive as *XML-DPDT*s[14].

Theorem 1. *For each bXSAG, there is an equivalent XML-DPDT.*

Theorem 2. *For each XML-DPDT, there is an equivalent XSAG which is both a bXSAG and a yXSAG.*

[14] Note, however, that there are bXSAGs that are no yXSAGs and vice versa, so Theorems 1 and 3 are not redundant.

Theorem 3. *For each yXSAG, there is an equivalent XML-DPDT.*

The lengthy proofs for these three theorems will be provided in the long version of this paper.

5 Efficient Evaluation of XSAGs

The proofs of Theorems 1 and 3 are based on a translation to DPDTs which, as a method for evaluating XSAGs, has the strong point that once the DPDT has been created, the query evaluation time is in principle independent of the size of the XSAG/DPDT and only depends on the input data.

Corollary 1. *An XSAG G can be evaluated on a tree T in time $O(f(|G|)+|T|)$ using only a stack of memory of size $O(depth(T))$.*

However, the DPDTs of the construction of the proof of Theorem 1 are of size *exponential* in the number k of attributes in the XSAG, i.e., f is $O(2^k)$.

The exponential-time compilation phase can be avoided by using a simple hybrid evaluation method in which the grammars (and in particular the regular expressions appearing in the grammar productions) are compiled into transducers which however *interpret* the attribution functions (rather than materializing the *graphs* of the attribution functions as is done in our proofs). Thus one obtains an XSAG evaluation method which runs scalably on streams and which is strictly polynomial in the size of the XSAG.

Theorem 4. *A bXSAG G can be evaluated on a tree T in time $O(|G|^2+|T|\cdot|G|)$ using a stack of size $O(depth(T))$.*

Theorem 4 also makes use of the fact that DFAs for one-unambiguous regular expressions can be computed in polynomial (actually, quadratic) time[15] [6].

For yXSAGs, the construction of the proof of Theorem 3 is in addition exponential in the maximum depth of the parse trees of the regular expressions used (which only depend on the XSAG). This can be resolved by pushing attributes onto the stack at yXSAG regular expression nodes as well. The stack consumption of course remains proportional to the depth of the input tree.

The main technical challenge we have to deal with when evaluating yXSAGs is the matching of attributed regular expressions on the stream and the invocation of attribution functions at the right time.

A finite-state transducer (FST) is an NFA with output, which in each transition $q \xrightarrow{a/w} q'$ from state q to q' on input symbol a outputs a fixed word w. A deterministic finite-state transducer (DFT) is an FST that is deterministic, i.e.,

[15] This construction – that of the Glushkov automaton of a regular expression – also provides a procedure for deciding one-ambiguity with the same complexity. A regular expression is known to be one-unambiguous precisely if its Glushkov automaton is deterministic.

which is a DFA if the output is ignored and for which no two transitions $q \overset{a/v}{\rightarrow} q'$ and $q \overset{a/w}{\rightarrow} q'$ exist such that $v \neq w$.

Below, in regular expressions of the form $\rho.\odot$, let $\odot$ be a new end-marker symbol that does not occur in ρ.

Theorem 5. *Let ρ be a regular expression. Then, there is an FST $\mathcal{A}^{[]}(\rho)$ which*

1. *recognizes $L(\rho.\odot)$,*
2. *is deterministic iff ρ is strongly one-unambiguous,*
3. *if ρ is strongly one-unambiguous, outputs the bracketing of word w for each $w.\odot \in L(\rho.\odot)$, and*
4. *can be computed in time $O(|\rho|^3)$.*

Using the DFT construction of Theorem 5, the preprocessing phase for yXSAGs takes time cubic in the size of each of the productions.

Theorem 6. *A yXSAG G can be evaluated on a tree T in time $O(|G|^3 + |T| \cdot |G|)$ using a stack of size $O(depth(T) \cdot |G|)$.*

6 Discussion and Conclusions

The goal of this paper was to develop a framework for query formulation which

1. satisfies our criteria for scalable query processing on streams,
2. has a good and well-justified foundation, and
3. is user-friendly, i.e. allows to state many common queries quickly and easily.

