Syntax-directed Transformations of XML Streams

Stefanie Scherzinger[*,†] Alfons Kemper[‡]

[†] Technische Universität Wien, Vienna, Austria, *scherzinger@wit.tuwien.ac.at*
[‡] Technische Universität München, Munich, Germany, *alfons.kemper@in.tum.de*

Abstract

We discuss the TransformX framework for syntax-directed transformations of XML streams. In this framework, we define stream transformations as a special form of attributed extended regular tree grammars where all attributes can be evaluated in a single pass over the input, a necessity in stream processing. In the tradition of tools such as Yacc, the TransformX parser generator translates attribute grammars to Java source code, which is then compiled and evaluated. We motivate our approach in developing this tool, present the theoretical foundations, and study the complexity of program generation. We further provide details on our prototype implementation and first experimental results.

1 Introduction

In the database community, the efficient processing of XML streams has been identified as a nontrivial research challenge. In this context, we make the following three assumptions:

1. XML streams are XML documents of possibly infinite length.
2. Applications for XML stream processing need to be completely main-memory based.
3. Schema information is provided with the data stream.

Applications for XML stream processing need to deal with assumptions (1) and (2), and ideally exploit the fact that a schema description is available according to assumption (3).

Recently, a lot of attention has been devoted to the efficient evaluation of Boolean or node-selecting queries on XML streams [3, 10, 12, 13]. Yet so far, there are not many frameworks which allow for *transformations* of XML streams, and even fewer that follow a syntax-directed approach.

Setting aside the first assumption, i.e., for XML documents small enough to fit into main memory, powerful transformations may be easily specified, for instance using XQuery [25] or XSLT [26]. The corresponding query engines typically operate on in-memory tree representations of the complete input document. Consequently, they cannot scale to XML streams.

Lately, approaches have been developed to adapt existing XSLT and XQuery query engines to assumption (1) by reducing main memory consumption. State-of-the-art techniques produce incremental output [4], perform a projection on the input before loading it into main memory [19], and even evaluate parts of the query in an event-based approach [15, 16]. In contrast, the scripting language STX [11] has been specifically tailored to the transformation of XML streams.

However, the tools mentioned above do not employ schema information, as available by assumption (3), in order to assist users in specifying transformations. There are various schema languages for XML, among the most widely known are XML Schema and DTDs. The latter are a dialect of extended regular tree grammars, which are well-suited for specifying *attribute grammars*.

Attribute grammars [1] are widely agreed to carry a strong intuition for specifying syntax-directed translations: The programmer takes a grammar describing the input and inserts *attribution functions* inside the grammar productions. Parser generators automatically translate attribute grammars into the complete code for processing the input. After compilation, the program executes the attribution functions while the input is being parsed.

Thus, attribute grammars allow users to specify many transformations conveniently, as users merely need to "fill in the blanks" instead of having to program transformations from scratch. Further, attribute grammars make users aware of the structure of the input, which may reveal possibilities for optimization. This is especially interesting in stream processing, where it is crucial that transformations can be evaluated efficiently w.r.t. main memory and CPU consumption.

This paper proposes a full-scale framework – called *TransformX* – for XML transformations based on attribute grammars and introduces the TransformX parser generator which automatically translates attributed extended regular tree grammars to Java code. The output of a TransformX attribute grammar need not be XML. Rather, TransformX attribute grammars may be regarded as a

[*] This research has been partly funded by the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF) under grant 31.963/46-VII/9/2002.

comprehensive programming language for XML streams, with all the capabilities of Java: Users may introduce variables, strings, and arrays, and use their own data structures and methods. This allows for a wide range of applications, from filtering and transformation tasks to interaction with databases (e.g., using JDBC), where we may store parts of the stream in databases or enrich the stream by joining streaming and persistent data.

For the examples shown in this paper and based on our own experience, application development with TransformX attribute grammars becomes a convenient task and is often preferable to writing programs directly from scratch, as the following example demonstrates.

Example 1 Consider the extended regular tree grammar $G = (Nt, T, P, bib)$ defining an XML bibliography. G has nonterminals *Nt*, grammar start symbol *bib*, terminals T, and productions P defined as follows:

$$Nt = \{bib, pub, year, title, author\}$$

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

$$\begin{array}{rcl} P: \quad bib & ::= & \text{bib}\big(pub^*\big) \\ pub & ::= & \text{book}\big(year.title.author.author^*\big) \\ pub & ::= & \text{article}\big(year.title.author.author^*\big) \\ year & ::= & \text{year}\big(PCDATA\big) \\ title & ::= & \text{title}\big(PCDATA\big) \\ author & ::= & \text{author}\big(PCDATA\big) \end{array}$$

From an XML stream conforming to this grammar, we want to derive a transformed stream consisting of root node "books", followed by all the book subelements. Each book is given a numerical identifier "id" as its first child node, and the order of title and year are reversed.

Using a TransformX attribute grammar, we specify which parts of the document are to be transformed by adding attribution functions to productions: In the tradition of parser generators such as Yacc [18], the TransformX attribute grammar in Figure 1 consists of three sections separated by "**%%**":

a) In the *definition section*, we state the name of the generated Java class. Further, we declare an output stream "output" and a string-buffer "buffer". String-buffers may be written to, their contents copied to the output, or cleared[1].

b) The *rules section* contains the attributed grammar productions with attribution functions enclosed in curly braces. Here, we specify that the XML start and end tags of the root node are renamed to "books".

   Upon seeing a book, the corresponding start and end tags are output and the numerical identifier is inserted. The year is buffered, yet the title is output directly. The buffered year is output before the authors, thus reversing the order of title and year.

[1]We provide more details on these classes in Section 4.2.

```
%class Example1
%echo TXWriter output = new TXWriter(System.out);
%echo TXBuffer buffer = new TXBuffer();

%%
 bib    ::= { output.write("⟨books⟩"); } bib(pub*)
             { output.write("⟨/books⟩"); }
 pub    ::= { output.write("⟨book⟩");
              output.write("⟨id⟩"+ i++ +"⟨/id⟩"); }
             book(({ buffer.write($this); } year).
                  ({ output.write($this); } title).
                  ({ output.write(buffer); buffer.clear();
                     output.write($this); } author.author*))
             { output.write("⟨/book⟩"); }
 pub    ::= article(year.title.author.author*)
 year   ::= year(PCDATA)
 title  ::= title(PCDATA)
 author ::= author(PCDATA)

%%
int i = 0;
```

Figure 1: TransformX attribute grammar of Example 1.

c) The *class member section* provides room for auxiliary attributes and methods which can then be accessed inside attribution functions. In our example, we declare an integer-counter "i".

This simple example shows how the underlying grammar conveniently provides us with context information, e.g. as to which titles are part of books and which are part of articles. If the same transformation were programmed using a SAX parser, the code would be scattered with corresponding case distinctions.

