Institut für Informatik
der Technischen Universität München
Lehrstuhl für Informatik II

Efficient XML Processing with Tree Automata

Alexandru Berlea

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Alfons Kemper
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Helmut Seidl
	2.	Univ.-Prof. Dr. Klaus U. Schulz, Ludwig-Maximilians-Universität München

Die Dissertation wurde am 29.06.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.10.2005 angenommen.

Acknowledgments

This work would not exist without the beneficial influence of many people. I am very thankful to all of them. I use this opportunity to express my profound gratitude to a few of them: Helmut Seidl, for offering me the possibility to perform this work and for his steady guidance and warmheartedness; Pete Bankhead, for his meticulous proofreading and his detailed suggestions which helped improving the presentation of this work; Neeraj Verma and Peter Ziewer for their helpful comments and suggestions; my colleagues in Trier and München, for the pleasant work atmosphere and their kind helpfulness; my family for their support and confidence. I owe a lot to Maria for her love and care.

Abstract

An essential task for XML applications is querying, i.e. identifying locations in the input data with certain specified properties. The present work considers an expressive XML query language and provides efficient algorithms for its implementation. The techniques introduced are applied in the XML querying tool Fxgrep.

Some XML documents may be too large to be built in memory. For these, special algorithms have to be provided. The same holds true for XML data which must be processed while being received, rather than being completely available in advance. In such cases XML data is seen as a stream of events to which the application has to react in order to perform the desired processing. Fxgrep provides therefore an event-based processing mode which avoids the in-memory construction of the input, and in addition recognizes matches of queries at the earliest possible moment.

Typical XML queries identify only individual locations in the input. A useful extension is to retrieve k locations which are in a specified context. Binary queries (obtained for $k = 2$) are identified in this work as a useful case, especially in view of XML transformations. Besides for unary queries we therefore develop algorithms for the evaluation of binary queries. These techniques are at the base of our XML transformation tool Fxt.

The practical results are based on the compilation of XML queries to grammars, as used in XML schema languages, and on tree automata based constructions, tailored to our application domain.

A more detailed overview of the addressed topics and the contributions presented is given in the introductions to the first and second parts of this work.

Zusammenfassung

Diese Arbeit betrachtet eine ausdrucksstarke Sprache zur Suche in XML-Dokumenten, was eine grundlegende Aufgabe von XML-Anwendungen ist. Für diese Anfrage-Sprache werden effiziente Algorithmen zur Auswertung bereit gestellt und in dem Tool Fxgrep realisiert.

Sehr große XML-Dokumente können nicht als Bäume im Speicher aufgebaut werden. Dasselbe gilt für XML-Daten, die verarbeitet werden müssen, noch während sie empfangen werden. In diesen Fällen werden spezielle Algorithmen benötigt, die die XML-Eingabe statt als Baum als Strom von Ereignissen ansehen. Diese verarbeiten die Eingabe, indem sie auf Ereignisse reagieren. Deshalb verfügt Fxgrep über einen strom-basierten Bearbeitungsmodus, der es so weit wie möglich vermeidet, eine interne Repräsentation des Dokuments im Speicher aufzubauen, und zudem Treffer der Anfrage zum frühest möglichen Zeitpunkt signalisiert.

Typische XML-Anfragen suchen nur nach einzelnen Teildokumenten, die über ihre Eigenschaften sowie die Eigenschaften ihres Kontexts definiert sind. Sinnvoll sind aber auch Anfragen, die nach k-Tupeln von Teildokumenten mit gegebenen Eigenschaften suchen. Zweistellige Anfragen werden in dieser Arbeit als einen im Hinblick auf XML-Transformationen besonders nützlichen Fall identifiziert. Hier werden darum neben der Behandlung von einstelligen Anfragen insbesondere Techniken entwickelt, um zweistellige Anfragen zu beantworten. Diese Methoden bilden die Grundlage unseres effizienten Transformationstools Fxt.

Die eingeführten Algorithmen erzeugen aus Anfragen Grammatiken, wie sie auch in Schema-Sprachen eingesetzt werden, und konstruieren aus diesen maßgeschneiderten Baumautomaten.

Ein ausführlicherer Überblick über die behandelten Themen sowie die hier vorgestellten neuen Forschungsergebnisse befindet sich in den Einführungen zum ersten und zweiten Teil der Arbeit.

Publications

Some of the results of this work have been presented in the following publications.

[1] Alexandru Berlea and Helmut Seidl. Binary Queries for Document Trees. In *Nordic Journal of Computing*, vol. 11(1), p. 41-71, March 2004.

[2] Alexandru Berlea and Helmut Seidl. Transforming XML Documents Using Fxt. In *Journal of Computing and Information Technology*, vol. 10(1), p. 19-35, March 2002.

[3] Alexandru Berlea. Binary Queries and XML Transformations. In *Berliner XML Tage 2004*, October 2004.

[4] Alexandru Berlea and Helmut Seidl. Querying and Transforming XML Documents Using Tree Automata. In *European Joint Conferences on Theory and Practice of Software (ETAPS)*, April 2003.

[5] Alexandru Berlea and Helmut Seidl. Binary Queries. In *Extreme Markup Languages 2002*, August 2002.

[6] Alexandru Berlea and Helmut Seidl. Fxt - A Transformation Language for XML Documents. In *XML Conference & Exposition 2001*, December 2001.

Contents

List of Figures

List of Tables

Part I

Grammar Queries on Forests

Chapter 1

Introduction

The Extensible Markup Language (XML) [XML98a] is a specification released by the W3C Consortium basically providing a syntax for sequentially representing hierarchically structured information. Rather than a language, XML is a metalanguage as it allows the specification of (the syntax of) other languages, hence the claim of extensibility in its name. A language defined according to the XML standard is called an XML application or XML language.

As the information handled in various application domains has a hierarchical structure, XML syntax can be used to represent data in a very large number of fields. Indeed, since the publication of the standard in 1998, XML has been used for representing information in innumerable areas. XHTML (layouts for Web-pages [XHT02]), XSL:FO (printable pages [XSL01]), WML (mobile telephones [WML01]), SVG (graphics [SVG03]), DBLP (bibliographic records [DBL05]), MathML (mathematical formulas [Mat03]), VoiceXML (audio user interfaces [Voi04]), SMIL (interactive audio-visual presentations [SMI98]), WSDL (descriptions of interfaces of Web services [CCMW01]), RSS (news distribution [RSS03]), OpenOffice.org XML (office documents [Org02]), XMI (software modeling [XMI03]), CML, PSD (chemistry [CML03, PSD]) and LegalXML (collections of laws [Leg02]) are just a few of the better known XML languages[1].

The spreading of XML is favored by the increased need of exchanging information in a standardized way, such that communicating entities are coupled as loosely as possible in order to achieve better interoperability. XML has become almost indispensable as an exchange format between communicating applications.

Another major advantage of XML is that it makes possible the separation of information content from information presentation. This is highly desirable when different presentations of the same content are needed, for instance XHTML for presence on the Internet, and Printable Document Format (PDF) for documents to be printed. Maintaining a separate document for each layout is inconvenient, due to the overhead required to keep all copies consistent whenever the informational content changes. Instead, one can maintain a single document containing the information represented in an XML language and one transformation program from the XML language to each desired layout. Thereby, keeping a layout up to date simply requires the running of the corresponding transformation program on the content whenever this changes.

[1]An extensive list of XML languages can be found at `http://www.oasis-open.org/cover/xml.html`.

Exploiting the advantages of using XML thus intrinsically requires the ability to transform XML documents, either to convert them from and to the exchange formats, or to produce a desired layout. As a basic task in XML applications, transforming XML data has an important contribution to their overall performance. Therefore, care must be taken to implement transformations as efficiently as possible. A concurrent desideratum is that transformations are specified as intuitively as possible, especially if XML is to gain even more user acceptance. As part of XML transforming, *querying*, i.e. locating parts of documents with some specified properties, is a fundamental task in XML processing and should therefore follow the same desiderata.

The first part of the work is concerned with XML querying. Our contributions in particular are as follows.

A powerful method for specifying k-ary queries Most of the XML-queries identify only individual locations in the XML input data. We introduce a very expressive method which allows the specification of k-ary queries, that is, queries retrieving k locations which are in a specified context. The method is based on grammars, which are already applied in XML, but mainly only as schema languages. We call these queries k-ary grammar queries.

Efficient implementation of binary queries We identify binary queries (obtained for $k = 2$) as an important special case, particularly in view of their use in transformations. We introduce a tree-automata based algorithm which allows the efficient evaluation of binary grammar queries. The grammar formalism and its implementation are further extended to allow the specification of queries using boolean connectives.

Query evaluation on XML streams Some XML documents may be very large, which makes it prohibitively expensive to keep them completely in the main memory while processing them. Also, there are increasingly many real-life applications in which the document to be processed is received linearly via some communication channel – as an XML stream –, rather than being completely available in advance. Special algorithms have to be developed to cope with these constraints. We solve the problem of answering grammar queries on XML streams by providing an algorithm which is very efficient in terms of time and highly adaptive in terms of memory consumption.

Practical implementation in Fxgrep The viability of the algorithms and ideas presented in this part has been demonstrated by implementing them in the XML querying tool Fxgrep [NB05], which provides access to the powerful grammar querying formalism via a more intuitive, and thus more user-friendly, specification language.

This part is organized as follows. In Chapter 2 we introduce terminology, definitions, notations and a classic automata construction which are used throughout this work. Fxgrep is introduced in Chapter 3. The forest grammars used for specifying queries, the languages specified by them, and their recognition are presented in Chapter 4. Chapter 5 presents how forest grammars can be used to specify k-ary queries and how these queries, in particular the binary ones, can be evaluated. Equipping grammar queries with boolean connectives is addressed in Chapter 6. Answering queries on XML streams is presented in Chapter 7. The next part then introduces an XML transformation language based on the querying techniques introduced in this first part.

Chapter 2

Preliminaries

Hierarchically structured information, such as XML documents, can be conceptually represented as trees. XML processing is thus basically tree processing. An even more basic task is string processing. Regular expressions are an intuitive yet quite expressive way of specifying properties of strings. Furthermore, they are at the basis of our more elaborated patterns to be located in trees, as presented in the following chapters. Therefore, we start in Section 2.1 with a presentation of regular expressions and the classic Berry-Sethi automata construction checking the conformance to a specified regular expression. In Section 2.2 we then introduce trees and a couple of related definitions and notations which will be used throughout the remainder of the work. In Section 2.3 we briefly present the basic XML concepts, notations and terminology and relate them to the usual tree terminology.

2.1 Regular Expressions

Let Σ be a finite set. We call Σ *alphabet* and its elements *symbols*. The number of elements of Σ is denoted as $|\Sigma|$. The set Σ^* of *strings* w over the alphabet Σ is defined as follows:

$$w \in \Sigma^* \text{ iff } w = \lambda \text{ or } w = aw_1 \text{ with } a \in \Sigma \text{ and } w_1 \in \Sigma^*$$

where λ is the *empty string*.

The set of *regular expressions* over the alphabet Σ denoted as $\mathcal{R}_\Sigma$ is defined as:

$$\begin{array}{ll} r \in \mathcal{R}_\Sigma \text{ iff} & r = \emptyset, \text{ or} \\ & r = \lambda, \text{ or} \\ & r = a \text{ and } a \in \Sigma, \text{ or} \\ & r = r_1? \text{ and } r_1 \in \mathcal{R}_\Sigma, \text{ or} \\ & r = r_1{}^* \text{ and } r_1 \in \mathcal{R}_\Sigma, \text{ or} \\ & r = r_1 r_2 \text{ and } r_1, r_2 \in \mathcal{R}_\Sigma, \text{ or} \\ & r = r_1 \,|\, r_2 \text{ and } r_1, r_2 \in \mathcal{R}_\Sigma. \end{array}$$

Parentheses may be omitted, in which case the composition of a regular expression is given by using the following operator precedence: ?, *, concatenation and $|$, from the strongest to the weakest. The number of occurrences of symbols from Σ in a regular expression r is denoted as $|r|_\Sigma$.

The *language of a regular expression* $r \in \mathcal{R}_\Sigma$ is a set $[\![r]\!] \subseteq \Sigma^*$ defined as follows:

$$\begin{array}{ll}
[\![\emptyset]\!] & = \emptyset \\
[\![\lambda]\!] & = \{\lambda\} \\
[\![a]\!] & = \{a\}, \text{for all } a \in \Sigma \\
[\![r?]\!] & = \{\lambda\} \cup [\![r]\!] \\
[\![r^*]\!] & = \{\lambda\} \cup \{w_1 \dots w_n \mid n > 0, w_i \in [\![r]\!] \text{ for all } 1 \leq i \leq n\} \\
[\![r_1 r_2]\!] & = \{w_1 w_2 \mid w_1 \in [\![r_1]\!], w_2 \in [\![r_2]\!]\} \\
[\![r_1 \mid r_2]\!] & = [\![r_1]\!] \cup [\![r_2]\!]
\end{array}$$

A language is called a *regular string language* if it is the language of a regular expression.

Finite Automata

The membership of a string to a regular string language can be tested by a finite automaton. A *finite automaton* over an alphabet Σ is a tuple $A = (Q, q_0, F, \delta)$ consisting of a set of *states* Q, an *initial state* $q_0 \in Q$, a set of *final states* $F \subseteq Q$ and a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$. The language $\mathcal{L}_A$ accepted by the automaton A is defined as follows:

$$\begin{array}{lll}
\lambda \in \mathcal{L}_A & \text{iff} & q_0 \in F \\
a_1 \dots a_n \in \mathcal{L}_A & \text{iff} & \text{there are } q_1, \dots, q_{n+1} \in Q \text{ such that } q_1 = q_0 \text{ and} \\
 & & (q_i, a_i, q_{i+1}) \in \delta \text{ for all } 1 \leq i \leq n.
\end{array}$$

If δ is a function, rather than a relation, A is called *deterministic finite automaton* (DFA). Otherwise, A is called *non-deterministic finite automaton* (NFA).

The Berry-Sethi Construction

One method to construct an NFA accepting the language of a regular expression is the algorithm proposed by Berry and Sethi [BS86]. Given a regular expression r this constructs an NFA, $Berry(r) = (Q, q_0, F, \delta)$, accepting exactly the language $[\![r]\!]$ as follows.

If $r = \lambda$ then $Berry(r) = (\{q_0\}, q_0, \{q_0\}, \emptyset)$ where q_0 is some arbitrarily chosen state. If $r = \emptyset$ then $Berry(r) = (\{q_0\}, q_0, \emptyset, \emptyset)$ where q_0 is some arbitrarily chosen state.

Otherwise, a set P of positions p is generated s.t. $|P| = |r|_\Sigma$. Further, a bijection f from the set of occurrences of symbols in r into P is defined. A regular expression $\overline{r} \in \mathcal{R}_P$ is constructed by replacing each occurrence o with $f(o)$. Then, for each subexpression $\overline{r}_1$ of $\overline{r}$ the following information is computed in the given order.

1. $Empty(\overline{r}_1)$, denoting whether $\lambda \in [\![\overline{r}_1]\!]$, given as follows:

$$\begin{array}{ll}
Empty(p) & = \mathit{false} \\
Empty(r_1?) & = \mathit{true} \\
Empty(r_1{}^*) & = \mathit{true} \\
Empty(r_1 r_2) & = Empty(r_1) \text{ and } Empty(r_2) \\
Empty(r_1 \mid r_2) & = Empty(r_1) \text{ or } Empty(r_2)
\end{array}$$

2. $First(\overline{r}_1)$, denoting the symbols with which strings from $[\![\overline{r}_1]\!]$ may start:

$$\begin{array}{ll} First(p) & = \{p\} \\ First(r_1?) & = First(r_1) \\ First(r_1{}^*) & = First(r_1) \\ First(r_1 \,|\, r_2) & = First(r_1) \cup First(r_2) \\ First(r_1 r_2) & = First(r_1) \cup \begin{cases} First(r_2), & \text{if } Empty(r_1) \\ \emptyset \quad\quad\quad , & \text{otherwise} \end{cases} \end{array}$$

3. $Follow(\overline{r}_1)$, denoting the symbols which can immediately follow after a string w from $[\![\overline{r}_1]\!]$ within a string from $[\![\overline{r}]\!]$ or \$ (an auxiliary symbol) if w might be a suffix of a string in $[\![\overline{r}]\!]$:

$$\begin{array}{l} \text{If } \overline{r}_1 = \overline{r} \text{ then } Follow(\overline{r}_1) = \{\$\} \\ \text{If } \overline{r}_1 = r_1? \text{ then } Follow(r_1) = Follow(\overline{r}_1) \\ \text{If } \overline{r}_1 = r_1{}^* \text{ then } Follow(r_1) = Follow(\overline{r}_1) \cup First(r_1) \\ \text{If } \overline{r}_1 = r_1 \,|\, r_2 \text{ then } Follow(r_1) = Follow(r_2) = Follow(\overline{r}_1) \end{array}$$

$$\begin{array}{l} \text{If } \overline{r}_1 = r_1 r_2 \text{ then} \\ \quad Follow(r_2) = Follow(\overline{r}_1) \\ \quad Follow(r_1) = First(r_2) \cup \begin{cases} Follow(\overline{r}_1), & \text{if } Empty(r_2) \\ \emptyset \quad\quad\quad\quad , & \text{otherwise} \end{cases} \end{array}$$

Given the definitions above, $Empty()$ and $First()$ can be computed in a bottom-up while $Follow()$ can be computed in a top-down manner. Using $Empty(), First()$ and $Follow()$, the NFA is defined as follows. The set of states is $Q = \{q_0\} \cup P$ with some arbitrarily chosen start state $q_0 \notin P$. The set F of final states is obtained as:

$$F = \begin{cases} \{p \in P \mid \$ \in Follow(p)\} \cup q_0, & \text{if } Empty(\overline{r}) \\ \{p \in P \mid \$ \in Follow(p)\} \quad\quad , & \text{otherwise} \end{cases}$$

Let sym be the inverse of function f, i.e. the mapping of each position p to the symbol occurring at $f^{-1}(p)$. The transition relation is given by:

$$\begin{array}{ll} \delta = & \{(q_0, sym(p), p) \mid p \in First(\overline{r})\} \cup \\ & \{(p, sym(p_1), p_1) \mid p, p_1 \in P, p_1 \in Follow(p)\} \end{array}$$

Example 2.1: Consider the regular expression $r = a^*(a \,|\, b)b^*$. We choose as the set of positions $P = \{1, 2, 3, 4\}$ and associate i with the i-th symbol occurrence, hence $sym = \{(1, a), (2, a), (3, b), (4, b)\}$ and $\overline{r} = 1^*(2 \,|\, 3)4^*$. The syntax tree of $\overline{r}$ is depicted in Figure 2.1. The internal nodes of the tree denote the subexpressions of $\overline{r}$. The Berry-Sethi construction proceeds as follows.

1. $Empty(1) = Empty(2) = Empty(3) = Empty(4) = \mathit{false}$
   $Empty(1^*) = Empty(4^*) = \mathit{true}$
   $Empty(2 \,|\, 3) = \mathit{false}$
   $Empty(1^*(2 \,|\, 3)) = \mathit{false}$
   $Empty(1^*(2 \,|\, 3)4^*) = \mathit{false}$

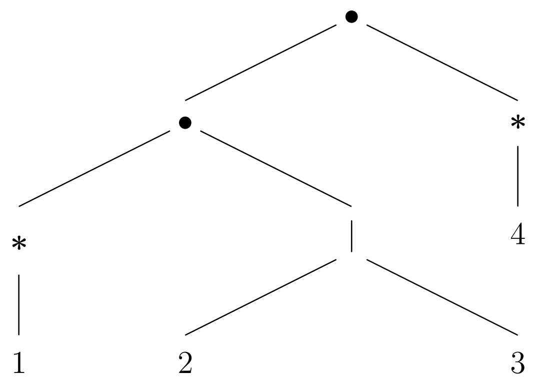


Figure 2.1: Syntax tree of the regular expression $1^*(2\,|\,3)4^*$

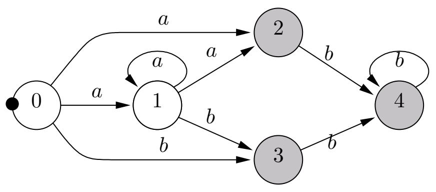


Figure 2.2: NFA obtained by the Berry-Sethi construction for $a^*(a\,|\,b)b^*$

2. $\mathit{First}(i) = \{i\}$ for all $1 \leq i \leq 4$
   $\mathit{First}(1^*) = \{1\}$, $\mathit{First}(4^*) = \{4\}$
   $\mathit{First}(2\,|\,3) = \{2,3\}$
   $\mathit{First}(1^*(2\,|\,3)) = \{1,2,3\}$
   $\mathit{First}(1^*(2\,|\,3)4^*) = \{1,2,3\}$

3. $\mathit{Follow}(1^*(2\,|\,3)4^*) = \{\$\}$
   $\mathit{Follow}(1^*(2\,|\,3)) = \{4,\$\}$, $\mathit{Follow}(4^*) = \{\$\}$
   $\mathit{Follow}(4) = \{4,\$\}$
   $\mathit{Follow}(1^*) = \{2,3\}$, $\mathit{Follow}(2\,|\,3) = \{4,\$\}$
   $\mathit{Follow}(1) = \{1,2,3\}$, $\mathit{Follow}(2) = \{4,\$\}$, $\mathit{Follow}(3) = \{4,\$\}$

By choosing $q_0 = 0$ it follows that $Q = \{0,1,2,3,4\}$, $F = \{2,3,4\}$ and $\delta = \{(0,a,1),(0,a,2),(0,b,3),(1,a,1),(1,a,2),(1,b,3),(2,b,4),(3,b,4),(4,b,4)\}$. The obtained NFA is depicted in Figure 2.2, where the initial state is marked by the • symbol and final states are depicted in gray. □

An NFA obtained by the Berry-Sethi construction has the important property that all transitions coming into the same state are labeled by the same symbol. We denote the label of the incoming transitions into an NFA state y by $\mathit{in}(y)$.