We can argue that XSAGs satisfy these three desiderata.

(1) Each XSAG can be translated into a DPDT with a stack discipline that assures that the size of the stack remains proportional to the depth of the XML tree. This is known to be the minimum amount of memory required to do any meaningful (sequential) processing of XML data [10]. Of course queries are evaluated strictly in linear time.

(2) Throughout the paper, we have explained and justified our design choices. Regular tree grammars are a commonly accepted grammar formalism for XML (as are DTDs, which are restricted regular tree grammars). In the right-hand sides of the productions of such grammars, we use regular expressions to be able to parse nodes with an unbounded number of children. Our restriction of these regular expressions to strongly one-unambiguous ones in the case of yXSAGs allows for precisely those expressions for which the parse trees of words can be unambiguously generated using a lookahead of only one symbol (a necessity in stream processing). Having the regular expressions inside grammar productions available for attribution allows to conveniently define attribute grammars for unranked trees, and to approach the usability of XML Query with a formalism that allows for much better control of complexity.

We have precisely characterized the expressive power of XSAGs relative to deterministic pushdown transducers.

Note that our formalism fully fits into the classical framework of attribute grammars (and more precisely, L-attributed grammars), even if we did not introduce, say, the distinction between synthesized and inherited attributes.

(3) A number of examples in this paper and our experiences with many more demonstrate that XSAGs are of practical value, and that they fill an important void in the design space of tailored query languages.

Earlier in this paper, we defined XSAGs with attributes ranging exclusively over a finite domain to be able to assure scalability and memory bounds in the strongest sense. However, it is desirable and often justified to generalize this framework to make certain uniformity assumptions and to allow for values from an infinite domain. In the future, we plan to carry out a more detailed study of conservative extensions of our formalism with small buffers (using uniformity assumptions for numbers, small strings, and small subtrees).

Acknowledgments

We thank Peter Buneman, Nicole Schweikardt, Bernhard Stegmaier, Val Tannen, and the anonymous reviewers of DBPL 2003 for their suggestions, which helped to substantially improve the presentation of the paper.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling. I: Parsing*, volume 1. Prentice-Hall, 1972.
3. M. Benedikt, C. Y. Chan, W. Fan, J. Freire, and R. Rastogi. "Capturing both Types and Constraints in Data Integration". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 277–288, 2003.
4. M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. "DTD-Directed Publishing with Attribute Translation Grammars". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 838–849, 2002.
5. T. Bray, J. Paoli, and C. Sperberg-McQueen. "Extensible Markup Language (XML) 1.0". Technical report, W3C, Feb. 1998.
6. A. Brüggemann-Klein and D. Wood. "One-Unambiguous Regular Languages". *Information and Computation*, **142**(2):182–206, 1998.
7. L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. "Query Processing of Streamed XML Data". In *Proc. 11th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 126–133, 2002.
8. T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. "Processing XML Streams with Deterministic Automata". In *Proc. of the 9th International Conference on Database Theory (ICDT'03)*, 2003.
9. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Reading, MA USA, 1979.

10. C. Koch. "Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach". In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, Berlin, Germany, 2003.
11. D. Lee, M. Mani, and M. Murata. "Reasoning about XML Schema Languages using Formal Language Theory". Technical Report RJ 10197 Log 95071, IBM Research, Nov. 2000.
12. B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. "A Transducer-Based XML Query Processor". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 227–238, 2002.
13. F. Neven. "Extensions of Attribute Grammars for Structured Document Queries". In *Proc. 8th International Workshop on Database Programming Languages (DBPL)*, pages 99–116, 1999.
14. F. Neven and J. van den Bussche. "Expressiveness of Structured Document Query Languages Based on Attribute Grammars". *Journal of the ACM*, **49**(1):56–100, Jan. 2002.
15. D. Olteanu, T. Kiesling, and F. Bry. "An Evaluation of Regular Path Expressions with Qualifiers against XML Streams". In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE)*, Bangalore, Mar. 2003. Poster Session.
16. World Wide Web Consortium. "XML Query". http://www.w3c.org/XML/query/.