Also, the grammar component may reveal potential for optimization. In this example, it becomes apparent that titles can be written directly to the output without being buffered first. □

The source code produced by the TransformX parser generator can then be compiled and executed. The resulting program will validate the input against the grammar component and evaluate the corresponding attribution functions while the input is being parsed. Thus, writing applications for XML stream processing with TransformX attribute grammars is mainly facilitated by "filling in" the blanks in a grammar, while the remaining parts of the program are generated automatically.

Contributions The contributions of this paper are the following.

- We formally define TransformX attribute grammars.
- We discuss how users can specify Java-based TransformX attribute grammars, how they may copy subtrees directly from the input to the output, handle

main memory buffers, and add their own data structures in practice.

- We introduce the TransformX parser generator and study the time complexity of translating TransformX attribute grammars to Java code.
- We present a prototype implementation and show first experiments with our framework.

Structure In the following section, we give a brief overview over the related work. In Section 3, we present the necessary theoretical background in language theory. The TransformX attribute grammars are introduced in Section 4, while their translation to Java source code by the TransformX program generator is studied in Section 5. In Section 6, we discuss our prototype implementation and experiments with it. We conclude with a summary and suggestions on how we might optimize the evaluation of TransformX attribute grammars in future work.

2 Related Work

To our knowledge, the scripting language STX [11] is the most related tool for XML stream processing. STX employs a processing model where stylesheets are evaluated in a single pass over the input, and allows for the specification of powerful transformations of XML streams. While the syntax of STX resembles XSLT, TransformX attribute grammars specify syntax-directed transformations.

Attribute grammars have recently been revisited in the context of XML, for instance for grammar-directed XML publishing [5, 6]. Some of the theory of attribute grammars relevant in the context of structured documents has been studied in [14, 20–22]. In [20], regular languages are used as *conditions* for attribute assignment. This approach seems to be hard to use in practice and infeasible on streams, as an unbounded number of nodes which have already passed may have to be kept in memory.

In contrast, the XML Stream Attribute Grammars (XSAGs) [14] define strictly linear-time one-pass XML transformations. XSAGs are as expressive as deterministic pushdown transducers with a natural stack discipline that assures that the size of the stack remains strictly proportional to the depth of the XML input document, and which can only accept well-formed XML documents. TransformX attribute grammars also only accept well-formed XML documents. Yet here, the attribution functions are encoded in a Turing-complete programming language, allowing for more powerful transformations. Consequently, TransformX attribute grammars cannot be *guaranteed* to scale to streams. Apart from the difference in expressiveness, TransformX attribute grammars are also more flexible w.r.t. where attribution functions can be specified in the grammar component (see Remark 1 in Section 4).

In the altSAX framework [22], tree transformations are specified as non-circular attribute grammars, which are then translated to equivalent event-based applications using program composition techniques. The altSAX grammar component consists of four generic productions for parsing XML in binary-tree notation. In particular, there is a single production for all labeled nodes. If nodes with different labels are to be processed differently, users need to program case distinctions inside attribution functions. In contrast, the attribute grammars in our work and in [14] are based on extended regular tree grammars which provide more context information. At the very least, there is at least one production for each different node label.

Once a TransformX attribute grammar has been specified, the TransformX parser generator is responsible for its translation into correct Java code. In certain aspects, this is similar to existing parser generators for word languages [1] (as opposed to regular tree languages): The look and feel resembles that of the well-known compiler tool Yacc [18], while the processing model is related to that of LL(1) parser generators which perform a single pass over the input and do not construct a parse tree in main memory.

3 Preliminaries

We assume that *regular expressions* are constructed from a set of atomic symbols using the concatenation operator, union operator, and the Kleene star, denoted $.$, $\cup$, and *.

Regular Tree Grammars For the standard meaning of grammars and their derivations, we refer to [2] for basic and to [17] for extended grammars.

Definition 1 (ERTG) 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 grammar start symbol. □

ERTGs are a convenient way to specify a class of unranked labeled trees. Thus, they present a natural grammar mechanism for XML documents without attributes[2].

We introduce a syntactic macro *PCDATA* to describe character content of leaf nodes: Let $Char = \{c_1, \ldots, c_n\}$. As a shortcut, we define the regular expression macro $PCDATA := (\hat{c}_1 \cup \cdots \cup \hat{c}_n)^*$ using new nonterminals $\hat{c}_1, \ldots, \hat{c}_n$ and productions $\hat{c}_i ::= c_i(\epsilon)$ for each $1 \leq i \leq n$. *PCDATA* can be used just like a terminal on the right-hand sides of grammar productions. For sake of syntactic simplicity and brevity, we will consider *PCDATA* as a terminal as we already did in Example 1.

[2] Our data model of XML without attributes does not restrict the applicability of our work, since the attributes of a node can be modeled as special children which precede all other children.

Definition 2 An XML document is *well-formed* iff it conforms to an ERTG $G = (Nt, T, P, s)$ where for all productions $s ::= t(\rho)$, $t \in Tag$. □

A well-formed document contains at least one element (the root element) and has XML start and end tags properly nested within each other. Furthermore, the first symbol in the document must be the start tag of the root node. An XML document is *malformed* if it is not well-formed. The process of checking whether a document is well-formed w.r.t a specific ERTG is called *validation*.

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 $\bigcup \theta(nt)$.

Example 2 Consider the ERTG from Example 1. For regular expression $\rho = pub^*$ from the right-hand side of the bib-production, $\tau(\rho) = (\text{book} \cup \text{article})^*$. □

Each extended regular tree grammar can be alternatively considered as an extended context-free word grammar (CFG), which is obtained by simply rewriting each $tag(\rho)$ on the right-hand side of a production 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. [2]). 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 unambiguously refer to the *atomic symbols* in the regular expressions to be able to access or assign attributes. In attribute grammars, a straightforward solution [20] is to require for right-hand side regular expressions ρ that $\tau(\rho)$ is *unambiguous* [7].

Example 3 Consider a grammar with productions

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

The regular expression $(book_1 \cup book_2)^*$ is unambiguous, but $\tau((book_1 \cup book_2)^*) = (\text{book} \cup \text{book})^*$ is not. Therefore, when processing the tags of the children of the *bib* node, we cannot determine which of the two book-productions (with their possibly different attributions when we consider attribute grammars) are to be applied. □

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 $\tau(\rho)$ in the definition of TDLL(1) grammars. That is, we will require that $\tau(\rho)$ can be unambiguously parsed with just one symbol of lookahead.

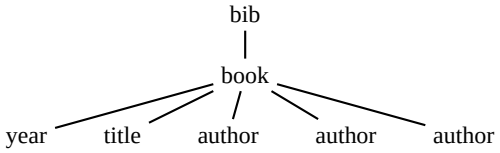


Figure 2: Document tree of Example 7.

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. The reverse of a marking (indicated by #) is obtained by dropping the subscripts.