2.2 Trees and Forests

Let Σ be a set that we call *alphabet*. The sets $\mathcal{T}_\Sigma$ of trees t and $\mathcal{F}_\Sigma$ of *forests* f over Σ is defined as follows:

$$\begin{aligned} t \in \mathcal{T}_\Sigma &\quad \text{iff} \quad t = a\langle f \rangle \text{ with } a \in \Sigma \text{ and } f \in \mathcal{F}_\Sigma \\ f \in \mathcal{F}_\Sigma &\quad \text{iff} \quad f = \varepsilon \text{ or } f = tf_1 \text{ with } t \in \mathcal{T}_\Sigma \text{ and } f_1 \in \mathcal{F}_\Sigma \end{aligned}$$

where ε denotes the *empty forest*. Given $t = a\langle f \rangle$, the symbol a denotes the *label* and f the *children* of t. To denote the label of t we also write $lab(t) = a$.

A tree $a\langle \varepsilon \rangle$ is called a *leaf* and may be denoted by $a\langle\rangle$ or simply by a. Also, rather than $t\varepsilon$ or εt, we write t. The notation t can be thus interpreted both as a tree or a forest consisting of exactly one tree. Both interpretations are valid in most usage contexts. We will explicitly note the intended interpretation when the distinction is relevant and if it is not obvious from the context.

Let $t = a\langle t_1 \dots t_n \rangle$. The trees t_i are the *children* of t, while t is the *father* of all t_i trees for $1 \leq i \leq n$. Two trees t_i and t_j with $1 \leq i, j \leq n$ and $i \neq j$ are called *siblings*. If $i < j$ then t_i is a *left sibling* of t_j and t_j is a *right sibling* of t_i.

Note that, as required by the XML format, our trees are *unranked*, that is the sequence f of children of a tree $a\langle f \rangle$ may have an arbitrary length. We could have used as well a ranked representation like in the traditional tree theory, as each tree or forest can be reversibly encoded into a unique ranked tree (see for example [Neu00]). Working with the encoded representations however complicates both the operations on trees and forest and the intuitions behind them, hence we preferred the straightforward unranked representation. Also note that the trees defines above are *ordered*, i.e. the order of the children is relevant.

Nodes, Paths and Locations

Any subtree t of a forest f is uniquely identified by a *node*. A node is a string of natural numbers, denoting the *path* leading to t, formally defined as follows. The set $\Pi(f) \subseteq \mathbb{N}^*$ contains all *paths* π in f and is defined as follows:

$$\begin{aligned} \Pi(\varepsilon) &= \{\lambda\} \\ \Pi(t_1 \dots t_n) &= \{\lambda\} \cup \{i\pi \mid \ 1 \leq i \leq n, \pi \in \Pi(f_i) \text{ for } t_i = a_i\langle f_i \rangle\} \end{aligned}$$

where $\mathbb{N}^*$ is the set of strings over the alphabet of positive natural numbers and λ denotes the empty string.

The *nodes* of a forest f are elements of the set $N(f) = \Pi(f) \setminus \{\lambda\}$. For $\pi \in N(f)$, $f[\pi]$ is called the *subtree of f located at π* and is defined as follows:

$$(t_1 \dots t_n)[i\pi] = \begin{cases} t_i \quad , & \text{if } \pi = \lambda \\ f_i[\pi], & \text{if } \pi \neq \lambda \text{ and } t_i = a\langle f_i \rangle \end{cases}$$

For a node π, we define $last_f(\pi)$ as the number of children of π:

$$last_f(\pi) = max(\{n \mid \pi n \in N(f)\} \cup \{0\})$$

with $last_f(\pi) = 0$ iff π identifies a leaf.

Note that a path always locates a tree in a *forest*, not in a tree. Given a tree t, $t[\pi]$ denotes the tree located by π in the forest which consists of t only. One can see by definition that in this case π always begins with the symbol 1. In particular, one can use the subtree $t = f[\pi_1]$ located by a path π_1 in a forest f to further locate a subtree of t. In this case we have that $f[\pi_1][1\pi_2] = f[\pi_1\pi_2]$.

The *document order* is defined as the lexicographic order of the nodes of a forest f. Note that this is precisely the order in which the nodes are visited

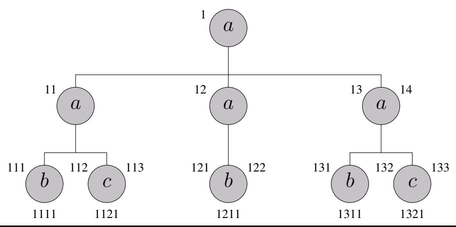


Figure 2.3: Locations in a tree

during a left-to-right depth-first search (DFS) traversal of f. Sometimes we need to precisely identify the locations reached during a DFS traversal of a forest. To this aim we define the set $L(f) \subseteq \mathbb{N}^*$ of *locations* in a forest f as:

$$\begin{array}{rcl} L(\varepsilon) & = & \{1\} \\ L(t_1 \ldots t_n) & = & \{i \mid 1 \leq i \leq n+1\} \cup \\ & & \{il \mid \ 1 \leq i \leq n, l \in L(f_i) \text{ for } t_i = a_i\langle f_i \rangle\} \end{array}$$

Figure 2.3 depicts these locations in a sample tree. The location at which the root of a subtree is reached (depicted to its left) equals the node at which the subtree is located.

2.3 XML Basics

This section is only meant to briefly introduce the essential XML constructs and to relate the XML terminology to the tree terminology. For a thorough introduction to XML we refer to the books dedicated to this subject, such as to [Har01].

Basically, an XML document is a serial representation of an ordered, unranked, labeled tree. The XML representation of a tree $a\langle f \rangle$ is an XML *element* given as *serialize*$(a\langle f \rangle)$, where:

$$\begin{array}{lcl} \mathit{serialize}(a\langle f \rangle) & ::= & \texttt{<a>}\mathit{serialize}(f)\texttt{</a>} \\ \mathit{serialize}(t_1 \ldots t_n) & ::= & \mathit{serialize}(t_1) \ldots \mathit{serialize}(t_n) \\ \mathit{serialize}(\varepsilon) & ::= & \lambda \end{array}$$

For example, $a\langle b\langle c \rangle d \rangle$ can be denoted in XML as `<a><b><c></c></b><d></d></a>` or using (irrelevant) white spaces for enhanced readability:

XML Example 1

```
<a>
  <b>
    <c></c>
  </b>
  <d></d>
</a>
```

Consider a tree $a\langle f \rangle$ and the corresponding XML element `<a>`*serialize*(f) `</a>`. The XML terminology denominates the symbol `a` *tag* or *element name*, `<a>` *start tag*, `</a>` *end tag* and *serialize*(f) *element content*. An element with empty content `<a></a>` is called *empty element* and might be as well denoted as `<a/>`. The element corresponding to the root of the top-level tree is the *root element*.

Additionally, XML elements can be provided with named properties via *attributes*. An attribute is a pair consisting of an *attribute name* and an *attribute value* given as arbitrary sequences of symbols. Attributes are specified along with the start tag of the element, after the tag name, as the attribute name followed by the "=" sign followed by the attribute value enclosed in single or double quotes, as for example in:

XML Example 2

```
<circle radius='25' x="40" y='60' color='green'/>
```

Note that attributes do not add to the expressiveness of XML as they could also be represented by using dedicated element names, as for example:

XML Example 3

```
<circle>
  <attributes>
    <name>radius</name><val>25</val>
    <name>x</name><val>40</val>
    <name>y</name><val>60</val>
    <name>color</name><val>green</val>
</circle>
```

Besides other elements, *text* and *processing instruction nodes* may occur anywhere within an element. A text node consists of a sequence of symbols which occur within the enclosing element. A processing instruction is intended to provide an *instruction* to some *target processor* of the XML representation and has the form `<?target attributes?>`. The attributes are as for elements and are to be interpreted by the target processor. Processing instructions are also allowed to occur before and after the root element. Therefore, an XML document is a forest rather than a tree as the root element might be preceded and followed by processing instruction nodes.

For example an XSL-enabled browser uses the processing instruction at the beginning of the following XML document:

XML Example 4

```
<?xml-stylesheet type="text/xsl" href="program2html.xsl"?>
<Program>
  <Output>Hello World!<newline/>Und Tschüs!</Output>
</Program>
```

to retrieve the stylesheet `program2html.xsl` and apply it to the document in order to obtain its Web presentation.

Similarly to attributes, processing instructions do not actually add to expressiveness, as their information can be provided via elements with a dedicated name.

An element like `Output` in XML Example 4 which encompasses both element and text nodes is said to have *mixed content*, while an element with only text nodes is said to have *text content*.

The symbols occurring in an XML document, in tags, attribute names and values, text nodes and processing instructions can be basically any Unicode [Con03] characters. Unicode is a character set containing virtually all symbols from any known alphabets used in any written language. The first version of the XML standard, 1.0 adhered to the 2.0 [Con96] version of the Unicode standard. The Unicode continuously evolves as new characters are added to the character set. To account for these changes a new version of the XML standard, 1.1 [XML04], was recently released. Also, to achieve a looser coupling between the two standards, XML 1.1 follows a strategy in which the names occurring in the different XML constructs are allowed to contain any Unicode characters except those which are explicitly forbidden, accounting thus for future evolutions of the Unicode character set.

Unicode characters can be electronically represented using different encodings. By default, an XML processor assumes that the encoding used is UTF-8, a variable length encoding which uses one byte for the usual Western characters and includes ASCII as a subset. If a different encoding is used, then this must be explicitly declared. The version of the XML standard to which a document adheres, as well as the encoding of the document, can be declared via the so-called XML *declaration*. The XML declaration is a special processing instruction which must occur at the very beginning of the document and might look like:

XML Example 5

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

XML Languages

The XML specification includes a method for specifying structural constraints for XML documents. The set of documents adhering to a set of given constraints is called an XML *language*. An XML language is defined using a *document type definition* (DTD). An XML document can be declared as belonging to an XML language by providing it with a *document type declaration*. The document type declaration indicates the root element and either directly provides the DTD, in which case the DTD is called *internal*, or it provides a reference to an *external* location where the DTD is to be found as for example in:

XML Example 6

```
<!DOCTYPE dblp SYSTEM "dblp.dtd">
```

saying that the DTD is given in the file named `dblp.dtd`.

An XML document which conforms to its declared DTD is called *valid*. Checking validity of XML documents is achieved by XML validating parsers. Checking the validity of XML is very important for applications which rely on a specific format for their XML input, especially if the source of the input is not controllable, as it is often the case in highly dynamic settings.

A DTD consists of declarations restricting the content of the elements occurring in XML documents conforming to the DTD. The content of an element might be restricted depending on the element's name. One can either specify that an element should have only text content, or mixed content, or give a *content-model* for it. A content model specified in a DTD is a regular expression over element names which has to be fulfilled by the string of element names of the enclosed elements. For example:

XML Example 7

```
<ELEMENT ulist (item+)>
```

specifies that an element named `ulist` must consist of one or more `item` elements. Furthermore one can specify which are the attributes that an element might have and whether they are required, optional, or that they have some fixed value.

Even though DTDs are the only means of specifying XML languages anchored in the XML specification, they are just one way of doing so. The languages used to specify XML languages are called XML *schema languages*, as they specify a *schema* to which the XML documents belonging to the XML language must conform. The structural constraints expressible with the proposed schema languages are in general more precise than those allowed by DTDs, and are basically subsumed by the capabilities of forest grammars, which will be introduced in Section 4.1.1. A comparison of forest grammars and the most frequently used schema languages is presented in Section 4.1.3.

Chapter 3

A Primer for Fxgrep

An important design purpose of tools which are to be used by people with various backgrounds, as in the case of XML query languages, is that they are as intuitive and easy to learn as possible. The small number of constructs needed to build complex regular expressions makes them a simple yet powerful way of specifying patterns of symbols. Consequently, they have been intensively used already for searching in flat (i.e. not hierarchically structured) documents.

The classic regular expressions can be used also to search for string patterns in hierarchically structured documents, yet they are not able to exploit the supplementary structure information in order to provide more precise patterns to be recognized. To account for this, the XML query language Fxgrep [NB05][1] was designed to extend the convenience of using regular expressions from strings to trees. This section is intended to be a primer on Fxgrep , as the language which constituted the motivation of the research presented in the remainder of this part of the work.

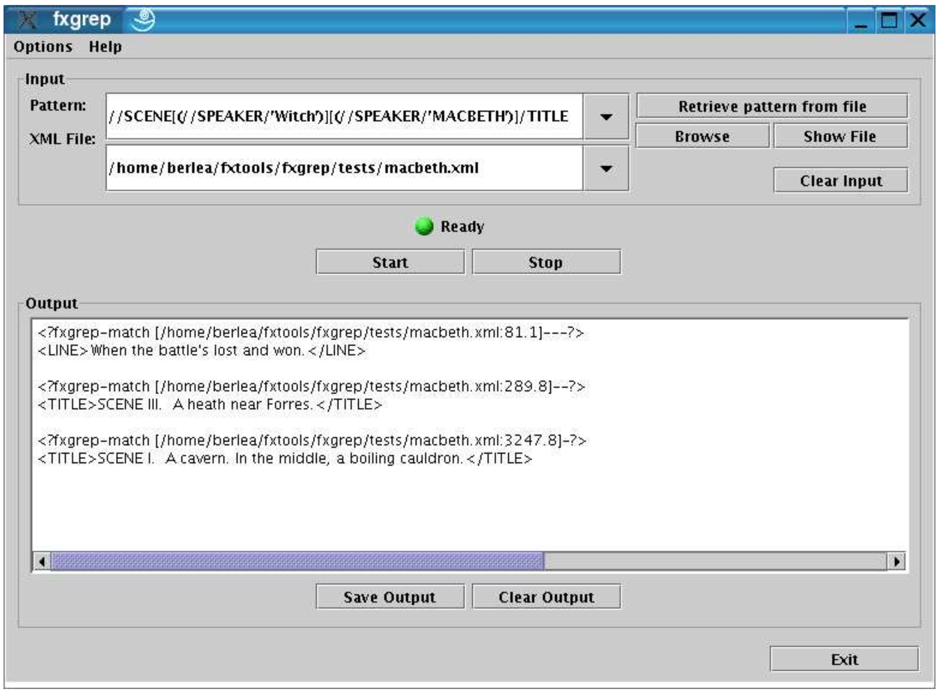
Fxgrep receives as input a query and an XML input document and responds with the locations in the XML input identified by the query. Queries can be concisely specified via *patterns*, the construction of which is introduced in the rest of this chapter. Fxgrep can be invoked from the command line or via a graphical user interface as depicted in Figure 3.1.

3.1 Paths

A quite familiar representative of hierarchically structured information is a file system, in which directories and files are organized in a tree structure. In this context, a query is simply a file name denoting the path to be followed to a file or directory of interest. It is thus sensible to use paths as a syntactical basis for a query language for XML documents, as XPath [XPa99, XPa05] does, the most prominent XML query language yet. Fxgrep builds upon the same analogy and is correspondingly *syntactically* similar to XPath.

✧ Completely analogously to a file name, the pattern `/a/b/c` returns XML elements labeled `c` which are children of `b` elements contained in the root element `a`.

[1]Fxgrep is an acronym for "the functional XML grep", where "functional" denotes that Fxgrep is written in a functional programming language (SML) and "grep" is an allusion to the Unix string search tool grep [Fou05].



Pattern: `//SCENE[(//SPEAKER/'Witch')][(//SPEAKER/'MACBETH')]/TITLE`

XML File: `/home/berlea/fxtools/fxgrep/tests/macbeth.xml`

```
<?fxgrep-match [/home/berlea/fxtools/fxgrep/tests/macbeth.xml:81.1]---?>
<LINE>When the battle's lost and won.</LINE>

<?fxgrep-match [/home/berlea/fxtools/fxgrep/tests/macbeth.xml:289.8]--?>
<TITLE>SCENE III.  A heath near Forres.</TITLE>

<?fxgrep-match [/home/berlea/fxtools/fxgrep/tests/macbeth.xml:3247.8]-?>
<TITLE>SCENE I.  A cavern. In the middle, a boiling cauldron.</TITLE>
```

Figure 3.1: Fxgrep Graphical User Interface

As opposed to file names, patterns may identify more then one node in the input tree. Also, paths do not need to be completely specified. The *deep-match* construct "//" may be used to denote an arbitrary number of steps in a path.

✧ The pattern `/a/b//c` returns XML elements labeled `c` which are descendants of `b` elements contained in the root element `a`.

Besides by their name, the element names in a path may be specified as regular expressions. The regular expression is to be fulfilled by the referred element names and must be single or double quoted and enclosed between the "<" and ">" symbols, as in the case of an XML element tag.

✧ The pattern `//<'(sub)*section'>` locates all `section`s, `subsection`s and `subsubsection`s elements in the input.

The last step in a path may not only be an element name, but could also be a regular expression specifying a model for a text node.

✧ The pattern `//section/title/"automat(a|on)"` locates the `title`s of `section`s containing the words `"automata"` or `"automaton"`.

3.1.1 Regular Path Expressions

Regular path expressions are known in the literature as regular expressions to be satisfied by the string of node labels on the path from the root to the node of interest. Fxgrep provides regular path expressions.

✧ The pattern `(king/)+ person` stands for `king/king/.../king/person` and identifies `person` nodes which have only `king` ancestors.

As we will see in Section 3.2, nodes occurring in Fxgrep patterns can be specified much more precisely as only by their label, by further qualifying them with structural and contextual constraints. By using qualifiers in regular path expressions one significantly exceeds the expressiveness of the ordinary regular path expressions.

3.1.2 Boolean Connectives for Paths

Conjunctions

To specify that the path to a match should simultaneously fulfill several path models, these can be connected via the "`&`" symbol.

- ✧ The pattern `((//king//)&(//count/duke//)) person` locates `person` elements that have *both* an ancestor `king` and an ancestor `duke` whose father node is an element `count`, as specified by the conjunction of the two connected (incomplete) path patterns `//king//` and `//count/duke//`, respectively.

Disjunctions

One can choose between several alternative paths to the match node by connecting them via the "`||`" operator.

- ✧ The pattern `//((king//)||(count/duke//)) person` locates `person` elements that have either an an ancestor `king` or an ancestor `duke` whose father node is an element `count`.

Negations

To specify that the path to a match must not satisfy a path pattern, this must be preceded by the "`!`" symbol.

- ✧ The pattern `!(//king/king//)person` locates `person`s who do not have two consecutive `king` ancestors.

3.2 Qualifiers

Nodes in paths can be qualified with both structural and contextual constraints. The structural constraints talk about the content of a node, while the contextual constraints are concerned with the surroundings of the node.

3.2.1 Structure Qualifiers

Similarly to XPath, nodes occurring in a pattern can be specified more precisely than by their name by indicating a supplementary condition to be fulfilled by a node provided within brackets following the node. Unlike in XPath, an Fxgrep qualifier is a regular expression to be fulfilled by the children of the subject node, as follows. Rather than a string regular expression, a qualifier is a regular expression over patterns. The children of a node fulfill a pattern regular expression if there is a contiguous sequence of them and an equally long sequence of patterns fulfilling the pattern regular expression, and every pattern in the sequence leads to at least one match when evaluated on the corresponding child. The following examples should clarify the idea.

- The pattern `//section[(subsection)+ conclusion]/title` locates `title`s of `section` elements. The qualifier of the `section` element in the path requires that the sought sections have one or more `subsection`s (each of them fulfilling the simple pattern `subsection`) followed by a `conclusion` element (fulfilling the simple pattern `conclusion`).