Definition 3 ([9]) 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 4 Consider the regular expression $\rho = a^*.a$ and its marking $\rho' = a_1^*.a_2$. Let $u = a_1$, $x = a_2$, $v = \epsilon$, $y = a_1$, 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 4 ([17]) A *TDLL(1) grammar* is an extended regular tree grammar where $\tau(s)$ is unambiguous and in which for each regular expression ρ on the right-hand side of a production, $\tau(\rho)$ is one-unambiguous. □

Example 5 The grammar discussed in Example 1 is a TDLL(1) grammar. Since the grammar of Example 3 contains a regular expression ρ such that $\tau(\rho)$ is not even unambiguous, that grammar is not TDLL(1). □

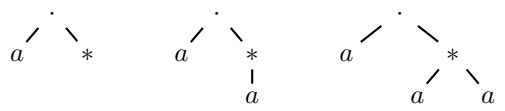
Example 6 DTDs are TDLL(1) grammars [14, 17]. □

For TDLL(1) grammars, the parse trees are simply the usual *document trees* associated with XML documents.

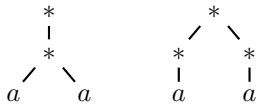
Example 7 The ERTG G from Example 1 is a TDLL(1) grammar. Thus, the XML document

⟨bib⟩
⟨book⟩ ⟨year/⟩⟨title/⟩⟨author/⟩⟨author/⟩⟨author/⟩ ⟨/book⟩
⟨/bib⟩

parses into the tree depicted in Figure 2. □



(a) $a, aa, aaa \in L(a.a^*)$.



(b) $aa \in L((a^*)^*)$.

Figure 3: Parse trees for words in regular languages.

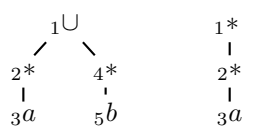
Strong One-Unambiguity TDLL(1) grammars allow us to use attributed extended regular tree grammars on XML streams. However, as we show in the following section, the ability to use attribution functions *inside* the regular expressions on the right-hand sides of productions will allow us to write many practical queries in a much more user-friendly fashion. As a prerequisite, we require for a right-hand side regular expression ρ that $\tau(\rho)$ is *strongly one-unambiguous*.

Let us briefly motivate this notion. Regular expressions provide a natural and intuitive way of assigning parse trees to words (see [14] for a formal definition). Specifying attribution functions inside a regular expression ρ corresponds to assigning attribution functions to the nodes in the parse trees for words in $L(\rho)$. In the setting of XML streams, this requires that all such parse trees can be unambiguously constructed with a lookahead of one token.

For instance, consider the parse trees for words $a, aa, aaa \in L(a.a^*)$ in Figure 3(a). These parse trees are unique, yet the parse tree for $aa \in L((a^*)^*)$, as shown in Figure 3(b), is not. While both regular expressions are one-unambiguous, $a.a^*$ is also *strongly* one-unambiguous. Let us now give a formal definition.

Intuitively, by a *bracketing* of a regular expression ρ, we refer to a *labeling of the nodes in the parse tree* of ρ using distinct indexes. We realize this by assigning the indexes in a depth-first left-to-right traversal of the parse tree. See Figure 4 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 that Σ and Γ are disjoint.

Definition 5 ([14]) Let ρ be a regular expression and let $\rho^{[]}$ 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 $u\alpha, u\beta \in L(\rho^{[]})$. □



(a) $a^* \cup b^*$. (b) $(a^*)^*$.

Figure 4: Parse trees of regular expressions.

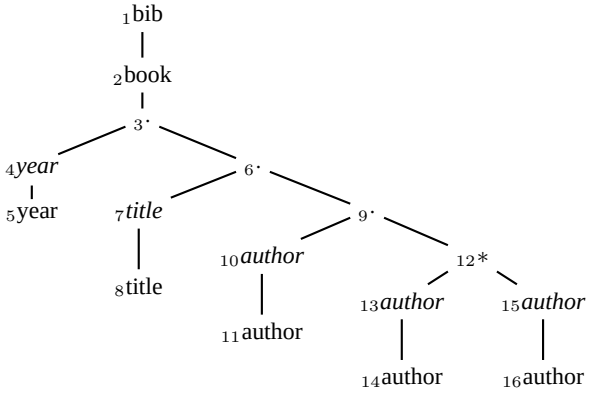


Figure 5: Extended parse tree of Example 9.

Example 8 Consider the regular expression $\rho = (a^*)^*$ with bracketing $\rho^{[]} = [_1.([_2.([_3.a.]_3)^*.]_2)^*.]_1$ derived from the parse tree in Figure 4(b).

ρ is one-unambiguous, but not strongly one-unambiguous: For the word $aa \in L(\rho)$, there are bracketings $[_1.[_2.[_3.a.]_3.[_3.a.]_3.]_2.]_1 \in L(\rho^{[]})$ and $[_1.[_2.[_3.a.]_3.]_2.[_2.[_3.a.]_3.]_2.]_1 \in L(\rho^{[]})$, so $u = [_1.[_2.[_3.a$, $\alpha =]_3.[_3$, $\beta =]_3.]_2.[_2.[_3$, $x = a$, and $v = w =]_3.]_2.]_1$. □

While DTDs are only required to have one-unambiguous content models [8] (i.e., right-hand side regular expressions in productions), we believe that most practical DTDs only use strongly one-unambiguous regular expressions in their productions [14].

If a production in a TDLL(1) grammar has a strongly one-unambiguous regular expression ρ on its right-hand side, we may incorporate the parse trees for words in $L(\rho)$ in the document parse tree, as they may also be constructed online. We refer to such parse trees as *extended parse trees* and introduce the following notation: Let $G = (Nt, T, P, s)$ be a TDLL(1) grammar. Let $P' \subseteq P$ be a set of productions with strongly one-unambiguous regular expressions on their right-hand sides and let T be an input document. Then $\mathcal{P}_G(T, P')$ is an extended parse tree for T which incorporates the parse trees for sibling nodes that are words in those regular languages $L(\rho)$ where ρ is a right-hand side regular expression in a production in P'. Obviously, for $P' = \emptyset$, $\mathcal{P}_G(T, P') = T$.

Example 9 Consider the ERTG G from Example 1 and the input document T from Example 7. Let p^{book} denote the book-production. The extended parse tree $\mathcal{P}_G(T, \{p^{\text{book}}\})$ is shown in Figure 5, where nodes associated with nonterminals are set in italics. Here, we assume that the operation "." associates to the right and that the production "*pub* ::= article$(\mathit{year}.\mathit{title}.\mathit{author}.\mathit{author}^*)$" of G is thus equivalent to "*pub* ::= article$(\mathit{year}.(\mathit{title}.(\mathit{author}.(\mathit{author}^*))))$". □

Notation for Classes and Types We assume the usual notions of classes, objects, and types, and further attributes and methods in object-oriented programming languages. In particular, we introduce the following notation.

Let C be a set of *class names* and let *att* be a set of names for *attributes*. A special constant *nil* represents the undefined value. If $\tau_1, \ldots, \tau_n$ are types and $A_1, \ldots, A_n$ distinct attribute names, then $[A_1 : \tau_1, \ldots, A_n : \tau_n]$ is a (tuple) type.

We assume an infinite set *meth* of method names. A method consists of a name, a signature, and an implementation (also called body). For a method name m, a signature of m is an expression of the form $m : c \times \tau_1 \times \cdots \times \tau_{n-1} \rightarrow \tau_n$ where c is a class name in C and each τ_i is a type.We assume that all classes provide a method *clone* which returns a new object of the same type and with the same value.

4 TransformX Attribute Grammars

We are now in the position to define the syntax and semantics of TransformX attribute grammars. Having laid these foundations, we show how users can specify Java-based TransformX attribute grammars in practice.

4.1 Abstract Definition

We define TransformX attribute grammars as TDLL(1) grammars with *attribution functions* added to productions. The key principle in the following syntax definition is that attribution functions may only be inserted such that we can unambiguously determine which attribution functions to invoke next when reading a specific token from the XML input stream.

Definition 6 (Syntax) Let $\tau_1, \ldots, \tau_k$ be types and $A_1, \ldots, A_k$ be distinct attribute names. Let *Att* denote the tuple type $[A_1 : \tau_1, \ldots, A_k : \tau_k]$.

Let $c \in C$ be a class name. Let $F_{\$[}, F_{\$]} \subseteq$ *meth* be disjoint sets of method names where $f_{\$[} \in F_{\$[}$ denotes a *first-visit attribution function* with signature

$$f_{\$[} : c \times \tau_1 \times \cdots \times \tau_k \rightarrow \mathit{Att}$$

and where $f_{\$]} \in F_{\$]}$ denotes a *second-visit attribution function* with signature

$$f_{\$]} : c \times \tau_1 \times \ldots \tau_k \times \tau_1 \times \cdots \times \tau_k \rightarrow \mathit{Att}.$$

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

$$\begin{array}{rcl} \mathit{aregex} & ::= & (\text{“}\{\text{”}\ F_{\$[}\ \text{“}\}\text{”})?\ \ \mathit{aregex}_0\ \ (\text{“}\{\text{”}\ F_{\$]}\ \text{“}\}\text{”})? \\ \mathit{aregex}_0 & ::= & \Sigma \mid \mathit{aregex}\ \text{“.”}\ \mathit{aregex} \mid \\ & & \mathit{aregex}\ \text{“}\cup\text{”}\ \mathit{aregex} \mid \mathit{aregex}\ \text{“}^*\text{”} \end{array}$$

A *TransformX attribute grammar* is an attributed extended regular tree grammar $G = (\mathit{Nt}, T, P, s)$ with nonterminals *Nt*, grammar start symbol s, terminals $T = \mathit{Tag} \cup \mathit{Char}$, and productions in P where each production is of one of the four forms

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

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

with $\mathit{nt} \in \mathit{Nt}$, $t \in T$, $f_{\$[} \in F_{\$[}$, $f_{\$]} \in F_{\$]}$, and if $t \in \mathit{Char}$ then $\alpha = \epsilon$, and if $t \in \mathit{Tag}$, then α is either

1. ϵ,
2. a regular expression over *Nt* such that $\tau(\alpha)$ is one-unambiguous, or,
3. an attributed regular expression over symbols *Nt* (containing at least one attribution function), such that the following holds: For the regular expression ρ obtained from α by removing the attributions (enclosed in curly braces), $\tau(\rho)$ is *strongly* one-unambiguous. □

Thus, given a TDLL(1) grammar, we may add attribution functions to a production depending on its right-hand side. In cases (1) and (2) of the above definition, attribution functions may be inserted only at the beginning and end of the right-hand side. In case of a strongly one-unambiguous regular expression on the right-hand side of a production (3), we may also specify attribution functions *inside* the regular expression.

Remark 1 While the above definition resembles the syntax of XSAGs [14], the syntax definition of TransformX attribute grammars is in fact more general (and the class of transformations possible with TransformX attribute grammars is much more powerful, as discussed in Section 2): With XSAGs, we distinguish between bXSAGs (where only cases (1) and (2) are allowed) and yXSAGs (where only cases (1) and (3) are allowed). While the latter are more convenient to use, they require a grammar component stricter than TDLL(1). With TransformX attribute grammars, we no longer make this distinction. Instead, we solely consider TDLL(1) grammars and only allow attributions inside those regular expressions which are strongly one-unambiguous. □

As the grammar components of TransformX attribute grammars are TDLL(1), we can validate input documents one token at a time. In particular, the (extended) parse trees of input documents may be unambiguously constructed online. Yet at no time during the evaluation of TransformX attribute grammars will it be necessary to actually maintain the entire (extended) parse trees in main memory. Rather, it is sufficient to keep the “path” from the root to the current node in main memory.

In the following, we will be somewhat imprecise and talk of *attribution functions in $F_{\$[}$ and $F_{\$]}$* when we actually refer to the method names in $F_{\$[}$ and $F_{\$]}$.

Let $G = (\mathit{Nt}, T, P, s)$ be a TransformX attribute grammar and let P' be the subset of productions with attributed regular expressions on their right-hand sides. We evaluate

G on an XML input tree T by assigning attribution functions from $F_{\$[}$ and $F_{\$]}$ to the nodes in the extended parse tree $\mathcal{P}_G(T, P')$, as specified by the grammar. We then refer to such parse trees as *attributed parse trees* and write $\mathcal{P}_G(T)$ as a shortcut for $\mathcal{P}_G(T, P')$.

Where the attribute grammar does not explicitly state attribution functions, we assume default attribution functions $f^d_{\$[}$ and $f^d_{\$]}$ which are implemented as follows:

Att $f^d_{\$[}(\tau_1\ a_1,\ \ldots,\ \tau_k\ a_k)\{$
  return new $[A_1 = a_1, \ldots, A_k = a_k]$;
$\}$

Att $f^d_{\$]}(\tau_1\ a_1,\ \ldots,\ \tau_k\ a_k,\ \tau_1\ b_1,\ \ldots,\ \tau_k\ b_k)\{$
  return new $[A_1 = b_1, \ldots, A_k = b_k]$;
$\}$

Example 10 From an abstract point of view, the TransformX attribute grammar of Example 1 may be written as $G = (Nt, T, P, s)$ with attribution functions

$$F_{\$[} = \{f^d_{\$[}, f^1_{\$[}, f^2_{\$[}, f^4_{\$[}, f^7_{\$[}, f^9_{\$[}\},\ F_{\$]} = \{f^d_{\$]}, f^1_{\$]}, f^2_{\$]}\},$$

and the attributed productions

$$\begin{aligned} \textit{bib} \quad &::= \quad \{f^1_{\$[}\}\ \text{bib}\big(\textit{pub}^*\big)\ \{f^1_{\$]}\} \\ \textit{pub} \quad &::= \quad \{f^2_{\$[}\}\ \text{book}\big(\big(\{f^4_{\$[}\}\ \textit{year}\big).\big(\{f^7_{\$[}\}\ \textit{title}\big). \\ &\qquad\qquad \big(\{f^9_{\$[}\}\ \textit{author.author}^*\big)\big)\ \{f^2_{\$]}\} \end{aligned}$$

(with the other productions just as in Example 1). Assume the input document T from Example 9. In the above attribute grammar, the book-production (denoted p^{book}) is the only production with an attributed regular expression on its right-hand side.