- The pattern `//section[(subsection/title/"part")+ conclusion]/title` locates `title`s of `section`s having one or more `subsection`s, the `title` of which contains the substring `"part"` (each of them fulfilling the pattern `subsection/title/"part"`), followed by a `conclusion` element.

Note that any node in a pattern can be qualified, in particular also the nodes occurring in qualifiers.

- The pattern `//section[(subsection[(theorem proof)+]/title/"part")+ conclusion]/title` locates the titles of sections as in the previous example, but supplementary requires that the subsections contain a non-empty sequence of `theorem`s followed by `proof`s.

Start and end markers

The symbols "`^`" and "`$`" can be used to denote the start and the end of a sequence to be matched by a regular expression, both for string and pattern regular expressions.

- The pattern `//section[^intro]` locates sections whose first child is an `intro` element. Compare with `//section[intro]` which locates sections which have some `intro` child element.

- The pattern `//section[^(theorem proof)+$]` locates sections consisting exclusively of `theorem`s followed by `proof`s. Compare with the pattern `//section[(theorem proof)+]` which locates sections containing a sequence of `theorem`s followed by `proof`s.

Attribute qualifiers

A special form of qualifiers are *attribute qualifiers* by which one can require that an element has an attribute with a specified name and possibly a specified value. An attribute qualifier consists of the symbol "`@`" followed by the specification of an attribute name and possibly succeeded by the symbol "`=`" and the specification of the attribute value. The name as well as the value of the attribute can be specified using regular expressions.

- The pattern `subsection[@title="Results"]` identifies subsections having an attribute `title` with value `"Results"`.

Wildcards

Often, one needs only to talk about the existence of a node or a sequence of nodes, without further specifying the appearance of the nodes. To denote an arbitrary node or an arbitrary sequence of nodes one can use the symbols "`.`" and "`_`", respectively.

- ✧ The pattern `//././section` locates sections having at least two ancestors.

- ✧ The pattern `//.[theorem _ proof]` locates elements which contain a `theorem`, followed by an arbitrary sequence of nodes, followed by a `proof`.

Boolean connectives for structure qualifiers

Conjunctions A node can have more than one qualifier, each of them being supplied in square brackets following the node. If a node has more than one qualifier than it must fulfill all the conditions defined by them.

- ✧ The pattern

  ```
  //section[(title/"soups")][(subsection/title/"tomatoes")]
  ```

  identifies sections having the word `"soups"` in the title and a subsection whose title contains the word `"tomatoes"`.

Negations Sometimes it is easier to specify what is disallowed, rather than what model is allowed for a node. A qualifier preceded by the symbol "`!`" specifies a condition which must not be fulfilled by the subject node.

- ✧ The pattern `//section[!conclusion subsection]` identifies sections that do not have a `conclusion` before a `subsection` element.

3.2.2 Context Qualifiers

Besides constraints on children, it is possible to specify constraints on siblings of a node, provided the node's father is explicitly denoted in a path. To do so, one specifies two pattern regular expressions *l* and *r* which are to be satisfied by the node's left and right siblings, respectively. Such a constraint is called a *context qualifier* and is given between brackets following the node's father in the form `[l#r]`. The symbol "`#`" denotes the subject node, which is the subtree where the path continues. The idea should become clear in the following examples.

- ✧ The pattern `//section[definition # theorem]/example` locates the `example`s directly under a `section` element if they are enclosed between a definition and a theorem. The symbol "`#`" denotes the child of the `section` element where the path continues, in this case the `example` element.

- ✧ The pattern `//section[definition # theorem]//example` locates the `example` elements located arbitrarily deep in a section under an element enclosed by a definition and a theorem. The symbol "`#`" denotes the child of the `section` element where the path continues, in this case the child element of the section which has a descendant `example`.

- The pattern `//section[#$]/subsection` locates the `subsection`s which are the last element in their section.

3.3 Binary Patterns

Rather than locating individual nodes in the input, we are sometimes interested in identifying tuples of nodes which are in some specified relationship. A particularly useful case is locating pair of nodes from some specified binary relation.

For example, rather than locating `person` elements which have some ancestor `king` we might be interested in having both the `person` and the corresponding `king` ancestors reported. To specify a binary relation in Fxgrep one must write a (unary) pattern as presented before, in which both elements of the relation are explicitly denoted. The first element of the relation has to be the target node in the pattern (the last node in the top level path). Specifying the second element of the relation is as easy as placing the special symbol, "`%`", in front of the node denoting the second element of the relation. This should be better understood by looking at the following examples.

- The pattern to locate `person` elements which have some ancestor `king` is `//king//person`. To have reported both the `person` and her `king` ancestors we place "`%`" in front of the `king` node in the pattern The binary pattern for the above query is thus `//%king//person`.

- The following unary pattern identifies all book titles whose author's names end in "escu": `//book[(author/"escu$")]/title`. Suppose we want to identify the titles as above, but together with the authors of the books with these titles. The binary query which simultaneously reports the authors having names ending in "escu" and the titles of their books is `//book[(%author/"escu$")]/title`.

Strictly speaking, a match of a binary pattern is a pair. The first node in the pair is called *primary match*. The second node in the pair is a node related to the first node as specified by the "`%`" symbol and is called *secondary match*. In practice, however, rather than reporting each primary and secondary match separately, if there are more secondary matches related to a same primary match, they are reported at once together with the primary match.

Consider for example the XML input file `library.xml`:

XML Example 8

```
<library>
  <book>
    <author>Mihai Eminescu</author>
    <author>Ion Ionescu</author>
    <title>Făt Frumos din tei</title>
  </book>
  <book>
    <author>Oscar Wilde</author>
    <title>A woman of no importance</title>
    <price>10</price>
  </book>
</library>
```

Evaluating the pattern `//book[(%author/"escu$")]/title` on `library.xml` produces:

XML Example 9

```
<match>
  <primary>
    <position>[library.xml:5.5]</position>
    <node><title>Făt Frumos din tei</title></node>
  </primary>
  <secondary>
    <position>[library.xml:3.5]</position>
    <node><author>Mihai Eminescu</author></node>
  </secondary>
  <secondary>
    <position>[library.xml:4.5]</position>
    <node><author>Ion Ionescu</author></node>
  </secondary>
</match>
```

It is possible to locate a primary match together with more than one set of related nodes. Each set of related nodes is specified by a “`%`” symbol preceding the corresponding node in the pattern. The sets of secondary matches are in this case reported together with a number denoting to which set of related nodes a match node belongs, as the ordinal number of the corresponding occurrence of the “`%`” symbol in the pattern, given as the value of an `ord` attribute.

For example evaluating the pattern:

```
//book[(%price)?][(%author)]/title["importance"]
```

on `library.xml` delivers the title of books containing the word `"importance"` together with the optional `price`, followed by the `author` of the book:

XML Example 10

```
<match>
  <primary>
    <position>[library.xml:9.5]</position>
    <node><title>A woman of no importance</title></node>
  </primary>
  <secondary ord="1">
    <position>[library.xml:10.5]</position>
    <node><price>10</price></node>
  </secondary>
  <secondary ord="2">
    <position>[library.xml:8.5]</position>
    <node><author>Oscar Wilde</author></node>
  </secondary>
</match>
```

3.4 Comparison with XPath

XPath has established itself since its standardization by the W3C Consortium as the most prominent XML query language. It is used both standalone or as part of other important languages like the XML Schema Language [XML01], XSLT [XSL99, XSL03] or XQuery [XQu05a]. As mentioned, Fxgrep shares with XPath the idea of using paths as a framework for expressing queries. In spite of these syntactic similarities, the two query languages are quite different as we present next.

3.4.1 XPath's Paths

Like Fxgrep, XPath is conceptually based on an analogy of locating subdocuments in a document with locating files in a directory tree. Nodes can be thus addressed by specifying the path from the root to them. The pattern `/book/sec/title`, as in the case of Fxgrep, locates the `title` elements, the father of which is a `sec` element whose father is the root element labeled `book`.

Fxgrep and XPath have quite different semantic models. XPath is defined in terms of an operational model. An XPath pattern specifies a number of successive selection steps. Each such step selects in turn nodes which find themselves in a specified tree relationship (called *axis* in XPath terminology) with a node selected by the previous step (the *context node*). Initially, the set of selected nodes contains only the root of the input.

The axes can be divided into *forward* and *reverse axes*, depending on whether they select nodes which are after or before the context node in document order. The forward axes are `self`, `child`, `descendant`, `descendant-or-self`, `following` and `following-sibling`. The reverse axes are `parent`, `ancestor`, `ancestor-or-self`, `preceding` and `preceding-sibling`.

The slash symbols in a pattern are thus to be seen as delimiters between the selection steps. Besides specifying a tree relationship, each selection step further specifies a so-called *node test*, i.e., the type of node to be selected (e.g., text node, processing instruction node, element with a specified name). This can be seen as a predicate required to filter the set of selected nodes. Thus, in general, a selection step has the form `treeRelationshipName::nodeTest`. For example `/book/sec/title` is an abbreviation for `/child::book/child::sec/child::title`.

Example 3.1: Consider the XPath pattern:

```
/child::book/descendant::sec/parent::node()/child::text()
```

According to the processing model of XPath this pattern is evaluated on a given input as follows. In the first step, `child::book` selects all the children of the root (i.e. the top-level processing instructions and the root element) and retains from them the element nodes named `book` (i.e. the root element if it has type `book`). Then, `descendant::sec` selects all the descendants of the `book` root element and retains the element descendants named `sec`. Next, the fathers of these `sec` elements are selected and retained whatever node type they have. Finally, the children of these father nodes are selected and those being text nodes are identified by the pattern. In the alternative, abbreviated syntax provided by XPath, the name of the `child` axis can be omitted, "`//`" stands for `descendant` and "`..`" for `parent`. Hereby, the pattern presented can be also expressed as: `/book//sec/../text()`. □

The presence of both forward and reverse axes allows the location steps to arbitrarily navigate in the input. Arbitrary navigation in the input in particular might prevent efficient stream-based implementations as it requires the input tree to be built up in memory. In [OMFB02] a set of equivalences are defined which can be used to transform absolute XPath patterns (i.e. whose initial context node for the evaluation is the root node) into equivalent patterns without reverse axes. In general however, XPath patterns are interpreted relative to other nodes selected from the input. In particular, this is the case for XPath select patterns that are used in XSLT and XQuery.

3.4.2 XPath's Qualifiers

Besides by node tests, the set of nodes selected in an XPath step can be further filtered by specifying a set of *predicates*. The predicates are given between square brackets following the node test. A predicate can be an arbitrary XPath expression. The predicate is evaluated for each of the nodes in the set and only those nodes for which it returns true are retained. In particular, predicates can be as follows.

Arithmetic expressions

An XPath qualifier can be an arbitrary arithmetic expression. The qualifier is fulfilled if the node to be filtered is on the position specified by the expression in the input document, when considered in document order on the set of nodes selected by the current step. One simple use case is counting or indexing of matches.

✧ The XPath pattern `//book[42]` locates the 42nd `book` node in the input, in document order.

Patterns

An XPath pattern occurring in a predicate denotes for each node in the set of nodes to be filtered, the set of nodes obtained by evaluating the pattern in that node's context. Such a predicate expresses a form of existential quantification as presented below.

- **Static data value comparisons:** If an XPath pattern occurs in a predicate in a comparison with some simple value, then the predicate is true if the denoted set of nodes contains at least one node with that value. We call this kind of comparisons *static data value comparisons* as one term of the comparison is known statically, before the input is read.

  ✧ The XPath pattern `book[author="Kafka"]` identifies `book` nodes having an `author` child, the text value of which is `"Kafka"`.

- **Dynamic data value comparisons:** If the predicate consists of a comparison of two XPath patterns, then the predicate is evaluated to true if there exists a node in the set denoted by the first pattern and a node in the set denoted by the second pattern such that the result of performing the comparison on the string-values of the two nodes is true. We call this kind of comparisons *dynamic data value comparisons* as both terms in the comparison are only known when the XML input document is read.

  ✧ The XPath pattern `//book[author=title]` locates the `book`s whose proud `author`s have chosen as `title` their own names.

- **Stand alone patterns:** If the predicate consists only of an XPath pattern, the pattern is evaluated in the context of each node in the set of nodes to be filtered. A node from the set of nodes to be filtered is retained if the set of nodes obtained by evaluating the XPath pattern given as predicate in its context is not empty.

  ✧ The XPath pattern `//sec[subsec]` identifies `sec` elements that have one or more `subsec` children. The pattern `//sec[subsec/theorem]` selects the `sec` elements having a `subsec` child which has a `theorem` child.

3.4.3 Differences in Expressiveness

XPath is not directly comparable with Fxgrep, that is, there are queries expressible by Fxgrep but not expressible by XPath, and also queries expressible by XPath but not expressible by Fxgrep.

Fxgrep cannot express the non-regular features of XPath, mainly: (1) indexing and counting of matches as possible in XPath via arithmetic expressions as qualifiers, and (2) dynamic data value comparisons.

On the other side, XPath is not able to express most of the regular features of Fxgrep. In XPath, structure qualifiers for a node may only contain one pattern, being thus only able to refer to one child of the node, as opposed to Fxgrep where a structure qualifier may impose a regular condition on a sequence of children.

Regular structural and contextual conditions are in general not expressible in XPath even though some simple regular conditions can be expressed by using, for example, counting of matches.

✧ The Fxgrep pattern `//sec[^subsec*$]`, locating `sec` elements whose children are all `subsec` elements, could be expressed in XPath as `//sec[count(subsec)=count(*)]`, where the qualifier requires that the number of `subsec` children equals the number of all children.

Expressing simple contextual conditions on nodes is possible by using the explicit navigation allowed in XPath and the fact that a step in a path might navigate in arbitrary directions in the input document (e.g. by navigating to the node's left or right siblings and imposing some constraints on them). The explicit navigation however makes the specification more difficult and error-prone.

✧ Locating `theorem` elements which are preceded by a `lemma` and followed by a `corollary` element, achieved in Fxgrep by using the pattern `//.[lemma#corollary]/theorem`, can be performed in XPath by the pattern
```
//theorem[name(preceding-sibling::*[1])="lemma"]
         [name(following-sibling::*[1])="corollary"]
```

Regular contextual conditions are also in general not expressible even though some simple regular conditions can be expressed in a rather cumbersome manner by using counting of matches.

✧ The Fxgrep pattern `//b[^c*#d*$]/a`, locating `a` elements, the father of which is a `b` element, and the left and right siblings of which are all `c` and `d` elements, respectively, can be expressed in XPath as:
```
//b/a[count(preceding-sibling::*)=count(preceding-sibling::c)]
     [count(following-sibling::*)=count(following-sibling::d)]
```

The examples evidence one fundamental conceptual difference between the two pattern languages. Fxgrep patterns specify properties of the nodes to be identified in a declarative way. In contrast, XPath patterns adhere to a rather *operational* style of specification, which basically consists of a succession of navigation steps.

Furthermore, no regular path expressions can be expressed in XPath. The only kind of deep-matching allowed by XPath is "`//`" (arbitrary descendant). Also, XPath can only locate individual nodes in the input as opposed to the binary patterns of Fxgrep.

3.5 Bibliographic Notes

The research interest in developing query languages for hierarchically structured data has been very vivid since the introduction of XML. Technically speaking, many approaches to querying have been proposed, using different formalisms, for example logic-, automata- or grammar-based, as it will presented in Section 5.6. Nevertheless, most of these formalisms are too complex to be directly usable by non-specialist users. Instead, given the spreading of XML in different application domains, an XML query language has to be concise and easy to learn by providing a small number of constructs, while being able to fulfill the various domain-specific requirements. We refer to such a language as a *pattern-language* as opposed to more sophisticated, yet for the non-specialist less understandable languages. Typically, the XML query tools would provide a pattern language and automatically translate the patterns into the internally used querying setting, as Fxgrep does.

Most of the proposals made are targeted at the XPath pattern language, which became the de facto industry standard XML query language. Many of them are generally able to implement different subsets of XPath. Most of them are subsumed by an XPath fragment called *Core XPath* [GKP03], mainly featuring location paths and predicates using location paths but not arithmetics and data value comparisons. Some of the research works extend XPath with simple regular path expressions [FS98, AF00, DFFT02, DF03, GMOS03].

Even though formalisms for expressing queries similarly powerful with the grammar formalism underlying Fxgrep exist (see Section 5.6), to the best of our knowledge, no other pattern language has been provided which allows one to express in a concise and declarative way the regular and contextual constraints as in the pattern language of Fxgrep.

The pattern language of Fxgrep was originally designed by Neumann and Seidl [Neu00] and contained simple paths with deep matching, structure qualifiers with boolean conectives and context qualifiers as presented above. We extended the Fxgrep pattern language in order to test the practical suitability of the concepts presented in this work by regular path expressions, regular expressions for element and attribute names, boolean connectives for paths as well as the possibility of expressing binary queries.

The task of searching tree-structured documents was present long before the arrival of XML in application-domains such as linguistics. Linguistic queries are typically performed on collections of natural language texts manually annotated with syntactical information, called *corpora*. The corpora are conceptually labeled trees, similar to XML documents, and can also be encoded in some XML format. The linguistic queries typically identify syntactical constructs by specifying tree relationships, similarly to XPath, or for this matter Fxgrep.

One of the most popular query-tools for linguistic corpora are *tgrep* [Pit96], published as early as 1992, long before the appearance of XPath, and its successor *tgrep2* [Roh04]. tgrep patterns are build together from dominance and precedence relationships among nodes, being thus similar to XPath, but using a different syntax. Finding matches of patterns requires a pre-processing phase in which the corpus to be searched is annotated with index information. Like other proposed linguistic query languages, tgrep is tied to the specific data format of the searched document and cannot be used for general tree-structured documents.

A natural choice is to represent linguistic data in XML and use general-purpose XML query languages like XPath or Fxgrep to search for linguistic patterns. XPath lacks however some features needed for linguistic queries, where not only vertical but also precise horizontal relationships among nodes need to be specified. For example, linguistic queries often need to refer to *immediately following* nodes [BCD$^+$05], i.e. nodes on the path from the following sibling to its leftmost descendant. One solution to this problem is extending XPath with more axes for horizontal navigation derived from a basic *immediately following* axis, as in *LPath* [BCD$^+$05]. The immediately-following relationship can be easily expressed in the underlying grammar formalism of Fxgrep. Given its ability to specify accurate horizontal and vertical constraints , Fxgrep is suitable for application domains like linguistics where such precise contextual specifications are needed.

Chapter 4

Regular Forest Languages

Specifying and checking conformance of XML documents to a schema, i.e. their membership to an XML language, is a very important task for XML processing. The XML specification [XML98a] introduced document type definitions as a basic schema language. Since then, more powerful schema languages have been defined, which allow a more precise specification of XML languages. Among the better known are XML Schema Language [XML01], DSD [KMS00] and RelaxNG [OAS01].

The main purpose of schema languages is to specify the structure of the documents conforming to the defined XML language. The structural properties of XML languages specifiable using the various proposed schema languages are essentially captured by *regular forest languages*. That is, XML languages are essentially regular forest languages. Correspondingly, checking conformance to a schema basically means testing membership in a regular forest language.

Since the structural conditions expressible with regular forest languages are at the basis of Fxgrep, in this chapter we briefly review how these can be specified, in Section 4.1, and recognized, in Section 4.2.

4.1 Specifying Regular Forest Languages

Regular forest languages constitute a very expressive and theoretically robust formalism for specifying properties of forests. One way of specifying regular forest languages is by using forest grammars. In fact, the proposed XML schema languages essentially specify more or less restricted forms of forest grammars. The relation between forest grammars and XML schema languages is discussed in Section 4.1.3. The reason why forest grammars are chosen among the other possibilities for specifying regular forest languages is that, in our opinion, they are more comprehensible than the other formalisms.

4.1.1 Forest Grammars

A *forest grammar* is a tuple $G = (\Sigma, X, R, r_0)$, where Σ and X are alphabets of terminal and non-terminal symbols, respectively, $R \subseteq X \times \Sigma \times \mathcal{R}_X$ is a set of productions and $r_0 \in \mathcal{R}_X$ is the *start expression*[1]. As Σ and X are visible from

[1]Recall that $\mathcal{R}_X$ is the set of regular expressions over X.

the set of productions R we omit them when there is no risk of confusion and write $G = (R, r_0)$. We denote a production $(x, a, r) \in R$ as $x \to a\langle r\rangle$. We write $x \to a$ rather than $x \to a\langle\lambda\rangle$.