The assignment of attribution functions to the nodes of the extended parse tree $\mathcal{P}_G(T, \{p^{\text{book}}\})$ in Figure 5 is straightforward: For instance, node 1 is assigned $f^1_{\$[}$ and $f^1_{\$]}$. Node 4 is assigned $f^4_{\$[}$ and the default second-visit attribution function $f^d_{\$]}$, while node 5 is assigned both default attribution functions $f^d_{\$[}$ and $f^d_{\$]}$. Further, $f^9_{\$[}$ and $f^d_{\$]}$ are assigned to node 9. □

TransformX attribute grammars are defined as L-attributed grammars, i.e., attribute grammars whose attributes are evaluated in a single depth-first left-to-right traversal of the document tree [1]. Each node v is assigned two attribution functions $f^v_{\$[}$ and $f^v_{\$]}$. During the traversal, each node v is visited twice (the visits are referred to by $\$[$ and $\$]$), first from the preceding sibling or the parent of v (if v has no preceding sibling) and a second time on returning from the rightmost child of v. Accordingly, we evaluate $f^v_{\$[}$ in the first visit, and $f^v_{\$]}$ in the second visit.

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 7 (Semantics) Let $\tau_1, \ldots, \tau_k$ be types and $A_1, \ldots, A_k$ be distinct attribute names. Let *Att* denote the type $[A_1 : \tau_1, \ldots, A_k : \tau_k]$.

Let $F_{\$[}, F_{\$]} \subseteq$ *meth* be the first- and second-visit attribution functions respectively. Then we evaluate a TransformX attribute grammar G on an attributed parse tree $\mathcal{P}_G(T)$ as follows. In a depth-first left-to-right traversal of $\mathcal{P}_G(T)$, we compute for each attribute A_i (with $i = 1, \ldots, k$) and each node v, the four assignments $(a_i)^v_{\$[.in}$, $(a_i)^v_{\$[.out}$, $(a_i)^v_{\$].in}$, and $(a_i)^v_{\$].out}$ (inductively) as follows.

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

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

In the first visit to node v, we evaluate

$$[A_1 = (a_1)^v_{\$[.out}, \ldots, A_k = (a_k)^v_{\$[.out}] = f^v_{\$[}\big((a_1)^v_{\$[.in}, \ldots, (a_k)^v_{\$[.in}\big)$$

and in the second visit to v, we compute

$$[A_1 = (a_1)^v_{\$].out}, \ldots, A_k = (a_k)^v_{\$].out}] = f^v_{\$]}\big((a_1)^v_{\$[.out}, \ldots, (a_k)^v_{\$[.out}, (a_1)^v_{\$].in}, \ldots, (a_k)^v_{\$].in}\big)$$

In case $f^v_{\$[}$ or $f^v_{\$]}$ exit the program execution, the evaluation terminates and the input is rejected. □

Note that a TransformX attribute grammar may reject the input in two different ways, namely if it is not valid with respect to the grammar component or if an attribution function exits the program execution.

4.2 Java-based TransformX Attribute Grammars

In the tradition of parser generators such as Yacc, users define Java-based TransformX attribute grammars in a single input file, as exemplified in Figures 1 and 6. We defer the handling of I/O until after we have described how to read and write basic TransformX attribute grammars.

Specification Let $\bar{\tau}_1, \ldots, \bar{\tau}_n$ be types and let $B_1, \ldots, B_n$ be distinct attribute names. Further, let F_{aux} be a set of method names. Then a TransformX attribute grammar with

- class name $c \in C$,
- types $\tau_1, \ldots, \tau_k$, distinct attribute names $A_1, \ldots, A_k$, and type *Att* $= [A_1 : \tau_1, \ldots, A_k : \tau_k]$,
- attribution functions in $F_{\$[}$ and $F_{\$]}$, where the bodies of attribution functions may refer to $B_1, \ldots, B_n$ and methods in F_{aux}, so that users may introduce auxiliary attributes and methods. Further,
- an attributed TDLL(1) grammar $G = (Nt, T, P, s)$,

is encoded as a Java-based TransformX attribute grammar in a single specification file, consisting of three sections:

(a) In the *definition section*, we set up the execution environment and declare $A_1, \ldots, A_k$:

- "**%class** c" specifies the name of the generated class
- Import statements and class definitions are stated in a literal Java block, enclosed in braces "**%**{ **...** **%**}".
- Keyword "**%att**" precedes the declarations and initializations of $A_1, \ldots, A_k$.
- The nonterminal from the first production in the rules section is assumed to be the grammar start symbol. It may be redefined by the statement "**%start** s".

(b) The *rules section* contains the attributed productions. Consider a first-visit attribution function $f^v_{\$[} \in F_{\$[}$ assigned to a node v in the attributed parse tree:

Att $f_{\$[}(\tau_1\ \$[.A_1, \ldots, \tau_k\ \$[.A_k)\{$
$\ldots$*body*$\ldots$
return new $[A_1 = \$[.A_1, \ldots, A_k = \$[.A_k]$;
$\}$

Within the body of $f^v_{\$[}$, we write "$\$[.A_i$" to access the corresponding input parameter. Likewise, consider a second-visit attribution function $f^v_{\$]} \in F_{\$[}$, assigned to v:

Att $f^v_{\$]}(\tau_1\ \$[.A_1, \ldots, \tau_k\ \$[.A_k,\ \ \tau_1\ \$].A_1, \ldots, \tau_k\ \$].A_k)\{$
$\ldots$*body*$\ldots$
return new $[A_1 = \$].A_1, \ldots, A_k = \$].A_k]$;
$\}$

Within the body of $f^v_{\$]}$, we may access two versions of A_i according to Definition 7, which we distinguish using prefixes $\$[$ and $\$]$: "$\$[.A_i$" yields the value of A_i as computed by $f^v_{\$[}$. By "$\$].A_i$", we access the value of A_i as computed by the attribution functions assigned to the rightmost child node of v.

In place of the method names for attribution functions in Definition 6, we insert the implementations inside the productions in the definition section. In particular, the bodies are enclosed in curly braces and declared without return statements.

(c) The *class member section* contains the declarations of the attribute names $B_1, \ldots, B_n$ together with their types $\bar{\tau}_1, \ldots, \bar{\tau}_n$, and the methods with names in F_{aux}.

As Yacc users will have noticed, we made an effort to adopt the familiar Yacc notation as far as possible.

In a TransformX attribute grammar, we distinguish between two kinds of attributes:

1. The auxiliary attributes (denoted $B_1, \ldots, B_n$ above) declared in the class member section, and
2. the attributes from the actual TransformX attribute grammar, as declared in the definition section (see $A_1, \ldots, A_k$ above).

```
%class SectionProcessor
%{
   class IntAtt {
      IntAtt( int v )          { value = v; }
      public IntAtt clone()    { return new IntAtt(value); }
      public int value;        }
%}
%att IntAtt count = new IntAtt(0);
%start doc
%%
   sec   ::=   { $[.count.value++;  /* f^sec_$[ */ }
               section(sec* ∪ par*)
               { int before = $[.count.value;  /* f^sec_$] */
                 int after  = $].count.value;
                 if ( is_odd(after - before) ) odd++;  }
   par   ::=   paragraph(ε)
   doc   ::=   document(sec*)
               { real r = odd / $].count.value;  /* f^doc_$] */
                 System.out.println("Result=" + r +"%");  }
%%
private int odd = 0;
private boolean is_odd( int x ) { return (x % 2 == 0); }
```

Figure 6: TransformX attribute grammar of Example 11.

In general, the attributes of (1) encode a global state in the parsing process, e.g., how many books have been encountered in the XML stream so far. In contrast, the attributes of (2) are best employed to transfer information between the first- and second-visit attribution functions assigned to the same node in the attributed parse tree.

The following example illustrates how the two kinds of attributes are best applied. It also aims at demonstrating the various syntactical constructs, so it is not necessarily the shortest and most elegant encoding.

Example 11 For documents conforming to the recursive TDLL(1) grammar $G = (Nt, T, P, s)$ with productions

$$\begin{aligned} doc &::= \text{document}(sec^*) \\ sec &::= \text{section}(sec^* \cup PCDATA) \end{aligned}$$

we want to compute the percentage of section-nodes with an odd number of section-nodes in their subtrees.

Consider the document tree in Figure 7, where the subtree rooted at node 2 contains an odd number of section-nodes (exactly 3), while the subtree rooted at node 6 contains zero and thus an even number of section-nodes.

The Java-based TransformX attribute grammar in Figure 6 specifies this computation: In the definition section,

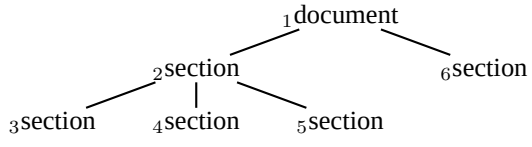


Figure 7: Document tree T of Example 11.

input	node		comments
⟨document⟩	1	$f^{d}_{\$[}(0) \to 0$	odd = 0
⟨section⟩	2	$f^{sec}_{\$[}(0) \to 1$	
⟨section⟩	3	$f^{sec}_{\$[}(1) \to 2$	
⟨/section⟩	3	$f^{sec}_{\$]}(2,2) \to 2$	
⟨section⟩	4	$f^{sec}_{\$[}(2) \to 3$	
⟨/section⟩	4	$f^{sec}_{\$]}(3,3) \to 3$	
⟨section⟩	5	$f^{sec}_{\$[}(3) \to 4$	
⟨/section⟩	5	$f^{sec}_{\$]}(4,4) \to 4$	
⟨/section⟩	2	$f^{sec}_{\$]}(1,4) \to 4$	odd = 1
⟨section⟩	6	$f^{sec}_{\$[}(4) \to 5$	
⟨/section⟩	6	$f^{sec}_{\$]}(5,5) \to 5$	
⟨/document⟩	1	$f^{doc}_{\$]}(0,5) \to 5$	"Result = 0.2%"

Figure 8: Run of Example 11

a new class *IntAtt* holding integer values is defined, and an auxiliary attribute "count" of type *IntAtt* is declared and initialized to zero.

In the class member section, we declare an auxiliary attribute "odd" which states how many section-nodes with an odd number of sections in their subtrees have been seen so far. By declaring "odd" private, we can control the access to this class member.

The rules section contains the implementations of attribution functions $f^{sec}_{\$[}$, $f^{sec}_{\$]}$, and $f^{doc}_{\$]}$: Attribute "count" is incremented whenever a tag "⟨section⟩" is read. Upon reading "⟨/section⟩", the number of section nodes in the subtree below is computed by subtracting "$[.count.value" (the number of section-nodes before processing the subtree) from "$].count.value" (the number of section-nodes after processing the subtree). If the result is an odd number, then the auxiliary attribute "count" is updated accordingly. When the closing tag of the root node is read, the total number of nodes (contained in "$].count.value") is used to compute the result.

As this TransformX attribute grammar does not contain any productions with attributed regular expressions, the extended parse tree is the same as the document tree. Consequently, node 1 in Figure 7 is assigned the attribution functions $f^{d}_{\$[}$ and $f^{doc}_{\$]}$. All other nodes are assigned the attribution functions $f^{sec}_{\$[}$ and $f^{sec}_{\$]}$.

In Figure 8, we trace the execution of the above TransformX attribute grammar on this input document. The first column shows the current XML tag being read. The second column identifies the current node in the depth-first left-to-right traversal of the attributed parse tree. In the third column, we state which attribution function is being evaluated, together with the values of its input- and output-parameters. □

I/O with TransformX Attribute Grammars Typical transformations of XML streams will involve reading parts of the input. So-called *sink streams*, either intended to create output or to serve as internal buffers, are declared and instantiated by the "**%echo**" statement in the definition section.

All sink streams are derived from abstract class *Writer* as specified by the Java API [24], and thus provide methods *write*, *flush*, and *close*. Users may add their own implementations of sink streams. The TransformX framework already contains two customized classes, as used in Example 1:

- Class *TXWriter*, a subclass of *OutputStreamWriter*, is intended for copying data to output streams, e.g., to standard output or files.
- Class *TXBuffer*, a subclass of *StringWriter*, is used to buffer parts of the input in in-memory string buffers. TXBuffer provides method *clear()* for discarding any buffered contents.

Let o be a sink stream and let $f^{v}_{\$[}$ be a first-visit attribution function assigned to a node v in an attributed parse tree. If the implementation of $f^{v}_{\$[}$ contains the statement "o.write($this)", the subtree rooted at v will be copied to o. Yet if a first-visit $f^{w}_{\$[}$ assigned to a descendant node w of v contains the statement "o.omit($this)", then the subtree rooted at w will not be copied.

Example 12 Changing the book-production in the TransformX attribute grammar of Example 1 to

pub ::= { output.write($this); }
book(*year*.({ output.omit($this); } *title*).*author*.*author**)

will output books together with their XML start and end tags, year and author children, but without title. □

The echo-mechanism may be implemented by introducing additional attributes in the definition section which signal whether a subtree is to be output or not. Due to space restrictions, we refer to [14, 23] for details on the implementation of the analogous macros *ECHO* and *ECHO_OFF*.

5 The TransformX Parser Generator

The TransformX parser generator translates TransformX attribute grammars into the source code of a Java class which encapsulates the attribution functions, user-defined attributes and auxiliary methods, and performs the transformation of the XML input stream. Compilation of the source code with the Java compiler yields the executable classfiles. In the following, we discuss the translation to source code and its time complexity.