Intuitively, and using the terminology from schema languages, a production $x \to a\langle r\rangle$ specifies that the children of an a element derived using the production must conform to the *content model* r. Also, the start expression is a content model which must be fulfilled by the sequence consisting of the root element and the possible preceding and following processing instructions.

Example 4.1: Consider for example the following excerpt from a file `sample.dtd` containing a DTD for books:

XML Example 11

```
<!ELEMENT BOOK (TITLE, SUBTITLE?, CHAPTER+, APPENDIX?)>
<!ELEMENT CHAPTER (TITLE, (CHAPTER|PAR)+)>
<!ELEMENT APPENDIX (CHAPTER+)>
```

Further suppose that the root element is `BOOK` as declared in the following document type declaration:

XML Example 12

```
<!DOCTYPE BOOK SYSTEM "sample.dtd">
```

The same can be specified using a forest grammar with the following productions:

$$\begin{array}{lcl} x_{book} & \to & \texttt{BOOK}\langle x_{title}\ x^?_{subtitle}\ x^+_{chapter}\ x^?_{appendix}\rangle \\ x_{chapter} & \to & \texttt{CHAPTER}\langle x_{title}\ (x_{chapter}|x_{par})^+\rangle \\ x_{appendix} & \to & \texttt{APPENDIX}\langle x^+_{chapter}\rangle \end{array}$$

We obtain the equivalent forest grammar by choosing as start expression x_{book}, corresponding to the root element in the DTD, and further assuming the presence in the grammar of productions for the non-terminals x_{title}, $x_{subtitle}$ and x_{par}, according to the DTD definitions of the elements `TITLE`, `SUBTITLE` and `PAR`, respectively. □

In the following we give the formal definition of conformance to a schema specified by a forest grammar.

A set of productions R together with a distinguished non-terminal $x \in X$ or a regular expression $r \in \mathcal{R}_X$ defines a *tree derivation* relation $\mathcal{D}eriv_{R,x} \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_X$ or a *forest derivation* relation $\mathcal{D}eriv_{R,r} \subseteq \mathcal{F}_\Sigma \times \mathcal{F}_X$, respectively, as follows:

$$\begin{array}{lll} (a\langle f\rangle, x\langle f'\rangle) \in \mathcal{D}eriv_{R,x} & \text{iff} & x \to a\langle r\rangle \in R \text{ and } (f, f') \in \mathcal{D}eriv_{R,r} \\ (t_1 \ldots t_n, t'_1 \ldots t'_n) \in \mathcal{D}eriv_{R,r} & \text{iff} & x_1 \ldots x_n \in [\![r]\!] \text{ and } (t_i, t'_i) \in \mathcal{D}eriv_{R,x_i} \\ & & \text{for } i = 1, \ldots, n \\ (\varepsilon, \varepsilon) \in \mathcal{D}eriv_{R,r} & \text{iff} & \lambda \in [\![r]\!] \end{array}$$

If $(f, f') \in \mathcal{D}eriv_{R,r}$, we say that f' is a *derivation* of f w.r.t. R and r. In the following we omit R when it is clear from the context which set of productions is meant. Given a grammar $G = (R, r)$ we write $(f, f') \in \mathcal{D}eriv_G$ iff $(f, f') \in \mathcal{D}eriv_{R,r}$ and say that f' is a derivation of f w.r.t. the grammar G.

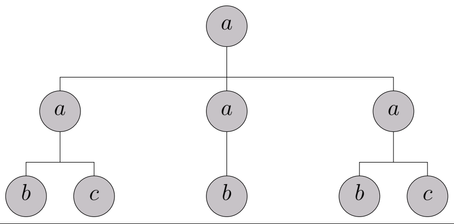


Figure 4.1: The tree representation of t from Example 4.3

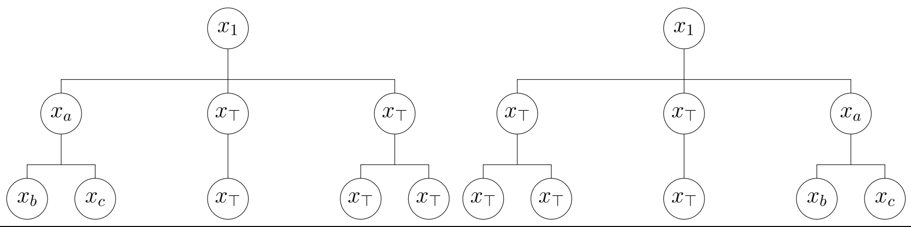


Figure 4.2: Possible derivations of t from Example 4.3

If $(f, f') \in \mathcal{D}eriv_G$ for some f', then f conforms to the schema G. Observe that a derivation f' is a relabeling of f and can be seen as a proof of the validity of f according to the schema G. If $lab(f'[\pi]) = x$ we say that f' *labels* $f[\pi]$ with x.

Note also that forest grammars have been also called unranked tree or hedge automata elsewhere [BKMW01]. From this viewpoint, non-terminals are states, productions are transitions, and derivations are accepting runs of the automaton.

Example 4.2: Let R be the set of following productions:

$$\begin{aligned} x_a &\rightarrow a\langle (x_a|x_b)^* \rangle \\ x_b &\rightarrow b \end{aligned}$$

Let $f = a\langle ab \rangle$ and suppose we want to check whether there is a derivation of f w.r.t. R and x_a. We can proceed in a bottom-up manner. It is easy to see that $(a, x_a) \in \mathcal{D}eriv_{x_a}$ and $(b, x_b) \in \mathcal{D}eriv_{x_b}$. Since $x_a x_b \in [\![(x_a|x_b)^*]\!]$ we have that $(ab, x_a x_b) \in \mathcal{D}eriv_{(x_a|x_b)^*}$. It follows that $(a\langle ab \rangle, x_a\langle x_a x_b \rangle) \in \mathcal{D}eriv_{x_a}$. $\square$

Example 4.3: Let R_2 be the set of following productions:

$$\begin{array}{lll} (1)\ x_\top \rightarrow a\langle x_\top^* \rangle & (4)\ x_1 \rightarrow a\langle x_\top^* (x_1|x_a) x_\top^* \rangle & (6)\ x_b \rightarrow b\langle x_\top^* \rangle \\ (2)\ x_\top \rightarrow b\langle x_\top^* \rangle & (5)\ x_a \rightarrow a\langle x_b x_c \rangle & (7)\ x_c \rightarrow c\langle x_\top^* \rangle \\ (3)\ x_\top \rightarrow c\langle x_\top^* \rangle & & \end{array}$$

Let t be the tree textually represented by the following XML document:

XML Example 13

```
<a>
  <a><b/><c/></a>
  <a><b/></a>
  <a><b/><c/></a>
</a>
```

The tree t is graphically presented in Figure 4.1. Two possible derivations of t w.r.t. R and the regular expression $x_1|x_a$ are depicted in Figure 4.2. □

The *meaning* $[\![R]\!]$ of a set of productions R assigns sets of trees to non-terminals $x \in X$ and sets of forests to regular expressions $r \in \mathcal{R}_X$ as follows:

$$\begin{array}{l} t \in [\![R]\!]\, x \text{ iff there is } t' \in \mathcal{T}_X \text{ with } (t, t') \in \mathcal{D}eriv_{R,x} \\ f \in [\![R]\!]\, r \text{ iff there is } f' \in \mathcal{F}_X \text{ with } (f, f') \in \mathcal{D}eriv_{R,r} \end{array}$$

If $t \in [\![R]\!]\, x$ or $f \in [\![R]\!]\, r$ we say that t can be derived from x or f can be derived from r, respectively.

Example 4.4: Let R be the set of productions from Example 4.2. It is easy to see that $[\![R]\!]\, x_b$ is the set consisting only of the tree b. The set $[\![R]\!]\, x_a$ consists of all trees, the internal nodes of which are all labeled a, and the leaves of which are labeled either a or b. □

The *regular forest language* specified by a forest grammar $G = (R, r_0)$ is the set of forests $\mathcal{L}_G = [\![R]\!]\, r_0$.

Example 4.5: Consider the grammar $G = (R_2, x_1|x_a)$ over $\{a, b, c\}$ with the productions R_2 as defined in Example 4.3.

$\mathcal{L}_G$ is the set of documents in which there is a path from the root to a node labeled a, whose children are a node labeled b and a node labeled c , and whose ancestors are all labeled a. The first three productions make $x_\top$ account for trees with arbitrary content. As specified by production (5), x_a stands for the a element with the b and the c children. Productions (6) and (7) say that these children can have arbitrary content. Finally, production (4) specifies that the a specified by (5) can be at arbitrary depth in the input, and all its ancestors must be labeled a. □

4.1.2 Practical Extensions

To use forest grammars as a specification language in a practical setting, such as XML processing, a couple of useful extensions need to be made as presented below.

4.1.2.1 Text Nodes

The grammar formalism as introduced is not yet able to handle XML documents in which elements have text content. Let $\mathcal{U}$ be the set of Unicode characters. Our definition of trees t and forests f in Section 2.2 can be adapted in order to allow XML text nodes as well as follows:

$$\begin{array}{lll} t \in \mathcal{T}_\Sigma & \text{iff} & t = a\langle f\rangle \text{ with } a \in \Sigma \text{ and } f \in \mathcal{F}_\Sigma \text{ or } t = \alpha^* \\ f \in \mathcal{F}_\Sigma & \text{iff} & f = \varepsilon \text{ or } f = tf_1 \text{ with } t \in \mathcal{T}_\Sigma \text{ and } f_1 \in \mathcal{F}_\Sigma \end{array}$$

where α^* denotes an arbitrary sequence of characters $\alpha \in \mathcal{U}$.

4.1.2.2 External Predicates

To handle text nodes one can extend the definition of forest grammars to include a set of *external predicates* P. The purpose of external predicates is to express properties which cannot be captured via content models. In particular, an external predicate can test whether a node is a text node.

An external predicate $p \in P$ is a boolean function of type $\mathcal{T}_\Sigma \mapsto \mathcal{B}$ which takes a tree as argument and returns one of the two boolean values in $\mathcal{B}$, *true* or *false*. A *forest grammar with external predicates* is a tuple $G = (\Sigma, X, P, R, r_0)$ with Σ, X and r_0 as before and $R \subseteq (X \times \Sigma \times \mathcal{R}_X) \cup (X \times P)$. As before, we denote a production $(x, a, r) \in (X \times \Sigma \times \mathcal{R}_X)$ or $(x, p) \in (X \times P)$ as $x \rightarrow a\langle r\rangle$ or $x \rightarrow p$, respectively. Intuitively, a production $x \rightarrow p$ is applicable in a derivation of the tree t when the predicate p is true for t.

Formally, the *tree derivation* relation $\mathcal{D}eriv_{R,x} \in \mathcal{T}_\Sigma \times \mathcal{T}_X$ and the *forest derivation* relation $\mathcal{D}eriv_{R,r} \in \mathcal{F}_\Sigma \times \mathcal{F}_X$ defined by set of productions R together with a distinguished non-terminal $x \in X$ and a regular expression $r \in \mathcal{R}_X$ respectively, are correspondingly extended as follows:

$$\begin{array}{lll}
(a\langle f\rangle, x\langle f'\rangle) \in \mathcal{D}eriv_{R,x} & \text{iff} & x \rightarrow a\langle r\rangle \in R \text{ and } (f, f') \in \mathcal{D}eriv_{R,r} \\
(t, x) \in \mathcal{D}eriv_{R,x} & \text{iff} & x \rightarrow p \text{ and } p(t) = true \\
(t_1 \ldots t_n, t'_1 \ldots t'_n) \in \mathcal{D}eriv_{R,r} & \text{iff} & x_1 \ldots x_n \in [\![r]\!] \text{ and } (t_i, t'_i) \in \mathcal{D}eriv_{R,x_i} \\
 & & \text{for } i = 1, \ldots, n \\
(\varepsilon, \varepsilon) \in \mathcal{D}eriv_{R,r} & \text{iff} & \lambda \in [\![r]\!]
\end{array}$$

The *meaning* $[\![R]\!]$ of R is similarly given by:

$$\begin{array}{l}
t \in [\![R]\!]\, x \text{ iff there is } t' \in \mathcal{T}_X \text{ with } (t, t') \in \mathcal{D}eriv_{R,x} \\
f \in [\![R]\!]\, r \text{ iff there is } f' \in \mathcal{F}_X \text{ with } (f, f') \in \mathcal{D}eriv_{R,r}
\end{array}$$

Finally, the language of G is as before $\mathcal{L}_G = [\![R]\!]\, r_0$.

Thereby, one can express that a node within a content model is a text node by referring it via a new terminal x_{text} for which a production $x_{text} \rightarrow p$ exists where p is a predicate testing whether its argument is a sequence of Unicode characters, i.e.:

$$p(t) = \begin{cases} true\ , & \text{if } t = \alpha^* \\ false, & \text{otherwise} \end{cases}$$

In general, predicates can be used to specify arbitrary properties of subtrees which are not expressible with the original forest grammars formalism. For example, one can use them to express that some text nodes have a required datatype, as needed in XML schema languages like XML Schema [XML01] or RelaxNG [OAS01].

4.1.2.3 Wild Card Symbols

In a practical specification language one is often interested in merely indicating the occurrence of some entity without further specifying it. For this purpose special place-holder symbols, also called *wildcards*, have to be provided. In the forest grammar formalism we use the wildcard "$*$" to denote an arbitrary label of the node and "." to denote an arbitrary node. Furthermore, we use "_" as an abbreviation for ".*", to denote an arbitrary sequence of nodes.

Example 4.6: Let R be the following set of productions:

$$\begin{array}{l} x_1 \rightarrow a\langle \_ (x_1|x_2) \_ \rangle \\ x_2 \rightarrow *\langle x_b \, . \rangle \\ x_b \rightarrow b\langle \_ \rangle \end{array}$$

The grammar $G = (R, x_1|x_2)$ specifies the language of documents in which there is a path from the root to a node the ancestor nodes of which are all labeled with a, and which has two children, the first a b node and the second an arbitrary node. □

4.1.3 Forest Grammars and XML Schema Languages

As previously mentioned, XML schema languages basically specify forest grammars. There are however some differences in terms of expressiveness that we address here.

DTDs, as opposed to forest grammars, do not allow the specification of context-dependent content models for elements – as presented in the following example:

Example 4.7: Consider the modification of the productions from Example 4.1 as follows:

$$\begin{array}{lll} x_{book} & \rightarrow & \texttt{BOOK}\langle x_{title}\ x^?_{subtitle}\ x^+_{chapter_1}\ x^?_{appendix}\rangle \\ x_{chapter_1} & \rightarrow & \texttt{CHAPTER}\langle x_{title}\ (x_{chapter_2}|x_{par})^+\rangle \\ x_{chapter_2} & \rightarrow & \texttt{CHAPTER}\langle x_{title}\ x^+_{par}\rangle \\ x_{appendix} & \rightarrow & \texttt{APPENDIX}\langle x^+_{chapter_1}\rangle \end{array}$$

Note that we specify different content models for chapters on the top-level and chapters occurring inside other chapters. Rather than allowing arbitrarily nested chapters as in Example 4.1, the new productions only allow top-level chapters to contain sub-chapters. This is not possible to express with a DTD where all elements with the same name must be associated with the same content model. □

Another limitation of DTDs as compared to forest grammars is the requirement of content models to be *unambiguous*, i.e. that the corresponding finite string automata, as obtained by the Berry-Sethi construction, are deterministic. The restriction ensures that every word can be unambiguously parsed using a lookahead of one symbol.

Similarly to forest grammars and as opposed to DTDs, XML Schema and RelaxNG allow both to specify context-dependent and non-deterministic content models. In contrast to forest grammars, they also allow the specification of the datatype of the text nodes more precisely, for example whether it should represent an integer, a float or a date. This goes beyond the basic capabilities of forest grammars. However, it is possible to define this kind of requirements using forest grammars with external predicates, by using a new non-terminal and a corresponding external predicate for each basic type needed, which tests whether the text can be converted into a value of the corresponding required type.

Another feature provided by schema languages (DTDs and XML Schema, not RelaxNG) is specifying uniqueness and reference constraints. Uniqueness constraints are used to ensure that there are no two elements with the same property, e.g. with an identical value of an attribute with a given name. Reference constraints are meant to ensure that a property of an element identifies an existing property of another element, e.g. that the value of an attribute of an element identifies another element which contains the same value in another attribute. This kinds of constraints cannot be expressed using forest grammars. While this is an important feature, it does not actually belong to the structural constraints and can be handled in applications after checking conformance to the schema.

4.2 Recognizing Regular Forest Languages

In this section we briefly review how the structural constraints specified via forest grammars can be efficiently checked.

Neumann showed that the expressive power of forest grammars is equal with that of regular tree grammars [Neu00]. That is, for every forest grammar G there is exactly one regular tree grammar G' such that the ranked tree language specified by G' is exactly the image of the forest language specified by G through an encoding function which maps every forest to a ranked tree. One such encoding can be obtained by representing the arbitrarily long sequences of sibling nodes in a similar way to how lists are represented in functional programming languages, via a binary constructor *cons* and a nullary constructor *nil*.

Therefore, testing the membership of a forest in a regular forest language (specified by a forest grammar G) is equivalent to testing the membership of its ranked encoding in the corresponding regular (ranked) tree language (specified by the regular tree grammar G'). It is well known that regular tree languages are recognized by bottom-up tree automata [GS97]. Hence, recognizing regular forest languages could be in principle solved using the classic bottom-up tree automata. In fact, most of the research literature handling XML processing use this to ignore the unrankedness of XML trees.

Nevertheless, this has a few drawbacks. Firstly, in a practical setting it requires a supplementary overhead for the encoding step. Secondly, some natural one-to-one correspondences between the XML data model and the tree representation, as for example tree relationships, are not directly recognizable in the encoded tree. In contrast, constructions using the original unranked representations are more straightforward and easy to realize in practice. Therefore, we prefer to use a unranked variant of tree automata. One straightforward approach to recognizing regular forest languages is to use bottom-up forest automata [Neu00, NS98a] (also known as unranked tree automata [BKMW01]). However, their implementation may be very expensive [Neu00, NS98a].

As expressive as bottom-up automata but much more concise and efficient to implement in practice are the *pushdown forest automata* [NS98a, Neu00]. Any implementation of a bottom-up automaton has to choose a traversal strategy for the input tree. The idea of a pushdown forest automaton (PA) is based on the observation that, when reaching a node during the traversal, the information gained from the already visited part of the tree can be used at the transitions of the automaton at that node. This supplementary information allows a significant reduction in the size of the states and in the number of possible tran-

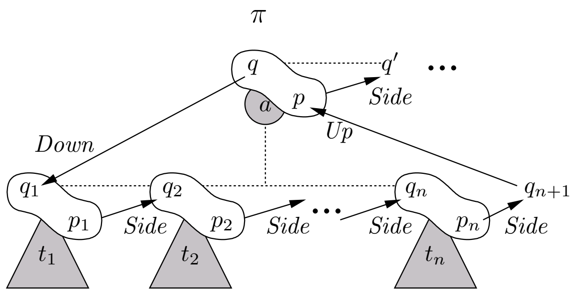


Figure 4.3: The processing model of a pushdown forest automaton

sitions to be considered by a deterministic PA as compared to the equivalent deterministic bottom-up automaton. Intuitively, in the case of a depth-first, left-to-right traversal, the advantage is that information gained by visiting the left siblings as well as the ancestors and their left siblings can be taken into account before processing the current node. The name of the automata (pushdown forest automata) is due to the fact that information from the visited part of the tree is stored on the stack (pushdown) which is implicitly used for the tree traversal.

Another advantage of PAs over bottom-up automata is that they can visit the elements in an XML input exactly in the order in which these are read from the input. Consequently, they do not need to materialize the tree representation of the input in memory, as they can handle the XML elements as they come, in an event-driven manner. This makes PAs suitable for applications in which the tree cannot be built in main memory, as for example in the case of very large XML documents. We take up again this topic in more detail in Chapter 7.

4.2.1 Pushdown Forest Automata

In addition to the tree states of classic tree automata, a PA also has *forest states*. Intuitively, a forest state contains the information gained from the already visited part of the tree (*context information*) at any point during the tree traversal. Let us consider a depth-first, left-to-right traversal. The following notations are essentially those introduced in [Neu00].

The behavior of a *left-to-right pushdown forest automaton* (LPA) is depicted in Figure 4.3, the notations of which are used in the following explanation. When arriving at some node π labeled a, the context information is available in the forest state q by which the automaton reaches the node. The automaton has to traverse the content of π and compute a tree state p, which describes π within the context q. In order to do so, the children of π are recursively processed. The context information for the first child, q_1, is obtained (via a *Down* transition) by refining q by taking into account that the father is labeled a. Subsequently the q_2 context information for the second child is obtained (via a *Side* transition) from q_1 and the information p_1 gained from the traversal of t_1. Proceeding in this manner, after visiting all children of π, enough context-information is collected in q_{n+1} in order to compute p (via an *Up* transition). After processing π, the context information for the subsequent node is updated into q'.