The evaluation of a TransformX attribute grammar G on an input document T is based on a separation of concerns. Accordingly, we define two modules:

- The *validator module* validates the XML input stream against the grammar component of G and outputs the sequence of attribution functions as encountered in the depth-first left-to-right traversal of the attributed parse tree.
- The *evaluator module* invokes the attribution functions as output by the validator. It stores the result computed by a first-visit attribution function on a

stack so that they are accessible to the second-visit attribution function which is assigned to the same node in the attributed parse tree.

Note that the validator and evaluator each maintain their own stack. This allows for optimizations, as outlined in our discussion of future work.

We assume the usual notion of deterministic pushdown transducers (*DPDT*s) [2] as deterministic pushdown automata with output.

Definition 8 Let G be a TransformX attribute grammar with grammar component G' and let T be an XML document tree. A *validator* $\mathcal{V}(G)$ is a DPDT which, if $T \in L(G')$, outputs the method names of the attribution functions as encountered on the depth-first left-to-right traversal of the attributed parse tree of T. □

We construct a validator from a TransformX attribute grammar $G = (Nt, T, P, s)$ in two steps:

1. For each production $p \in P$, we construct a deterministic finite-state transducer (DFT) [2] $\mathcal{A}^p$ which recognizes the regular language defined by the right-hand side of p and outputs the corresponding attribution functions at the same time.

   For productions with ϵ or one-unambiguous regular expressions on their right-hand side, the construction of $\mathcal{A}^p$ is straightforward. Let us briefly consider productions with attributed regular expressions.

   Given a regular expression β, let $\odot$ be a special endmarker symbol that does not occur in β. As shown in [14, 23], if a regular expression β is strongly one-unambiguous, then there is a DFT $\mathcal{A}^{[]}(\beta)$ which recognizes $L(\beta.\odot)$ and outputs the bracketing of word w for input $w.\odot \in L(\beta.\odot)$. Further, there is an $O(n^3)$ algorithm that checks whether β is strongly one-unambiguous and if so, outputs $\mathcal{A}^{[]}(\beta)$.

   By Definition 6 we know that for an attributed regular expression α on the right-hand side of a production, we obtain a regular expression ρ over nonterminals by ignoring the attribution functions in α. Further, $\tau(\rho)$ is strongly one-unambiguous. Obviously, this also implies the strong one-unambiguity of ρ.

   So given α and ρ, we may adapt $\mathcal{A}^{[]}(\rho)$ so that it outputs the method names of the attribution functions assigned to α instead of the bracketing, with the default-attribution functions in places where no attribution functions have been specified by the user.

2. The transitions of all DFTs in $\{\mathcal{A}^p \mid p \in P\}$ are then encoded into the transitions of a single DPDT. Let p be the current production. When reading a start tag $\langle t \rangle$, we store the current state of $\mathcal{A}^p$ on the DPDT stack and start the simulation of the DFT which corresponds to the new production. Consequently, when reading the matching end tag $\langle /t \rangle$, we resume the simulation of the previous DFT $\mathcal{A}^p$ by retrieving its state from the stack. Thus, we use the stack to switch between the DFTs, or rather, the productions they represent.

```
void evaluate((F_$[ ∪ F_$]) f)
{
  if (pred = 1 and f ∈ F_$[) {
    Att A := S.top().clone();
    S.push( f(A) );
    pred := 1;
  } else if (pred = 1 and f ∈ F_$]) {
    Att A := S.pop();
    Att B := A.clone();
    S.push( f(A, B) );
    pred := 2;
  } else if (pred = 2 and f ∈ F_$[) {
    Att A := S.pop();
    S.push( f(A) );
    pred := 1;
  } else if (pred = 2 and f ∈ F_$]) {
    Att B := S.pop();
    Att A := S.pop();
    S.push( f(A, B) );
    pred := 2;
  }
}
```

Figure 9: Method *evaluate*.

By simulating the DPDT on an input document, the document is validated against the grammar component of G. At the same time, the validator outputs the attribution functions in the correct order. We refer to [23] for details of this construction.

Proposition 1 *For a TransformX attribute grammar G, a validator $\mathcal{V}(G)$ can be constructed in time $O(|G|^3)$.*

The computationally most expensive step in the construction of a validator is the derivation of DFTs from regular expressions. For a one-unambiguous regular expression ρ, such a DFT may be constructed in quadratic time in the size of ρ [9]. As mentioned above, the DFT construction for an attributed regular expression ρ requires cubic time in the size of ρ [14, 23]. Thus, the time complexity for the construction of $\mathcal{V}(G)$ is in $O(|G|^3)$. In fact, the translation is only cubic in the sizes of the regular expressions in the productions of G, and linear in the size of G. For typical grammars, we may expect these regular expressions to be small.

Obviously, if a TransformX attribute grammar G does not contain any attributed regular expressions, then the validator can be constructed in time $O(|G|^2)$.

We assume the common notion of a stack with methods *top*, *pop*, and *push*.

Definition 9 Let $\tau_1, \ldots, \tau_k$ be types, let $A_1, \ldots, A_k$ be distinct attribute names, and let $Att = [A_1 : \tau_1, \ldots, A_k : \tau_k]$. Let G be a TransformX attribute grammar with attribution functions in $F_{\$[}$ and $F_{\$]}$.

An *evaluator* $\mathcal{E}(G)$ is a class with the following properties:

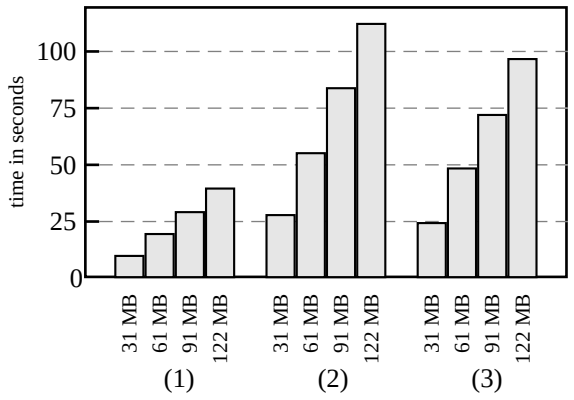


Figure 10: Runtime for datasets of various sizes.

1. Attribute S is a stack holding tuples of attributes. Attribute *pred* of type integer signals whether the attribution function last processed was in $F_{\$[}$ (if $pred = 1$), or in $F_{\$[}$ (if $pred = 2$).
2. Initially, $pred = 2$ and the stack holds the tuple $[A_1 = nil, \ldots, A_k = nil]$.
3. Method *evaluate* invokes one attribution function $f \in F_{\$[} \cup F_{\$]}$ at a time and stores attribute values on the stack so that a second-visit attribution function can access the results computed by the first-visit attribution function that is assigned to the same node in the attributed parse tree. The pseudo code for this method is shown in Figure 9. □

As the evaluator follows directly from its definition, it may be constructed in constant time. So by Proposition 1, the TransformX parser generator can translate a given TransformX attribute grammar G to (Java) source code in time $O(|G|^3)$.