Formally, an LPA $A = (P, Q, I, F, Down, Up, Side)$ over an alphabet Σ consists of a finite set of *tree states* P, a finite set of *forest states* Q, a set of *initial states*

$I \subseteq Q$, a set of *final states* $F \subseteq Q$, a *down-relation* $Down \subseteq Q \times \Sigma \times Q$, an *up-relation* $Up \subseteq Q \times \Sigma \times P$ and a *side-relation* $Side \subseteq Q \times P \times Q$. Based on *Down*, *Up* and *Side*, the behavior of A is described by the relations $\delta^A_{\mathcal{F}} \subseteq Q \times \mathcal{F}_\Sigma \times Q$ and $\delta^A_{\mathcal{T}} \subseteq Q \times \mathcal{T}_\Sigma \times P$ as follows, where the notations correspond to those in Figure 4.3:

1. $(q, a\langle t_1 \ldots t_n\rangle, p) \in \delta^A_{\mathcal{T}}$ iff $(q, a, q_1) \in Down$, $(q_1, t_1 \ldots t_n, q_{n+1}) \in \delta^A_{\mathcal{F}}$ and $(q_{n+1}, a, p) \in Up$ for some $q_1, q_{n+1} \in Q$.
2. $(q_1, t_1 f, q_{n+1}) \in \delta^A_{\mathcal{F}}$ iff $(q_1, t_1, p_1) \in \delta^A_{\mathcal{T}}$, $(q_1, p_1, q_2) \in Side$ and $(q_2, f, q_{n+1}) \in \delta^A_{\mathcal{F}}$ for some $p_1 \in P, q_2 \in Q$
3. $(q_1, \varepsilon, q_1) \in \delta^A_{\mathcal{F}}$ for all $q_1 \in Q$

The language accepted by the automaton A is given by:

$$\mathcal{L}_A = \{f \in \mathcal{F}_\Sigma \mid q_1 \in I, q_2 \in F \text{ and } (q_1, f, q_2) \in \delta^A_{\mathcal{F}}\}$$

An LPA performs a depth-first, left-to-right traversal of the input. Similarly, if we consider a depth-first, right-to-left traversal we obtain a *right-to-left push-down forest automaton* (RPA). An RPA $A = (P, Q, I, F, Down, Up, Side)$ is similar to an LPA but, as it proceeds on a forest from the right to the left, case 2 from above is replaced by:

2'. $(q_{n+1}, t_1 f, q_1) \in \delta^A_{\mathcal{F}}$ iff $(q_{n+1}, f, q_2) \in \delta^A_{\mathcal{F}}$, $(q_2, t_1, p_1) \in \delta^A_{\mathcal{T}}$ and $(q_2, p_1, q_1) \in Side$ for some $q_2 \in Q, p_1 \in P$.

If the *Down*, *Up* and *Side* transitions of a PA are functions rather than relations and there is exactly one start state, the PA is called *deterministic*. Otherwise, it is called *non-deterministic*. If a PA is deterministic we write $Down(q, a) = q_1$, $Side(q_1, p_1) = q_2$ and $Up(q_{n+1}, a) = p$ rather than $(q, a, q_1) \in Down$, $(q_1, p_1, q_2) \in Side$ and $(q_{n+1}, a, p) \in Up$, respectively.

4.2.2 From Forest Grammars to Pushdown Forest Automata

A compilation schema from a forest grammar $G = (R, r_0)$ into a deterministic LPA (DLPA) accepting the same regular forest language that we briefly review here has been given in [Neu00]. The idea is that the DLPA keeps at any time track of all possible content models of the elements whose content has not yet been seen in its entirety. The forest is accepted at the end if the sequence of top-level nodes conforms to r_0.

Let $r_1, \ldots, r_l$ be the regular expressions occurring on the right-hand sides in the productions R, where l is the number of productions. For $0 \leq j \leq l$, let $A_j = (Y_j, y_{0,j}, F_j, \delta_j)$ be the non-deterministic finite automaton (NFA) accepting the regular string language defined by r_j, as obtained by the Berry-Sethi construction (presented in Section 2.1). Recall that Y_j is the set of NFA states, $y_{0,j}$ the start state, F_j the set of final states and $\delta_j \subseteq Y_j \times \Sigma \times Y_j$ is the transition relation.

By possibly renaming the NFA states we can always ensure that $Y_i \cap Y_j = \emptyset$ for $i \neq j$. Let $Y = Y_0 \cup \ldots \cup Y_l$ and $\delta = \delta_0 \cup \ldots \cup \delta_l$. A DLPA $A_{\vec{G}}$ accepting $\mathcal{L}_G$ can be defined as $A_{\vec{G}} =(2^X, 2^Y, \{q_0\}, F, Down, Up, Side)$, where X is the set of non-terminals in G. A tree state synthesized for a node is the set of non-terminals from which the node can be derived. A forest state consists of the

NFA states reached within the possible content models of the current level and can be computed as follows.

We start with the content model r_0, i.e.:

$$q_0 = \{y_{0,0}\}$$

We accept the top level sequence of nodes if it conforms to r_0, i.e.:

$$F = \{q \mid q \cap F_0 \neq \emptyset\}$$

The possible content models of a node are computed from the content models in which the node may occur:

$$Down(q, a) = \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle r_j \rangle\}$$

When finishing a sequence of siblings we consider only the fulfilled content models in order to obtain the non-terminals from which the father node may be derived:

$$Up(q, a) = \{x \mid x \to a\langle r_j \rangle \text{ and } q \cap F_j \neq \emptyset\}$$

The possible content models are updated after finishing visiting the next node in a sequence of siblings:

$$Side(q, p) = \{y_1 \mid y \in q, x \in p \text{ and } (y, x, y_1) \in \delta\}$$

The resulting $A_{\vec{G}}$ is obviously deterministic, since it has one initial state and its transitions are functions rather than relations.

Example 4.8: The NFAs for the regular expressions occurring in grammar G with the set of rules specified in Example 4.3 (on page 31) are depicted in Figure 4.4. Consider as input the XML document depicted in Figure 4.1 (on page 31). The run of $A_{\vec{G}}$ on the tree representation of the input is shown in Figure 4.5, where the sets containing x-s are tree states and the sets containing y-s are forest states. The order in which the tree and forest states are computed is denoted by the subscripts at their right. Observe that the input tree, which is in the regular forest language specified by G, is accepted by $A_{\vec{G}}$ as it stops in the state $\{y_1\}$, which is a final state of the LPA. □

4.3 Bibliographic Notes

Originally, Neumann and Seidl have used *μ-formulas* [NS98b] and later *constraint systems* [NS98a] to specify regular forest languages. Forest grammars, as a more comprehensible mean of specifying regular forest languages, have been introduced in [Neu00], as an adaption of tree grammars from the ranked to the unranked tree case. An overview on how results from the ranked tree-theory carry over in general to the unranked case is presented by Brüggemann-Klein *et al.* in [BKMW01].

The correlation between the most popular available schema languages and regular forest languages has been studied by Murata *et al.* [MLM01]. For each

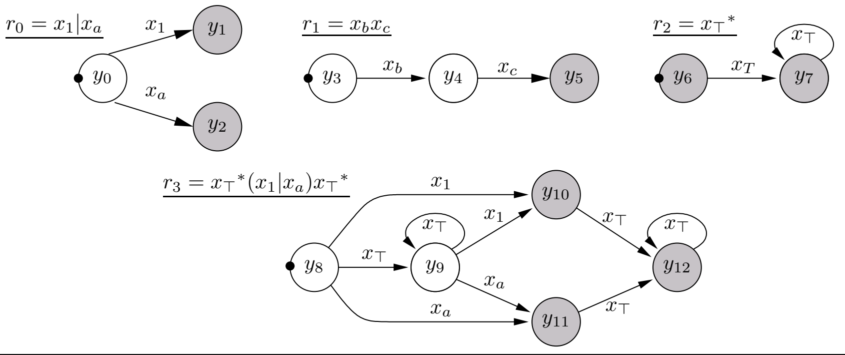

$r_0 = x_1|x_a$
$r_1 = x_bx_c$
$r_2 = x_\top{}^*$
$r_3 = x_\top{}^*(x_1|x_a)x_\top{}^*$
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12}
x_1 x_a x_b x_c $x_\top$

Figure 4.4: NFAs obtained by Berry-Sethi construction for regular expressions in Example 4.5

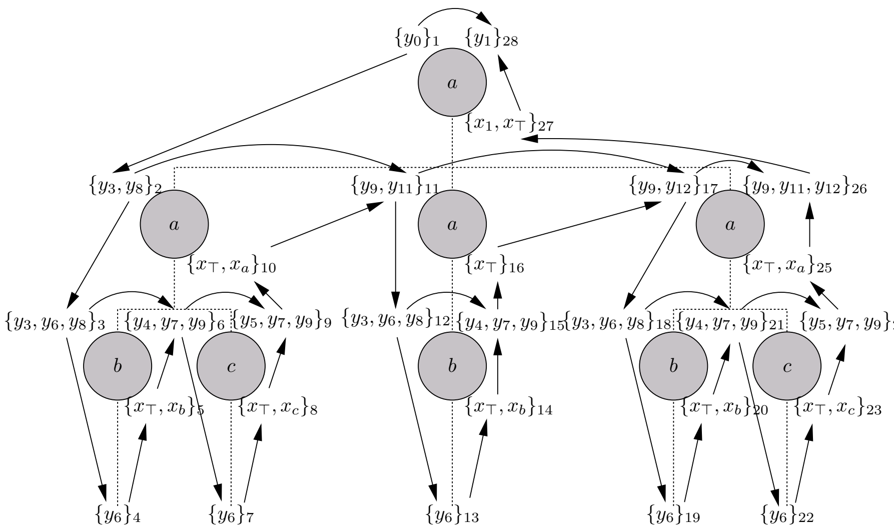

$\{y_0\}_1$ $\{y_1\}_{28}$
$\{x_1, x_\top\}_{27}$
$\{y_3, y_8\}_2$ $\{y_9, y_{11}\}_{11}$ $\{y_9, y_{12}\}_{17}$ $\{y_9, y_{11}, y_{12}\}_{26}$
$\{x_\top, x_a\}_{10}$ $\{x_\top\}_{16}$ $\{x_\top, x_a\}_{25}$
$\{y_3, y_6, y_8\}_3$ $\{y_4, y_7, y_9\}_6$ $\{y_5, y_7, y_9\}_9$ $\{y_3, y_6, y_8\}_{12}$ $\{y_4, y_7, y_9\}_{15}$ $\{y_3, y_6, y_8\}_{18}$ $\{y_4, y_7, y_9\}_{21}$ $\{y_5, y_7, y_9\}_{24}$
$\{x_\top, x_b\}_5$ $\{x_\top, x_c\}_8$ $\{x_\top, x_b\}_{14}$ $\{x_\top, x_b\}_{20}$ $\{x_\top, x_c\}_{23}$
$\{y_6\}_4$ $\{y_6\}_7$ $\{y_6\}_{13}$ $\{y_6\}_{19}$ $\{y_6\}_{22}$
a b c

Figure 4.5: The run of $A_G^{\vec{}}$ on an input tree

considered schema language a corresponding restriction of regular forest languages is identified. An algorithm for checking the conformance to a schema is presented for each such subclass.

Checking conformance to a schema which is a regular forest language can be very efficiently performed by a pushdown forest automaton as presented in this chapter. Pushdown forest automata have been introduced by Neumann and Seidl in [NS98a]. They show that every non-deterministic pushdown forest automaton can be made deterministic and that they are much more concise when compared to bottom-up automata. They also give a compilation schema from constraint systems to deterministic pushdown forest automata. The compilation of forest grammars to deterministic pushdown forest automata was introduced in [Neu00].

Chapter 5

Grammar Queries

Locating parts of documents with specific properties is a fundamental task in document processing and in particular in XML applications. We refer to this process as querying. Querying is used on its own in order to extract information from documents. Furthermore, especially in the context of XML, where documents are often dynamically created from different XML sources, querying accomplishes the basic function of locating sub-components used for creating new content in XML transformations. The importance of query-languages becomes apparent if one notes that XPath [XPa99], the XML query language proposed by the W3C Consortium, is integral part of many other important specifications, for example XML Schema Language [XML01], XSLT [XSL99] or XQuery [XQu05a].

In Section 5.1 we present how forest grammars can be used to specify powerful XML queries. Forest grammars queries form the basis of the Fxgrep pattern language presented in Chapter 3. Most of the attention in the study of XML query languages has been drawn by *unary queries*, which locate individual nodes from the input tree. In contrast, in Section 5.1, we present a formalism which can express *k-ary queries*, which are able to locate *k* nodes which simultaneously satisfy a specific property. In Section 5.2 we review an efficient construction based on pushdown automata which can be used to find matches of unary grammar queries. An efficient construction for locating binary queries is presented in Section 5.3. Finally, implementing *k*-ary queries in general is addressed in Section 5.4.

5.1 Specifying Queries

One possible way of identifying nodes of interest, as required by the task of querying, is to label them with special symbols. In this respect, derivations according to a forest grammar, which, as seen in the previous chapter, are relabeling of input forests, can be used as a means of specifying queries. The definitions of grammar queries, given in the remainder of this section pursue this observation.

5.1.1 Unary Queries

As previously suggested, given a grammar G, a non-terminal x specifies a query by identifying all nodes π in the input f for which there is a derivation f' w.r.t. G in which π is labeled with x. More generally, a *unary grammar query* Q is a pair (G, T) consisting of a forest grammar $G = (R, r_0)$ and a set of *target non-terminals* $T \subseteq X$ where X is the set of non-terminals in R. The *matches* of Q in an input forest f are given by the set $\mathcal{M}_{Q,f} \subseteq N(f)$ as follows:

$$\pi \in \mathcal{M}_{Q,f} \text{ iff } \exists (f, f') \in \mathcal{D}eriv_G, \exists x \in T \text{ and } lab(f'[\pi]) = x$$

We say that π is a match of Q in f w.r.t. the derivation f'.

Example 5.1: Consider the grammar G from Example 4.5 (on page 32). The query $Q_1 = (G, \{x_b\})$ locates nodes b having only a ancestors and only one sibling c to the right. The leftmost b in the input tree depicted in Figure 4.1 is a match, as one can see by definition by looking at the first derivation in Figure 4.2 (on page 31). Similarly, the rightmost b is a match as defined by the second derivation w.r.t. G.

The query $Q_2 = (G, \{x_a\})$ locates the a nodes which have a child b followed by a child c. These are the leftmost and the rightmost a nodes.

□

In general, as suggested in the example above, a single grammar can be flexibly used to specify many similar yet different queries, one for each non-terminal. This flexibility is in contrast with pattern languages, where for each query a significantly different pattern has to be specified.

Note that we decided for an *all-matches* semantic of our queries, i.e. all nodes π as in the definition are to be reported as matches. This is reasonable, because a user query typically is aimed at finding *all* locations with the specified properties, as for instance in XPath . Furthermore, we do not want to place on the user the burden of specifying the query via an unambiguous grammar, therefore the definition above refers to *any* derivation.

5.1.2 K-ary Queries

The definition of queries given in the previous section can be straightforwardly extended in order to identify k-tuples of nodes related via structural constraints, as imposed by forest grammars.

A *k-ary grammar query* is a pair $Q = (G, T)$ consisting of a forest grammar $G = (R, r_0)$ and a k-ary relation $T \subseteq X^k$ where X is the set non-terminals in R. The *matches* of Q in an input forest f are given by the k-ary relation $\mathcal{M}_{Q,f} \subseteq N(f)^k$:

$$(\pi_1, \ldots, \pi_k) \in \mathcal{M}_{Q,f} \text{ iff } \begin{array}{l} \exists (f, f') \in \mathcal{D}eriv_G, \exists (x_1, \ldots, x_k) \in T \text{ and} \\ lab(f'[\pi_i]) = x_i \text{ for } i = 1, \ldots, k \end{array}$$

We say that $(\pi_1, \ldots, \pi_k)$ is a match of Q in f w.r.t. the derivation f'. For $k = 1$ and $k = 2$ we obtain unary and binary queries, respectively.

Example 5.2: Consider the grammar G from Example 4.5 (on page 32). The binary query $Q_2 = (G, \{(x_b, x_c)\}$ locates pairs of nodes b and c having as father the same node a, and only a ancestors. The leftmost b and c in the input depicted in Figure 4.1 (on page 31) form a match pair, as one can see by definition by looking at the first derivation in Figure 4.2. Similarly, the rightmost b and c form a match pair as defined by the second derivation w.r.t. G. $\square$

5.1.3 Expressive Power of Grammar Queries

As noted in Section 4.2, forest grammars are as expressive as regular tree grammars. The proof by Neumann [Neu00] shows that for every forest grammar G there is exactly one regular tree grammar G' s.t. the language specified by G' is the image of the language specified by G through a bijective function *enc* mapping every unranked tree (or forest) to a unique ranked tree representation.

In particular, *enc* can be chosen s.t. an arbitrarily long sequence of sibling nodes is represented similarly to the way that lists are represented in functional programming languages via two constructor nodes *cons* and *nil*, with arity 2 and 0 respectively. This mapping ensures that every node in a forest f corresponds to exactly one node in $enc(f)$. Moreover, the construction presented in [Neu00] ensures that for every non-terminal x in G there is exactly one non-terminal x' in G' such that a (forest) derivation of some input forest f labeling a node with x exists iff a (tree) derivation of the ranked encoding $enc(f)$ exists labeling the corresponding node in the encoding with x'.

According to the definition of k-ary grammar queries, this implies that a tuple $(\pi_1, \pi_2, \ldots, \pi_k)$ of nodes $\pi_i \in N(f)$ is a match of a query $(G, (x_1, x_2, \ldots, x_k))$ iff the corresponding tuple of nodes $(\pi'_1, \pi'_2, \ldots, \pi'_k)$ of nodes $\pi'_i \in N(enc(f))$ is a match of a query $(G', (x'_1, x'_2, \ldots, x'_k))$. Therefore, the expressive power of forest grammar queries is equal to that of regular tree grammar queries.

It is well known that the class of languages specified by regular tree grammars (the regular ranked tree languages) is exactly the same as the class of languages specified by formulas of monadic second order logic (MSO) on trees without free variables [TW68]. Using this result, and casting the problem of finding matches of regular tree grammar queries into a language recognition problem, one can show that the expressive power of k-ary tree grammar queries is equal with that of MSO formulas with k free variables. A proof can be consulted in [NPTT05]. We conclude that the expressive power of our k-ary grammar queries is equal to that of MSO formulas with k free variables.

Queries specified directly via MSO formulas are not practicable due to their high evaluation complexity, yet they have been used as convenient benchmarks for comparing XML query languages [NS02] due to their large expressiveness. Indeed MSO queries subsume many of the fundamental features of the query languages which have been proposed for XML (as it will be presented in Section 5.6). Grammar queries have thus the same expressive power as MSO queries, while being efficiently implementable, at least in the unary and binary case, as we show in the next sections.

5.2 Recognizing Unary Queries

A construction for answering unary grammar queries using pushdown forest automata has been presented in [NS98a, Neu00]. In the present section we briefly review this construction. Knowing this construction helps under-

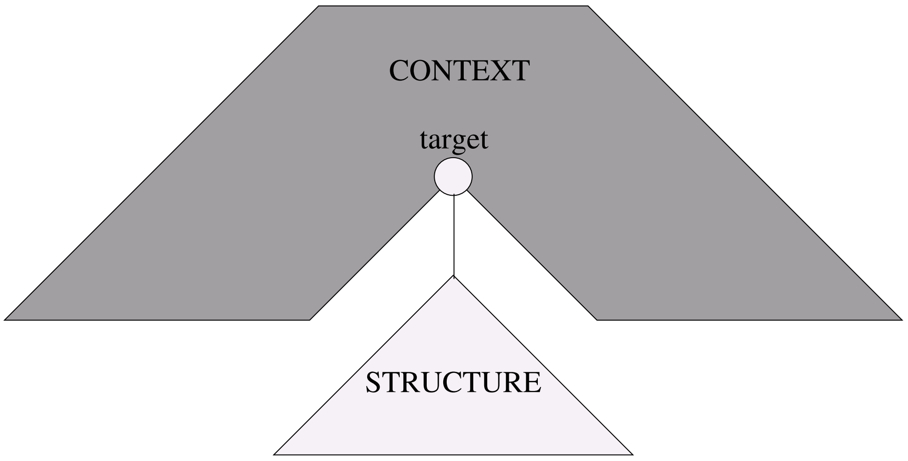


Figure 5.1: The contextual and the structural part of a query

standing its generalization for binary and k-ary queries which is presented in Section 5.3 and Section 5.4, respectively.

Specifying which are the subtrees of interest in a query typically consists of two conceptual parts, as described in Figure 5.1. The contextual part constrains the surrounding context of the subtrees of interest, whereas the structural part describes the properties of the subtrees themselves.

Example 5.3: Supposing we have an XML document which represents a conference article, where sections and subsections are encoded as XML elements, we might be interested in *subsections containing the word "automata"* occurring *in sections whose title contain the word "forest"*. The two emphasized parts denote the structural and the contextual part of the query, respectively. □

Example 5.4: As seen in Example 5.1 (on page 42), the query $Q_1 = (G, \{x_b\})$ locates the b nodes (structure) which have only a ancestors and a right c sibling (context). □

When specifying a query as a grammar $G = (R, r_0)$ together with a distinguished non-terminal x, one specifies at once the desired structure and context of some subtree t in a forest f. The structure is described by the productions which can be used in order to derive a tree t starting from x. The remaining productions of the grammar, which constrain the locations where x can occur in a derivation of f from r_0, capture the context part of the specification.

As argued in Section 4.2.1 a PA uses its forest states to remember information from the already visited part of the input. Therefore, by looking into the forest state of the PA after visiting a subtree t it should be possible to check a structural property of t as well as whether a contextual property can be satisfied considering the part of the context seen so far.

Example 5.5: Let Q_1 be the unary query from Example 5.1, identifying b nodes which have only a ancestors and only one c sibling to the right. Consider the run of the corresponding LPA on the input as depicted in Figure 4.5 (on page 39). One can see that by the time the automata has seen any of the b nodes, each of them fulfills the structural part (it is a b node) and the upper-left contextual part (all ancestors are a nodes). This is reflected in the forest states of the LPA when it leaves each of the b nodes, depicted at the upper right of each of them, respectively. In each of these forest states, the NFA state y_4, which is reached after reading an x_b, denotes that a derivation of the input forest may exist in which the respective node is labeled x_b.

However, since the right part of the context has not yet been seen, the LPA cannot decide at the time it leaves the b nodes whether they are indeed matches. $\square$

In order to decide whether a node is a match, in general, the remaining part of the context also has to be seen. The idea is to *remember* for each node the information collected after seeing only a part of the context and to let a second automaton proceed from the opposite direction (i.e. to perform a depth-first, right-to-left traversal if the first PA does a left-to-right traversal) in order to account for the remaining context.

Before proceeding in Section 5.2.2 to the construction based on the two PA runs for the evaluation of a grammar queries, we introduce in Section 5.2.1 a couple of useful notations which allow us to speak about the states of the PAs at a certain location.

5.2.1 Pushdown Forest Automata as Relabelings

A run of a deterministic PA on an input forest f can be seen as a *relabeling* of each node in f with the triple of states involved in the transitions at that node during the run[1].

Consider a DLPA A as defined in Section 4.2.1 (on page 36). The relabeling of f performed by A is a mapping $\vec{\lambda} : N(f) \to Q \times P \times Q$, $\vec{\lambda}(\pi i) = (\vec{q}_{\pi(i-1)}, \vec{p}_{\pi i}, \vec{q}_{\pi i})$, where, for the node πi, $\vec{q}_{\pi(i-1)}$, $\vec{p}_{\pi i}$ and $\vec{q}_{\pi i}$ are the forest state in which the node is reached, the tree state synthesized for the node and the forest state in which the node is left respectively, by A, i.e.:

$$\begin{array}{ll} \vec{q}_{\lambda 0} & = \vec{q}_0 \text{ (the initial state)} \\ \vec{q}_{\pi 0} & = Down(\vec{q}_\pi, a) \\ \vec{p}_\pi & = Up(\vec{q}_{\pi n}, a), \text{ if } n = last_f(\pi) \\ \vec{q}_{\pi i} & = Side(\vec{q}_{\pi(i-1)}, \vec{p}_{\pi i}) \end{array}$$

where $a = lab(f[\pi])$.

Similarly, a deterministic RPA (DRPA) B can be seen as a relabeling $\overleftarrow{\lambda}(\pi i) = (q_{\pi(i-1)}, p_{\pi i}, q_{\pi i})$, where $q_{\pi i}$, $p_{\pi i}$ and $q_{\pi(i-1)}$ are the forest state in which the node is reached, the tree state synthesized for the node and the forest state in which the node is left, respectively.

5.2.2 Locating Unary Matches

The state in which a DLPA leaves a node π synthesizes all the information collected after seeing the upper left context and all the content of π. Given this information, a second (DRPA) automaton, proceeding from right to left, will

[1]For a visualization, observe Figure 4.3 on page 36 where for the node denoted π, the above mentioned states correspond to q, p and q'.

have at every node the information necessary in order to decide whether the node fulfills the structural and contextual requirements of a query.

Consider a unary query (G, T). Let $A_{\vec{G}}$ be the DLPA accepting the language of grammar G, constructed as in Section 4.2.2 (on page 37). We now present how to construct the second DRPA $B_{\overleftarrow{G}}$ for the given grammar G. In the following we use notations as introduced in Section 5.2.1. That is, given a node π, we denote by $\vec{p}_\pi$ and $\vec{q}_\pi$ the tree state synthesized for π and the forest state in which π is left by $A_{\vec{G}}$, respectively. For $B_{\overleftarrow{G}}$, we denote by q_π and p_π, the forest state in which π is reached and the tree state synthesized for π by the DRPA, respectively.

By remembering $\vec{q}_\pi$ one can locally decide at each node during a second traversal of the input by $B_{\overleftarrow{G}}$ whether the node is a match of a query. Also, to avoid unnecessary re-computations by $B_{\overleftarrow{G}}$, $\vec{p}_\pi$ is remembered so as to account for the structure information collected at π.

The automaton $B_{\overleftarrow{G}}$ runs thus on an *annotation* $\vec{f}$ of the input forest f by $A_{\vec{G}}$, $\vec{f} \in \mathcal{F}_{\Sigma \times P \times Q}$ with $N(\vec{f}) = N(f)$ and $lab(\vec{f}[\pi]) = (lab(f[\pi]), \vec{p}_\pi, \vec{q}_\pi)$ for all $\pi \in N(f)$.

The construction of $B_{\overleftarrow{G}}$ is similar to that of $A_{\vec{G}}$ but follows the NFA transitions in reverse and considers corresponding NFA final states at rightmost siblings, as the input to the NFAs is seen from the right to the left. Additionally, $B_{\overleftarrow{G}}$ takes into account information collected by $A_{\vec{G}}$ in order to avoid considering NFA transitions which were not relevant for the conformance check performed by $A_{\vec{G}}$. The automaton $B_{\overleftarrow{G}} = (2^X, 2^Y, \{F_0\}, \emptyset, \mathit{Down}^{\leftarrow}, \mathit{Up}^{\leftarrow}, \mathit{Side}^{\leftarrow})$, where X, Y and F_0 are as in the definition of $A_{\vec{G}}$, is given by:

$$\begin{array}{ll}
\mathit{Down}^{\leftarrow}(q, (a, \vec{p}, \vec{q})) & = \{y_2 \mid y \in q \cap \vec{q},\ (y_1, x, y) \in \delta,\ x \to a\langle r_j \rangle \text{ and } y_2 \in F_j\} \\
\mathit{Up}^{\leftarrow}(q, (a, \vec{p}, \vec{q})) & = \vec{p} \\
\mathit{Side}^{\leftarrow}(q, p, (a, \vec{p}, \vec{q})) & = \{y \mid (y, x, y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\}
\end{array}$$

where we also provide the $\mathit{Side}^{\leftarrow}$ transition with the label $(a, \vec{p}, \vec{q})$ of the node over which it is executed.

Note that $p_\pi = \vec{p}_\pi$ for all π. When it is clear from the context which is the label $(a, \vec{p}, \vec{q})$ at a transition we will omit this argument.

The following proposition by Neumann [Neu00] shows how for every node π, the forest state q_π in which $B_{\overleftarrow{G}}$ arrives at π, containing information from the right context can be combined with the information for the remaining part of the input given in the annotation $\vec{q}_\pi$ in order to find matches of a unary query. A node is a match if both the forest states in which $A_{\vec{G}}$ leaves the node and in which $B_{\overleftarrow{G}}$ arrives at the node contain an NFA state reachable after seeing a target non-terminal from T.

Theorem 5.1: Let $Q = (G, T)$ be a unary query and $f \in \mathcal{L}_G$. With $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$ as above, $\pi \in \mathcal{M}_{Q,f}$ iff $y_1 \in q_\pi \cap \vec{q}_\pi$ and $(y, x, y_1) \in \delta$ for some $y, y_1 \in Y$ and $x \in T$.

This theorem is proven in [Neu00] as Theorem 7.1. □

Directly from Theorem 5.1 follows the corollary:

Corollary 5.1: $(f, f') \in \mathcal{D}eriv_G$ and $lab(f'[\pi]) = x$ iff $y \in q_\pi \cap \vec{q}_\pi$, $(y_1, x, y) \in \delta$ for some $y, y_1 \in Y$.

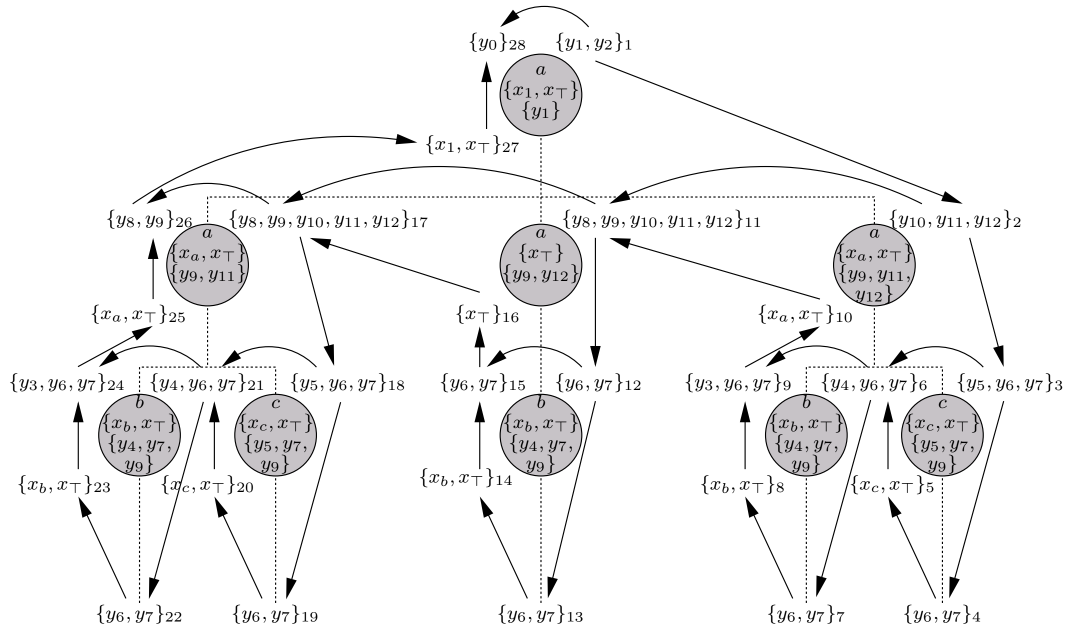


Figure 5.2: The run of the $B_{\overleftarrow{G}}$ on the input document annotated by the $A_{\vec{G}}$ in Example 4.8

This further implies that:

Corollary 5.2: If $(f, f') \in \mathcal{D}eriv_G$ and $lab(f'[\pi]) = x$, then $x \in p_\pi$.

By Corollary 5.1 there are $y \in q_\pi \cap \vec{q}_\pi$, $(y_1, x, y) \in \delta$. Since $y \in \vec{q}_\pi$, it follows by the definition of *Side* in $A_{\vec{G}}$ that there is $(y', x_1, y) \in \delta$ for some $x_1 \in p_\pi$. By the Berry-Sethi construction, all incoming transitions into an NFA state y are labeled with the same symbol. Therefore, $x_1 = x$ and thus $x \in p_\pi$. □

Example 5.6: Consider the run of $A_{\vec{G}}$ depicted in Figure 4.5 (on page 39). The run of $B_{\overleftarrow{G}}$ on the tree annotated by $A_{\vec{G}}$ is presented in Figure 5.2. The order in which the tree and forest states are computed is denoted by the subscripts at their right. Note how the rightmost b node is recognized as a match of the query $Q_1 = (G, \{x_b\})$. As noted in Example 5.5, the NFA state y_4 (having an incoming transition labeled x_b) in the annotation done by $A_{\vec{G}}$ denotes the node as a potential match after accounting for its upper left context and its content. The conformance of the right context is also fulfilled as the forest state in which $B_{\overleftarrow{G}}$ arrives at the node contains y_4 as well. Similarly, the leftmost b node is a match. On the contrary, the node b in the middle is not a match, as its right context does not contain a c sibling as required by the query. □

5.3 Recognizing Binary Queries

In this section we present a construction which allows the efficient evaluation of binary queries. The construction is based on the technique introduced in the previous section for the evaluation of unary queries. As opposed to unary queries, where the decision whether a node is a match can be taken when the node is visited by the second automaton, finding a match pair of a binary query requires postponing the decision at least until both nodes in the pair have been visited. We thus need a supplementary construction which allows the remembering of information distributed upon the tree, and the use of this information to detect matches.

We introduce the necessary construction in Section 5.3.1 and show how it can be used to efficiently evaluate a slightly restricted class of binary queries. In Section 5.3.2 we show how the approach works for general binary queries.

5.3.1 Recognizing Simple Binary Queries

Let $Q = (G, B)$ be a binary query. For convenience, we will first assume that $B = \{(x^1, x^2)\}$ for some $x^1, x^2 \in X$, where X is the set of non-terminals from G. We call such a query a *simple binary query*.

According to the definition, a pair (π_1, π_2) is a match for an input f iff there is a derivation f' of f w.r.t. G and $f'[\pi_1] = x^1$, $f'[\pi_2] = x^2$.

Observe that this implies that π_1 and π_2 are matches of the unary queries (G, x^1) and (G, x^2), respectively. Thereby, (π_1, π_2) is a binary match for Q iff:

(p) π_1 is a match of the unary query (G, x^1) and

(s) π_2 is a match of the unary query (G, x^2) and

(r) π_1 and π_2 are unary matches w.r.t. the same derivation f'.

We call the nodes fulfilling (p) and (s) *primary* and *secondary* matches, or, for short, *primaries* and *secondaries*, respectively.

We have already seen how unary matches can be located. Thus, testing (p) and (s) can be done by an automata construction as in Section 5.2. In order to implement binary queries, however, one must additionally be able to test (r).

5.3.1.1 Construction

In the following we show that binary queries can be efficiently answered by using a run of a DLPA $A_{\vec{G}}$ followed by a run of a DRPA $B_{\overleftarrow{G}}$, in a way which is similar to the case of unary queries. The $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$ automata are defined exactly as in Section 5.2.2. Primary and secondary matches can be thus recognized in the same way as in Section 5.2.2 and we keep the same notations as there.

In order to locate binary matches, we have to remember during the run of $B_{\overleftarrow{G}}$ which of the already visited nodes are primary or secondary matches, as potential components of binary matches. We accumulate these primaries and secondaries in set attributes l_1 and l_2, respectively, with which we equip each element of the tree and forest states of $B_{\overleftarrow{G}}$.

For a tree state p at node π and $x \in p$, $x.l_1$ contains primary matches and $x.l_2$ secondary matches which are found below π and are defined w.r.t. derivations which label $f[\pi]$ with x.

Similarly, for a forest state q at node π and $y \in q$, $y.l_1$ contains primary and $y.l_2$ secondary matches collected from the already visited right-sibling subtrees of $f[\pi]$. These are the matches defined w.r.t. derivations in which the word of non-terminals on the current level is accepted by an NFA reaching the current location in state y.

Similarly to attribute grammars, the values of the l_1 and l_2 attributes are defined by a set of local rules, as follows:

- For the elements of a forest state in which $B_{\overleftarrow{G}}$ arrives at a node π which has no right-siblings (i.e. π is the rightmost node among its siblings), the sets of primaries and secondaries collected from the right sibling subtrees are obviously empty. This is the case for the initial state F_0 at the root and for the states obtained by executing a $\mathit{Down}^{\leftharpoonup}$ transition:

  $$\text{If } y \in F_0 \text{ or } y \in \mathit{Down}^{\leftharpoonup}(q,(a,\vec{p},\vec{q})), \text{ then } y.l_1 = \emptyset,\ y.l_2 = \emptyset$$

- After finishing visiting the children of a node π, the sets of primaries and secondaries found below π are propagated and possibly updated with π if π is a primary or secondary match, respectively:

  $$\text{If } x \in \mathit{Up}^{\leftharpoonup}(q,(a,\vec{p},\vec{q})), \text{ then}$$

  $$x.l_1 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \to a\langle r_j \rangle\}, & \text{if } x = x^1 \\ \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \to a\langle r_j \rangle\} & , \text{ otherwise} \end{cases}$$

  $$x.l_2 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \to a\langle r_j \rangle\}, & \text{if } x = x^2 \\ \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \to a\langle r_j \rangle\} & , \text{ otherwise} \end{cases}$$

- At side transitions over a node π labeled with $(a,\vec{p},\vec{q})$, the list of primaries and secondaries found so far are obtained by combining the matches below π with the matches from the already visited part to the right:

  $$\text{If } y \in \mathit{Side}^{\leftharpoonup}(q,p,(a,\vec{p},\vec{q})), \text{ then}$$

  $$\begin{aligned} y.l_1 &= \bigcup\{y_1.l_1 \cup x.l_1 \mid (y,x,y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\} \\ y.l_2 &= \bigcup\{y_1.l_2 \cup x.l_2 \mid (y,x,y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\} \end{aligned}$$

Note that the rules allow a bottom-up, right-to-left evaluation of the attributes. Therefore, they can be evaluated directly along the run of $B_{\overleftarrow{G}}$, which performs a depth-first, right-to-left traversal. Moreover, the information used for the evaluation of attributes at a node π is the same as the information needed to compute the transitions at π. In the practical implementation (which will be addressed in Section 5.5), where transitions are computed as they are needed during the run of $B_{\overleftarrow{G}}$, the attributes can be thus computed at minimal costs.

Example 5.7: Consider the binary query $Q_2 = (G, \{(x_b, x_c)\}$ from Example 5.2, locating the b and the immediately following c children of a node a whose ancestors are exclusively a nodes, on the tree depicted in Figure 4.1 (on page 31). Figure 5.3 depicts how the l_1 and l_2 attributes are computed along the run of $B_{\overleftarrow{G}}$ on the input annotated by the run of $A_{\overrightarrow{G}}$ (shown in Figure 4.5 on

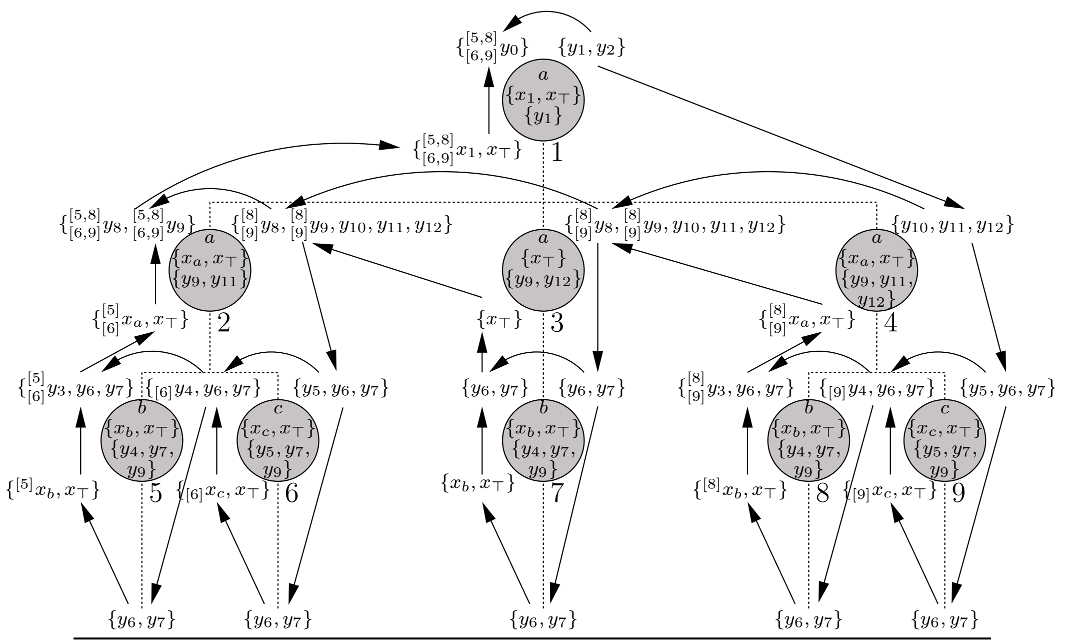


Figure 5.3: Evaluation of the l_1 and l_2 attributes

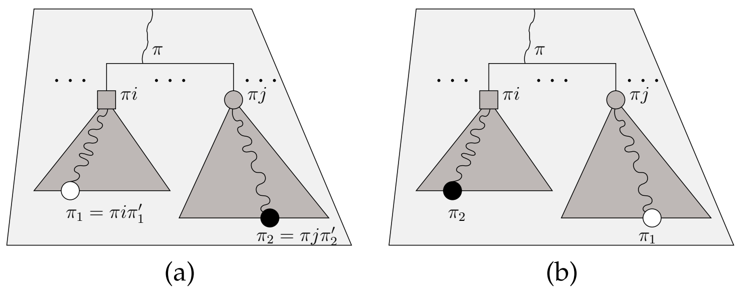


Figure 5.4: Relative positions of matches: π is nearest common ancestor or λ

page 39). The order of computation performed by the second automaton is the same as in the unary case (which was depicted in Figure 5.2 on page 47). Note that nodes are identified by ordinal numbers rather than by paths in order to increase readability. The attributes l_1, l_2 for an element x are depicted as ${}^{l_1}_{l_2}x$. Attributes with value $\emptyset$ are omitted. $\square$

5.3.1.2 Locating Binary Matches

Figure 5.4 (a) and (b), and Figure 5.5 (c), (d) and (e) show all possible relative positions of the primary (depicted in white) and the secondary component (depicted in black) of one binary match (π_1, π_2). In all five situations, due to the construction above, π_1 and π_2 belong to the attributes of one of the tree state $p_{\pi i}$ or forest state $q_{\pi i}$ in which the automaton reaches node πi (depicted by a square). This is where the binary match (π_1, π_2) will be detected at the $Side^{\leftarrow}(q_{\pi i}, p_{\pi i})$ transition.