6 Implementation and Experiments

We implemented the TransformX framework in C++ with *gcc* version 3.2. The generated Java class uses the SAX parser provided by the Java API and is compiled and executed with *Sun*'s JDK 1.4.1 [24].

Runtime is averaged over five runs and measured in seconds. It does not include the translation of TransformX attribute grammars, as these values were negligible.

Our test data is an XML bibliography conforming to the ERTG of Example 1 with an equal number of books and articles. We conducted the following experiments:

1. Validate the input against the ERTG, i.e., evaluate a TransformX attribute grammar with only default attribution functions,
2. output the complete input, and
3. evaluate the TransformX attribute grammar specified in Figure 1.

We execute these transformations on documents of different sizes. The results are shown in Figure 10. For all three transformations, the main memory consumption remains constant at 12 MB, which is about the amount of memory it takes for the JVM to run. Experiment (1) demonstrates the minimum time to parse and validate the input, while transformations (2) and (3) require more time, as additional processing steps are involved. This effect is also visible in the throughput achieved: Transformation (1) reaches 3.05 $\frac{\text{MB}}{\text{sec}}$, while transformations (2) and (3) reach a throughput of 1.09 $\frac{\text{MB}}{\text{sec}}$ and 1.26 $\frac{\text{MB}}{\text{sec}}$.

We refrain from benchmarking the TransformX framework against other systems, as in our framework, performance is entirely in the responsibility of the programmer: Knowledgable programmers can construct the fastest possible Java transformations. The TransformX framework facilitates this task by providing the grammar-based control over the data stream characteristics.

7 Conclusion and Future Work

We have motivated the benefits of syntax-directed transformations of XML streams. We have described the TransformX attribute grammars and parser generator as a tool for conveniently developing Java applications for XML stream processing.

Our next step will be to optimize the evaluation of TransformX attribute grammars by reducing the number of stack operations performed by the evaluator: The general idea is based on the observation that most practical TransformX attribute grammars contain only few user-defined attribution functions. Typically, the majority of nodes in attributed parse trees is assigned default attribution functions, especially if productions contain attributed regular expressions.

For an attributed parse tree P, we construct a tree minor P' by removing those nodes which are assigned "non-effective" attribution functions. To name an obvious example, consider nodes which are assigned both default attribution functions $f^d_{\$[}$ and $f^d_{\$]}$. In more interesting cases, the result of applying a first-visit attribution function does not need to be stored on the evaluator stack, because it is not accessed by succeeding attribution functions. A modified program analysis taking into account the semantics of TransformX attribute grammars may identify such cases.

If the evaluator invokes the attribution functions encountered in the traversal of P, but only stores the results on the stack for those attribution functions also encountered in the traversal of P', fewer stack operations are performed. This improves runtime and may even reduce main memory consumption, while the transformation of the XML stream remains equivalent. In first experiments, this optimization improves the runtime of transformation (3) in Section 6 by 15%. We plan to further explore this approach in our future work.

Acknowledgments

We thank Christoph Koch, Susumu Nishimura, Bernhard Stegmaier, and the anonymous reviewers of PLAN-X 2005 for their many helpful suggestions.

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*. Prentice-Hall, 1972.

[3] M. Altinel and M. Franklin. "Efficient Filtering of XML Documents for Selective Dissemination of Information". In *Proc. VLDB 2000*, pages 53–64.

[4] Apache XML Project. "Xalan-Java Version 2.0". http://xml.apache.org/xalan-j.

[5] M. Benedikt, C. Y. Chan, W. Fan, J. Freire, and R. Rastogi. "Capturing both Types and Constraints in Data Integration". In *Proc. SIGMOD 2003*, pages 277–288, 2003.

[6] P. Bohannon, P. Buneman, B. Choi, and W. Fan. "Incremental Evaluation of Schema-Directed XML Publishing". In *Proc. SIGMOD 2004*, pages 503–514, 2004.

[7] R. Book, S. Even, S. Greibach, and G. Ott. "Ambiguity in Graphs and Expressions". *IEEE Transactions on Computers*, **20**(2):149–153, Feb. 1971.

[8] T. Bray, J. Paoli, and C. Sperberg-McQueen. "Extensible Markup Language (XML) 1.0". Technical report, W3C, Feb. 1998.

[9] A. Brüggemann-Klein and D. Wood. "One-Unambiguous Regular Languages". *Information and Computation*, **142**(2):182–206, 1998.

[10] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. "Efficient Filtering of XML Documents with XPath Expressions". In *Proc. ICDE 2002*, pages 235–244, 2002.

[11] P. Cimprich, O. Becker, C. Nentwich, H. Jiroušek, M. Batsis, P. Brown, and M. Kay. "Streaming Transformations for XML (STX)", 2003. http://stx.sourceforge.net/documents/.

[12] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. "Processing XML Streams with Deterministic Automata". In *Proc. ICDT'03*, pages 173–189, 2003.

[13] A. Gupta and D. Suciu. "Stream Processing of XPath Queries with Predicates". In *Proc. SIGMOD 2003*, pages 419–430, 2003.

[14] C. Koch and S. Scherzinger. "Attribute Grammars for Scalable Query Processing on XML Streams". In *Proc. DBPL 2003*, pages 233–256, 2003.

[15] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. "FluXQuery: An Optimizing XQuery Processor for Streaming XML Data". In *Proc. VLDB 2004*, pages 1309–1312, 2004. Demonstration.

[16] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. "Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams". In *Proc. VLDB 2004*, pages 228–239, 2004.

[17] D. Lee, M. Mani, and M. Murata. "Reasoning about XML Schema Languages using Formal Language Theory". Technical Report RJ 10197 Log 95071, IBM Research, 2000. http://citeseer.ist.psu.edu/lee00reasoning.html.

[18] J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, Inc., 1992. Second Edition.

[19] A. Marian and J. Siméon. "Projecting XML Documents". In *Proc. VLDB 2003*, pages 213–224, 2003.

[20] F. Neven. "Extensions of Attribute Grammars for Structured Document Queries". In *Proc. DBPL 1999*, pages 99–116, 1999.

[21] 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.

[22] S. Nishimura and K. Nakano. "XML Stream Transformer Generation Through Program Composition and Dependency Analysis". *Science of Computer Programming*, 54(2–3):257–290, 2005.

[23] S. Scherzinger. "Scalable Query Processing on XML Streams". Diploma thesis, University of Passau, 2004. http://wit.at/people/scherzinger/thesis.pdf.

[24] Sun Microsystems. "JavaTM 2 Platform Standard Edition v1.4.1". http://java.sun.com/.

[25] World Wide Web Consortium. "XML Query (XQuery)". http://www.w3c.org/XML/query/.

[26] World Wide Web Consortium. "XSL Transformations (XSLT)". http://www.w3.org/TR/xslt/.