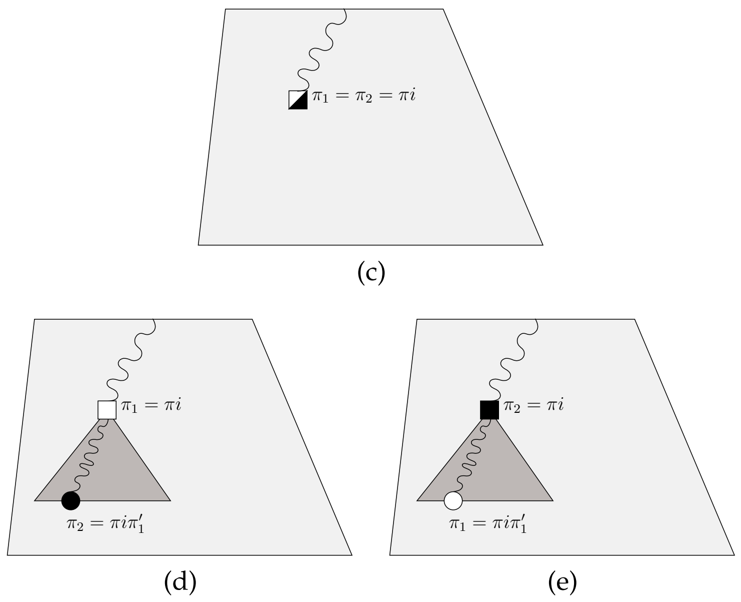


Figure 5.5: Relative positions of matches: equal, or one is a proper ancestor of the other

To see how, we need to observe that our construction ensures the following invariants:

(i_1) A node π_1 belongs to the l_1 or l_2 attribute of an element x of a tree state computed for a node πi iff π_1 is below πi and there is a derivation of the input forest which labels πi with x and π_1 with x^1 or x^2, respectively.

(i_2) A node π_2 belongs to the l_1 or l_2 attribute of an element y of a forest state in which $B_G^{\leftarrow}$ arrives at a node πi iff π_2 is in some right sibling subtree and there is a derivation of the input forest which labels πi with x, the label of the NFA transitions coming into y, and π_2 with x^1 or x^2, respectively.

This is formally expressed by the following theorem in which the involved nodes are named as in Figure 5.4 (a) (or (b)):

Theorem 5.2: (*Invariants ensured by the construction*)

(i_1) If $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $x \in p_{\pi i}$, $(y', x, y) \in \delta$ for some y', x then

$\pi_1 \in x.l_1$ (or $\pi_1 \in x.l_2$) iff

$\pi_1 = \pi i \pi'_1$, $\exists f_1$ s.t. $(f, f_1) \in \mathcal{D}eriv_G$, $lab(f_1[\pi i]) = x$ and $lab(f_1[\pi_1]) = x^1$ (or $lab(f_1[\pi_1]) = x^2$, respectively).

(i_2) $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $x \in p_{\pi i}$, $(y', x, y) \in \delta$ and $\pi_2 \in y.l_2$ (or $\pi_2 \in y.l_1$) iff

$\pi_2 = \pi j \pi'_2$, $j > i$, $\exists f_2$ s.t. $(f, f_2) \in \mathcal{D}eriv_G$, $lab(f_2[\pi i]) = x$ and $lab(f_2[\pi_2]) = x^2$ (or $lab(f_2[\pi_2]) = x^1$, respectively)

A formal proof is given in Appendix A.2. The idea is presented forthwith. □

Let $x \in p_{\pi i}$, $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $(y', x, y) \in \delta$. Let $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$. It is easy to see that (i_1) directly implies (p) and (i_2) implies (s). Less obvious but still true is that (i_1) and (i_2) also imply (r). It follows that every pair formed with $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$ is a binary match.

To see why (i_1) and (i_2) imply (r), let us define a function which given a forest f, a node π and a tree t constructs a forest f_1 by replacing in f the subtree located at π with t, formally $f_1 = f/^{\pi}\, t$ where:

$$\begin{aligned}(t_1 \dots t_i \dots t_n)/^{i}\, t &= t_1 \dots t \dots t_n\\ (t_1 \dots t_i \dots t_n)/^{i\pi}\, t &= t_1 \dots a\langle f/^{\pi}\, t\rangle \dots t_n, \text{ if } t_i = a\langle f\rangle\end{aligned}$$

If $f_1 = f/^{\pi}\, t$, we say that f_1 is obtained by *grafting* t into f at π.

The following theorem observes that given two derivations of a forest f which label a node π with the same symbol, a new derivation can be obtained by doing a relabeling of f in which the nodes below π are labeled as in one of the derivations and the rest of nodes as in the other.

Theorem 5.3: If $(f, f_1) \in \mathcal{D}eriv_G$, $(f, f_2) \in \mathcal{D}eriv_G$ and $lab(f_1[\pi]) = lab(f_2[\pi])$ then $(f, f_1/^{\pi}\, f_2[\pi]) \in \mathcal{D}eriv_G$ and

$$lab((f_1/^{\pi}\, f_2[\pi])[\pi_1]) = \begin{cases} lab(f_2[\pi_1]), & \text{if } \pi_1 = \pi\pi_2 \text{ for some } \pi_2\\ lab(f_1[\pi_1]), & \text{otherwise}\end{cases}$$

The proof is given in Appendix A.1. □

Using the notations of Theorem 5.2, let $f' = f_2/^{\pi i}\, f_1[\pi i]$. It follows by Theorem 5.3 that $(f, f') \in \mathcal{D}eriv_G$, $f'[\pi_1] = x^1$ and $f'[\pi_2] = x^2$, thus (r) also holds for (π_1, π_2). It follows that (π_1, π_2) is a binary match.

Example 5.8: Consider the side transition at node 8 in Figure 5.3. The element $_{[9]}y_4$ in the forest state in which node 8 is reached denotes that node 9 is a secondary match in the part of the tree already visited. The element $^{[8]}x_b$ in the tree state synthesized at node 8 denotes that 8 is a primary match found in the subtree 8. The fact that 8 and 9 are defined with respect to the same derivation can be seen from the fact that x_b is the label of the incoming transitions into y_4. Thus $(8, 9)$ is a binary match.

Similarly, $(5, 6)$ is detected as a match at the side transition at node 5. □

Thereby, we obtain how binary matches can be detected (where cases (a)-(e) correspond to the situations depicted in Figure 5.4 and Figure 5.5):

(a) Every pair (π_1, π_2) with $\pi_1 \in x.l_1$, $\pi_2 \in y.l_2$ is a binary match, as presented above.

(b) Similarly, one can show that every pair (π_1, π_2) with $\pi_1 \in y.l_1$, $\pi_2 \in x.l_2$ is a binary match.

(c) If $x = x^1 = x^2$ it is easy to see in invariant (i_1) that by definition $(\pi i, \pi i)$ is a binary match.

(d) If $x = x^1$ we also have by (i_1) that every pair $(\pi i, \pi_2)$ with $\pi_2 \in x.l_2$ is a binary match.

(e) Similarly, if $x = x^2$ we have by (i_1) that every pair $(\pi_1, \pi i)$ with $\pi_1 \in x.l_1$ is a binary match.

To see that *all* binary matches are detected as above, let, conversely, (π_1, π_2) be a binary match. If $\pi_1 = \pi i \pi_1'$ and $\pi_2 = \pi j \pi_2'$, $j > i$ then there is f' with $(f, f') \in \mathcal{D}eriv_G$, $f'[\pi i \pi_1'] = x^1$ and $f'[\pi j \pi_2'] = x^2$. Let $f'[\pi i] = x$. It follows by Corollary 5.1 that there are $y' \in q_{\pi i} \cap \vec{q}_{\pi i}$, $(y_1', x, y') \in \delta$. By Corollary 5.2 we have that $x \in p_{\pi i}$. By (i_1) it follows that $\pi_1 \in x.l_1$. By (i_2) there are $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $x \in p_{\pi i}$, $(y_1, x, y) \in \delta$ and $\pi_2 \in y.l_2$. It follows that there is πi, $x \in p_{\pi i}$, $y \in \vec{q}_{\pi i} \cap q_{\pi i}$, $(y_1, x, y) \in \delta$, $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$.

Similarly, for $\pi_2 = \pi i \pi_2'$, $\pi_1 = \pi j \pi_1'$, $j > i$, or $\pi_1 = \pi_2$, or $\pi_2 = \pi_1 i \pi_2'$, or $\pi_1 = \pi_2 i \pi_1'$ we obtain the converse of (b), (c), (d) or (e), respectively.

We have thus proven the following theorem:

Theorem 5.4: A pair (π_1, π_2) is a binary match iff there is $\pi \in N(f)$, $x \in p_\pi$, $y \in q_\pi \cap \vec{q}_\pi$, $(y', x, y) \in \delta$ and either:

(a) $\pi_1 \in x.l_1$, $\pi_2 \in y.l_2$ or

(b) $\pi_1 \in y.l_1$, $\pi_2 \in x.l_2$ or

(c) $\pi_1 = \pi_2 = \pi$, $x = x^1 = x^2$ or

(d) $\pi_1 = \pi$, $x = x^1$, $\pi_2 \in x.l_2$ or

(e) $\pi_2 = \pi$, $x = x^2$, $\pi_1 \in x.l_1$.

Complexity

Let n be the size of the input forest f, i.e. the number of nodes in f. The complexity of answering a binary query is given by the complexities of running $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$, computing the l_1 and l_2 attributes and that of locating binary matches.

The automaton $A_{\vec{G}}$ executes at each node one *Down*, one *Side* and one *Up* transition. As one can see in the definitions of the transitions, the time cost of each of these transitions does not depend on f. The run of $A_{\vec{G}}$ requires thus time $\mathcal{O}(n)$. Similarly, the run of $B_{\overleftarrow{G}}$ needs time $\mathcal{O}(n)$.

The l_1 and l_2 attributes have to be computed for each component of the states obtained after a $Side^{\leftarrow}$ and $Up^{\leftarrow}$ transition. For the complexity assessment let us suppose that m is the larger of the numbers of primary and secondary matches in f.

Consider now an $Up^{\leftarrow}$ transition. The set $x.l_1$ of primaries for each component is computed as the union of the sets $y.l_1$ of primaries. As the number of sets $y.l_1$ does not depend on f, and a set union can be computed in time $\mathcal{O}(m)$, the time for computing $x.l_1$ is in $\mathcal{O}(m)$. Similarly, $x.l_2$ is computed in time $\mathcal{O}(m)$. As the number of elements in the computed state does not depend on f either, executing $Up^{\leftarrow}$ can be done in time $\mathcal{O}(m)$. The sets $y.l_1$ and $y.l_2$ computed at $Side^{\leftarrow}$ transition for each component of the state are similarly computed in time $\mathcal{O}(m)$. It follows that the attributes can be computed in time $\mathcal{O}(n \cdot m)$.

As for the complexity of locating matches, let p be the number of binary matches in f. Note that each of the binary matches is located at exactly one of the *Side*$^{\leftharpoonup}$ transitions, namely at the *Side*$^{\leftharpoonup}$ transitions over the ancestor of one of the primary or secondary, which is a sibling of an ancestor of the other. As remembering each binary match only requires constant time, locating binary matches has the overall time cost in $\mathcal{O}(p)$.

The total time cost of answering binary queries is thus in $\mathcal{O}(n \cdot m + p)$. Since $p \leq m^2$ and $m \leq n$, the theoretical worst cost is in $\mathcal{O}(n^2)$. This corresponds to the case in which every pair of nodes from f is a binary match. In practice, however, the number of primary, secondary and binary matches tend to be irrelevant as compared to the input size. In this case, the time consumed is rather linear in the input size and binary queries can be answered almost as efficiently as unary queries.

5.3.2 Recognizing General Binary Queries

Let $Q = (G, T)$, where $T \subseteq X^2$, be a binary query. The construction is similar to that for simple binary queries but has to keep a set attribute for each nonterminal occurring in T.

Formally, let $X_1 = \{x \mid (x, x') \in T \text{ or } (x', x) \in T\} = \{x_1, \ldots, x_n\}$.

Rather than with two attributes as in the case of simple binary queries, we equip each element of a state in which $B_G^{\leftrightharpoons}$ visits the input with n attributes $l_1, \ldots, l_n$. The attributes l_i are computed as follows:

- ✧ If $y \in F_0$ (the initial state of $B_G^{\leftrightharpoons}$) or $y \in Down^{\leftharpoonup}(q, (a, \vec{p}, \vec{q}))$ then $y.l_i = \emptyset$
- ✧ If $x \in Up^{\leftharpoonup}(q, (a, \vec{p}, \vec{q}))$ then

$$x.l_i = \begin{cases} \{\pi\} \cup \bigcup\{y.l_i \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\}, & \text{if } x = x_i \\ \bigcup\{y.l_i \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\} & , \text{ otherwise} \end{cases}$$

- ✧ If $y \in Side^{\leftharpoonup}(q, p, (a, \vec{p}, \vec{q}))$ then

$$y.l_i = \bigcup\{y_1.l_i \cup x.l_i \mid (y, x, y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\}$$

for $i = 1, \ldots, n$.

As in the case of simple binary queries, matches are found at *Side*$^{\leftharpoonup}$ transitions of $B_G^{\leftrightharpoons}$. Let $Side^{\leftharpoonup}(q_\pi, p_\pi)$ be such a transition and let $x \in p_\pi$, $y \in q_\pi \cap \vec{q}_\pi$, $(y_1, x, y) \in \delta$. In order to find binary matches, one has to look for every $(x_i, x_j) \in T$ into the l_i and l_j attributes. Finding the match pairs is achieved similarly to finding match pairs in the case of simple binary matches.

Theorem 5.5: A pair (π_1, π_2) is a binary match iff there is $\pi \in N(f)$, $(x_i, x_j) \in T$, $x \in p_\pi$, $y \in q_\pi \cap \vec{q}_\pi$, $(y_1, x, y) \in \delta$ and either:

(a) $\pi_1 \in x.l_i$, $\pi_2 \in y.l_j$ or

(b) $\pi_1 \in x.l_j$, $\pi_2 \in y.l_i$ or

(c) $\pi_1 = \pi_2 = \pi, x = x_i = x_j$ or

(d) $\pi_1 = \pi, x = x_i, \pi_2 \in x.l_j$ or

(e) $\pi_1 = \pi, x = x_j, \pi_2 \in x.l_i$.

By definition, (π_1, π_2) is a binary match iff there is $(x_i, x_j) \in T$ and (π_1, π_2) is a simple binary match for $(G, (x_i, x_j))$. The proof follows immediately from Theorem 5.4 by observing that the attributes l_1 and l_2 from the construction for $(G, (x_i, x_j))$ equal l_i and l_j, respectively. $\square$

In a similar manner as in the case of simple binary queries one obtains that the complexity of answering binary queries is quadratic in the input size in the worst case and rather linear in the average case.

5.4 Recognizing *K*-ary Queries

The construction introduced for the evaluation of binary queries can be in principle extended to work for *k*-ary queries for arbitrary *k*'s. In order to locate matches of a query $(G, (x_1, \ldots, x_k))$ the construction has to keep a separate set attribute for each non-empty subset $A \subset \{x_1, \ldots, x_k\}$. The set attribute for A then contains all tuples of nodes which form a partial match corresponding to the elements in A. This is necessary because a complete match can be obtained by considering any pair of complementary partial matches. For example, for a query $(G, (x_1, x_2, x_3))$ one needs to consider putting together the partial matches corresponding to $\{x_1\}$ and $\{x_2, x_3\}$, or $\{x_2\}$ and $\{x_1, x_3\}$, or $\{x_3\}$ and $\{x_1, x_2\}$, respectively. However, the complexity of the construction grows exponentially with *k* which makes it impracticable for large *k*'s.

In the XML practice however many queries are expressed via XPath select patterns which conceptually are binary relations (namely, between the context node for the evaluation of the pattern and the set of nodes selected in that context). Therefore, binary queries can be satisfactorily used to cover a wide range of actual XML applications, as we discuss in the next part of the work.

Nevertheless, it is possible to implement *k*-ary queries very efficiently if one adopts a *disambiguating policy* for grammars. Our queries so far consider *all* possible derivations w.r.t. the given input grammar. Following all these derivations in parallel is the source of the exponential blowup in the evaluation complexity. A disambiguating policy is a set of rules which allows the choice of exactly *one* derivation from among the different derivations.

One disambiguating policy could be obtained for instance by requiring one to (1) always consider *left-longest* sequences in fulfillments of content models, i.e., in NFA runs, to prefer NFA transitions corresponding to symbols which are as left as possible in the corresponding regular expressions; and (2) always choose the first applicable production in the input grammar. A similar policy was in essence originally adopted in XDuce [HP00, HP03], in the context of its functional style pattern matching for XML documents (a comparison of XDuce with our approach is provided in Section 5.6). Similarly, one can adopt a *right-longest* policy, or non-deterministically choose one of the possible derivations, if one is for example interested in just one match, as in the case of a *one-match* policy.

The implementation in the presence of such a disambiguating policy can use the same first traversal of $A_{\vec{G}}$ to annotate the input tree. Only the second automaton $B_{\overleftarrow{G}}$ has to proceed differently. The adopted policy allows the maintenance of its states as singletons, rather than sets, by indicating exactly one NFA transition (y_1, x, y), and exactly one grammar production $x \rightarrow a\langle r_j \rangle$ to be considered at the $\overleftarrow{Side}$ and $\overleftarrow{Down}$ transitions, respectively. The x's considered at the NFA transitions are then the labels of the sought-after derivation. The k match nodes can be thus directly read from the annotation by the second automata, getting thus even linear time complexity.

5.5 Implementation Issues

The constructions presented in the previous section for the evaluation of unary and binary queries have been completely implemented in Fxgrep . We refrained from implementing general k-ary queries due to their exponential evaluation complexity. The efficient implementation of unary queries was presented in detail by Neumann [Neu00]. We briefly review here a few aspects which are important in the practical implementation in order to support efficiency and ease of use.

Lazy Evaluation The pushdown automata are efficiently implemented by computing their transitions only as they are needed. Transitions which are not required for the traversal of the input are not computed. This avoids the computation of possibly exponentially large transition tables. The number of transitions that are actually computed is at most linear in the size of the input document.

Caching Moreover, the automata do not need to compute transitions at every node, as many transitions are repeatedly executed. The first time a transition is needed, its computed value is cached, and the cached value is simply looked up for its subsequent uses. In practice only few transitions need to be computed even for large XML documents.

Pre-processing Further, information which is repeatedly used for the computation of transitions, and which does not depend on the input document can be computed by a preprocessor of the query and directly accessed when needed. For example, a transition $Down(q, a)$ is computed (only when the automaton $A_{\vec{G}}$ arrives in forest state q at a node labeled a, and only if the transition was not already computed) using the definition:

$$Down(q, a) = \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle r_j\rangle \text{ for some } x, y_1\}$$

To do so it can use the following pre-processed information:

$$\begin{array}{l} y_0s\_for\_y\ y = \{y_{0,j} \mid (y, x, y_1) \in \delta,\ x \to a\langle r_j\rangle\} \text{ for all } y \in Y \\ y_0s\_for\_a\ a = \{y_{0,j} \mid x \to a\langle r_j\rangle\} \text{ for all } a \text{ occurring in } G \end{array}$$

Therefore:

$$Down(q, a) = \begin{cases} y_0s\_for\_a\ a \cap \bigcup\limits_{y \in q} y_0s\_for\_y\ y, & \text{if } a \text{ occurs in } G \\ \emptyset & , \text{ otherwise} \end{cases}$$

Similar information is computed by the preprocessor for supporting the other transitions of the pushdown automata.

5.5.1 From Patterns to Grammar Queries

As presented in Chapter 3, Fxgrep allows the specification of queries also by using a more intuitive pattern language, rather than via grammar queries. Internally, patterns are automatically translated to grammar queries. In the following we show via a few examples how patterns can be automatically translated to grammar queries. A more formal and detailed translation schema for most of the Fxgrep pattern language can be found in [Neu00].

Basically, given a pattern, the corresponding grammar has a non-terminal for each symbol in the pattern denoting some XML node. For example, the pattern `/a/b/c` is translated to the query $(G, \{x_c\})$ where $G = (R, x_a)$, and R is the set productions:

$$\begin{array}{lcl} x_a & \rightarrow & \texttt{<a>}\_\, x_b \,\_\texttt{</a>} \\ x_b & \rightarrow & \texttt{<b>}\_\, x_c \,\_\texttt{</b>} \\ x_c & \rightarrow & \texttt{<c>}\_\texttt{</c>} \end{array}$$

The `a` element denoted by x_a has a child denoted by x_b which is preceded and followed by an arbitrary number of siblings. Given the same grammar G, the query (G, x_b) is equivalent to the pattern `/a/b[c]` and $(G, \{x_a\})$ to `a[b[c]]`.

As presented in Chapter 3, binary queries can be specified via binary Fxgrep patterns, by using a symbol "`%`" which may be placed anywhere in front of a node inside a pattern to indicate the secondary match position. Binary patterns are similarly automatically translated into binary grammar queries. The grammar productions are obtained as in the case of unary patterns. The primary target non-terminal is obtained as the non-terminal corresponding to the target node of the unary pattern. The secondary target non-terminal is the non-terminal corresponding to the node preceded by the "`%`" symbol. For example, the pattern `a/%b/c` is translated to the grammar query $(G, (x_c, x_b))$ with G as above.

Structural constraints for a node are directly reflected in the rule corresponding to the node. For example `/a/b[c+ d*]` is queried by $((R, x_a), \{x_b\})$ with the productions:

$$\begin{array}{lcl} x_a & \rightarrow & \texttt{<a>}\_\, x_b \,\_\texttt{</a>} \\ x_b & \rightarrow & \texttt{<b>}\_\, x_c^+ x_d^* \,\_\texttt{</b>} \\ x_c & \rightarrow & \texttt{<c>}\_\texttt{</c>} \\ x_d & \rightarrow & \texttt{<d>}\_\texttt{</d>} \end{array}$$

More than one structural constraint for a node is reflected in the grammar by productions with conjunctions of content models. Conjunctions of content models are addressed in Chapter 6. Intuitively, a rule containing a conjunction specifies that each content model in the conjunction has to be fulfilled by a node derived via that production and can be used to simultaneously specify more structural constraints. Note that, in a pattern, a structural constraint is given either explicitly in square brackets following the concerned node, or implicitly as the continuation of the path in which the node occurs. For example `/a[b]/c[d][e]` is matched by $((R, x_a), \{x_c\})$ with the productions:

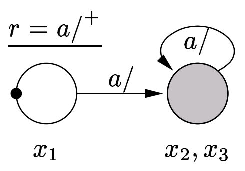


Figure 5.6: Finite automaton for a vertical regular path

$x_a \longrightarrow$ `<a>` $\_\, x_b\, \_$ & $\_\, x_c\, \_$ `</a>`
$x_b \longrightarrow$ `<b>` $\_$ `</b>`
$x_c \longrightarrow$ `<c>` $\_\, x_d\, \_$ & $\_\, x_e\, \_$ `</c>`
$x_d \longrightarrow$ `<d>` $\_$ `</d>`
$x_e \longrightarrow$ `<e>` $\_$ `</e>`

When a pattern contains a regular vertical path, the translation is guided by the finite automaton recognizing the regular path. To every state of the automaton for the vertical context we associate a new non-terminal, whereas the transitions correspond to the productions of the grammar accounting for context. An additional non-terminal is associated to final states in order to account for the structure. Consider, e.g., the pattern `(a/)+b` for which the automaton is depicted in Figure 5.6. The corresponding grammar query is $((R, x_1), \{x_3\})$ with the productions:

$x_1 \longrightarrow$ `<a>` $\_\, x_2 \mid x_3\, \_$ `</a>`
$x_2 \longrightarrow$ `<a>` $\_\, x_2 \mid x_3\, \_$ `</a>`
$x_3 \longrightarrow$ `<b>` $\_$ `</b>`

5.6 Bibliographic Notes

In this section we briefly survey existing query approaches for tree-structured data and relate them with ours where possible. The research on XML processing has been extremely prolific in the recent years. No claim can be made with regard to the exhaustiveness of the presentation.

We consider the following comparison criteria:

Expressiveness As a benchmark for assessing the expressive power of query languages for trees we use the monadic second order logic (MSO logic), which has proven to be particularly convenient in the context of tree-automata and logic-based approaches [NS03].

Extensibility to the *k*-ary case Most of the proposals only consider the unary case. We indicate where a query approach can be straightforwardly extended to express *k*-ary queries.

Evaluation complexity An important factor for practical implementations is how efficient are the proposed query languages. Despite the relatively large number of proposals, there are not many for which practical algorithms have been introduced, and even fewer practical query languages are actually implemented.

Some of the approaches have been summarized in a survey done by Neven and Schwentick [NS03], which mainly addresses only the unary queries. A somewhat dated overview on practical XML query languages can be found in [FSW99]. We start with a very brief summary of our grammar queries.

Forest Grammar Queries

The specification of unary queries using forest grammars has been introduced in [Neu00]. The definition of a match node of a query is given there *locally*, recursively in terms of the queries matched by the father node and the structural constraints fulfilled by the sibling nodes. In order to make our unary query definition extensible to the k-ary case we gave it in the global terms of derivations, which denote *global* computations over the input [BS02, BS04]. As presented in Section 5.1.3, the expressive power of our k-ary queries equals MSO formulas with k free variables. Evaluating unary grammar queries is done in time linear in the size of the input. Binary queries can be always evaluated in quadratic time in the worst case, yet most queries are answered in linear time as discussed in Section 5.3.1.2. Unary and binary grammar queries are completely implemented in the practical XML querying tool Fxgrep [NB05]. The complexity of evaluating k-ary queries generally grows exponentially with k. Nevertheless, by adopting a disambiguating policy to chose one among the different derivations allowed by a grammar, it is possible to evaluate k-ary queries in linear time, as discussed in Section 5.4.

Query Automata

Query automata (QA) defined by Neven and Schwentick [NS99] are *two-way tree automata*, i.e. automata which can perform both up and down transitions, together with a distinguished set of *selecting* states. A query automata is a unary query locating the nodes of some input which are visited at least once in a selecting state. Simple QA on unranked trees are less expressive even than first order logic. Intuitively, the reason for this limitation is due to their inability to pass information from one sibling to another. To achieve the expressive power of MSO logic, QA have to be extended with *stay transitions*, at which a two way string transducer reads the string of states of the children of some node and outputs for each child a new state. Queries of arity k are not considered. The complexity of query evaluation is linear in the size of the input [NS03].

Selecting Tree Automata

Similarly to query automata, Frick *et al.* [FGK03] use tree automata extended with a set of selecting states (called *selecting tree automata*) to specify unary queries. The semantics of queries is defined in terms of *runs* of the tree automata, either in an *existential* or a *universal* setting. In the existential setting, a node in some input is a match, if *some* accepting *run* on the input visits the node in a selecting state. In the universal one, a node is a match if *every* accepting *run* on the input visits the node in a selecting state. It is shown that the existential and the universal queries are equally expressive.

Runs of tree automata are in fact the same as derivations w.r.t. tree grammars, with automata states corresponding to non-terminals in grammars. The existential queries are thus basically unary forest grammar queries restricted to the ranked case. The proof that the expressiveness of selecting tree automata equals MSO logic unary queries comes therefore as no surprise.

For the evaluation of queries an algorithm is proposed which performs a bottom-up followed by a top-down traversal of the algorithm, the complexity of which is $O(m \cdot n)$, where m is the size of the encoding of the automaton and n the size of the input.

The selecting automata defined in [FGK03] are only used to express unary queries. As previously argued, the grammar queries formalism presented in Section 5.1.2 and introduced in [BS02] and [BS04] is in fact a generalization of the existential query formalism based on selecting automata, to k-ary queries on unranked trees. Recently, Niehren *et al.* [NPTT05] consider both existential and universal k-ary queries defined via selecting tree automata and show that they are equally expressive also in the k-ary case. Furthermore, [NPTT05] provide a proof that the k-ary queries defined by selecting tree automata capture precisely the MSO expressible queries, and that this result carries over to unranked trees. From this follows that grammar queries have the same expressiveness as MSO queries, as already mentioned in Section 5.1.3. Another implication is that forest grammar queries, defined using an existential semantics, also capture universal queries.

As a particularly efficiently implementable case, Niehren *et al.* identify k-ary queries specified via *unambiguous tree automata*, i.e. tree automata for which for every input tree there is at most one successful run. The proposed construction identifies the unique run in a bottom-up followed by a top-down traversal of the input and has a time-complexity linear in the input size. Specifying queries by unambiguous tree automata is similar to specifying a grammar query together with a disambiguating policy for derivations as mentioned in Section 5.4.

Attribute Grammar Queries

Neven and Van den Bussche [NB02] used attribute grammars (AGs) to specify queries on derivation trees of context-free grammars, hence they deal with queries on ranked trees. In particular they consider *boolean-valued attribute grammars* with propositional logic formulas as semantic rules (BAG). A BAG together with a designated boolean attribute defines a unary query which retrieves all nodes at which the attribute has the value *true*. It is shown that BAG unary queries have the same expressiveness as MSO logic.

In order to deal with unranked trees, Neven extends the BAG query formalism [Nev05]. He introduces *extended attribute grammars* which work on extended context free grammars (ECFGs) rather than on CFGs. Since an ECFG production has a regular expression r on the right hand side, the number of children of a node defined via the production may be unbounded. To be able to define an attribute for each child, the semantic rules identify them via the unique position in r to which the child has to correspond. A semantic rule for a given position defines attribute values for all children corresponding to that position. The semantic rule is basically given as a regular expression r_1 containing a special place-marker symbol "#". The attribute value of a child is assigned true if the sequence of children containing the place-marker "#" in front of the child under consideration matches the regular expression r_1. This allows the specification of the left and right context of a node. Neven shows that the query formalism based on extended BAGs is equally expressive as MSO logic.

BAGs can only express unary queries. No considerations are made regarding the complexity of query evaluation.

One possibility for expressing k-ary queries with AGs has been investigated by Neven and Van den Bussche [NB02] only for ranked trees. This is achieved via *relation-valued attribute grammars* (RAG). A RAG defines a query as some designated attribute at the root. The matches of the query are given by the relation computed as the value of this attribute. They show that RAGs are more expressive than MSO for queries of any arity.

Attribute grammars have been considered also in the context of XML stream processing [NN01, KS03, SK05]. We briefly review these approaches in Section 7.3.

Regular Hedge Expressions

In [Mur01], Murata considers providing a specification language to allow for the specification of context of a node more precisely than by simply expressing conditions on the path from the root to the node. The query formalism proposed is similar to the original proposal by Neumann and Seidl using μ-formulas [NS98b]. A query is specified as a pair consisting of two expressions for specifying the structure and the context of a match node, respectively. Murata's *hedge regular expressions*, used to express the structure of a match, are able to express regular forest languages. To express the contextual condition, the formalism for specifying structure is extended by using a special symbol to denote the desired node (obtaining *pointed* formulas). Murata, previously used pointed trees for specifying contextual conditions on ranked trees in [Mur97].

The formalism presented in [Mur01] is targeted at unary queries. The evaluation time of the queries is proven to be linear in the size of the input. The expressive power of the selection queries using regular hedge expressions is equal with MSO logic.

Tree-walking Automata

Tree-walking Automata (TWAs) are sequential automata working on trees. In contrast to classic tree automata, in which the control state is distributed at more than one node, a TWA always considers exactly one node of the input. Depending on the label of the node and its location the automaton changes its state and moves to a neighbor node[2]. A TWA specifies the language of trees on which the automaton starts at the root and ends also at the root in an accepting state. TWAs can be used to specify unary queries in a similar way as the selecting tree automata, by defining a set of selecting states. Recently it was proven that TWAs cannot define all regular tree languages [BC05]. This implies that TWAs are less expressive than unary MSO queries.

[2]Pushdown forest automata can be thus also seen as a kind of tree-walking automata enhanced with a pushdown.

Brüggemann-Klein and Wood [BKW00] proposed *caterpillars* as a technique for specifying context in queries. A caterpillar is a sequence of symbols from a *caterpillar alphabet* denoting movements and node-tests. The symbols specify a movement to the parent, left- or right-sibling, left- or rightmost child of a node or testing that a node is root, leaf, the first or the last among its siblings. A caterpillar sequence specifies a unary query as the nodes starting from which the sequence of movements and tests can be successfully performed. More generally the query can be specified via *caterpillar expressions* which are regular expressions over a caterpillar alphabet. Evaluating the matches specified by a caterpillar expression can be done in $O(m \cdot n)$, where m is the size of the input and n is the number of transitions in the finite-state automaton corresponding to the caterpillar expression. The formalism can express only unary queries.

Expressing binary queries on trees via TWAs was considered by van Best in [vB98]. A tree-walking automaton can compute a binary relation on tree nodes by starting at one node and finishing at the other. It is argued that ordinary TWAs cannot compute all MSO definable binary queries. To achieve the expressiveness of MSO binary queries *tree-walking marble/pebble automata* are introduced. A tree-walking marble/pebble automaton is a TWA enhanced with a number of *marbles* and one pebble. The automaton can place and pick-up the marbles and the pebble while visiting the nodes of the input, with the restriction that the last one placed must be the first one to be picked-up. Furthermore, when placing a marble on a node, the automaton is restricted to walk only below this node. It is shown that these automata can be used to compute binary relations defined by MSO logic formulas. The complexity of query evaluation is not assessed in [vB98].

To be able to deal with XML text nodes and attributes (generically called *data values*) and to allow the use of equality tests on them in queries, two extensions of TWAs are suggested by Neven *et al.* in [NSV01, NS03]. The extensions are given for string automata but they carry over also to tree automata. In the first approach the automata are extended with a finite number of registers and can check whether the data value of a node equals the content of some register when performing its transitions. The expressive power of this register automata is comparable neither with FO logic nor with MSO logic. That is, there are FO queries which are not expressible by the automata, but also queries expressible by these automata which are not captured by any MSO query. In the second approach, the automata are equipped with a number of pebbles which they can use according to a stack policy (last dropped first removed). The expressiveness of these pebble automata is shown to lie between FO and MSO logic. The complexity of query evaluation for both register and pebble automata is shown to be in PTIME.

Pure Logic Formalisms

In logical formalism queries are expressed directly as logic formulas. A query containing a free node variable expresses a unary query. Going from unary to k-ary queries is in principle easily done by using formulas with k free variables instead of formulas with one free variable. Queries on tree-structured data can be expressed using MSO logic on trees. Many of the query approaches mentioned so far have exactly the same expressiveness as MSO logic. Nevertheless expressing queries as MSO logic formulas is not practicable due to the prohibitively high (non-elementary) evaluation complexity.

Neven and Schwentick [Sch00, NS00] consider a restriction of MSO logic as a query mechanism. The logic, called FOREG, is an extension of FO logic which allows the formulation of constraints on children of nodes and on nodes lying on a given path, called *horizontal* and *vertical path formulas*, respectively. Horizontal and vertical path formulas are regular expression over formulas which are to be satisfied by the children of a node, or the pairs of end nodes of the edges on a path, respectively. It is shown that the expressiveness of FOREG lies precisely between FO and MSO logic.

Still, the evaluation complexity of FOREG queries is not practicable. Neven and Schwentick present syntactic restrictions of FO, FOREG and MSO logic which are still as expressible as the original logics but for which unary queries can be more efficiently evaluated. The restriction, basically allows the formulation of properties only on paths from the root to a node, rather than on arbitrary paths. It is shown that unary queries expressed in the restriction of FOREG (called *guarded FOREG*) and MSO (*guarded MSO*) can be evaluated in time $O(n \cdot 2^m)$ and $O(n \cdot 2^{m^2})$, respectively, where n is the size of the input and m the size of the formula.

Going from unary to k-ary queries in the guarded logic formalism is not directly possible due to the restricted use of variables in the guarded case. Nevertheless, Schwentick shows [Sch00] that expressing queries of arbitrary arity is possible by suitable combinations of unary queries as above and an additional kind of horizontal path formula (called *intermediate path formula*) which is able to talk about the sequence of siblings between the ancestors of two arbitrary nodes. While the expressiveness of the logic formalism is not modified by the addition of intermediate path formulas, it is shown that an algorithm exists which checks in time $O(n \cdot 2^m)$ whether a tuple of nodes verifies a formula on some input. Answering queries using this algorithm implies generating all the k-tuples of nodes from the input, incurring $\mathcal{O}(n^k)$ time. This gives the evaluation of k-ary queries the $\mathcal{O}(n^{k+1} \cdot 2^m)$ complexity. In particular, binary queries can be answered thus in time $\mathcal{O}(n^3 \cdot 2^m)$, which is less efficient when compared with the complexity of our algorithm.

XPath Evaluation

Gottlob *et al.* [GKP02] find out by experimenting with practical XPath processors that their evaluation time might grow exponentially with the size of the considered XPath pattern. This inefficiency is ascribed to a naive implementation which literally follows the XPath specification as a succession of selection and filtering steps. As opposed to this evaluation strategy, the authors propose a *bottom-up* evaluation, that is, an evaluation in which expressions are computed by first evaluating their sub-expressions and then combining their results. They show that the theoretical evaluation time might be bound by the class $O(n^3 \cdot m^2)$ where n is the size of the document and m the size of the query. The practicability of the approach is limited given its space complexity bounded by $O(n^4 \cdot m^3)$, which is due to the tabulation of the results of evaluating each subexpression considered during the evaluation. In principle, the reason for the space inefficiency is the same as for bottom-up automata, namely their ignorance of the upper context (opposedly to pushdown automata), which should nevertheless be available in a practical implementation. To alleviate this problem, Gottlob *et al.* give a translation of their bottom-up algorithm into a top-down one. It is shown that this has the same time-complexity and argued that it may compute less intermediate results.

The authors identify a fragment of XPath, called *Core XPath* which can be evaluated more efficiently. Core XPath is XPath without arithmetical or data value operations, being thus basically also a subset of Fxgrep . The complexity of the evaluation scheme proposed in [GKP02] is $O(n \cdot m)$. As opposed to Fxgrep where at most two traversals are needed both for unary and binary queries, the number of traversals of the input document is bounded by the number of steps in the pattern. Since addressing implementation strategies for XPath, [GKP02] only considers unary queries.

Numerical Document Queries

Seidl *et al.* [SSM03] introduce *Presburger tree automata*, which are able to check numerical constraints on children of nodes expressible by Presburger formulas. Whether the children of a node fulfill a Presburger constraint is independent of their relative order. Correspondingly, the tree automata checking Presburger constraints can be considered automata on unordered trees. Unary queries are defined similarly to selecting automata by specifying a distinguished set of states of a Presburger tree automaton. It is shown that unary query evaluation has linear time complexity in the size of the input. To capture the expressiveness of the Presburger tree automata, the authors introduce an extension of MSO logic with Presburger predicates on children of nodes (called PMSO). They show that, on unordered trees, PMSO is equivalently expressive with Presburger tree automata.

Furthermore, they consider expressing both numerical and order constraints on children. For this they extend the Presburger tree automata by allowing regular expressions of Presburger constraints in transitions. It is shown that expressiveness of these automata is captured by PMSO on ordered trees and that evaluating unary queries can be performed in polynomial time. As a case of special interest, they further consider expressing either numerical- or order-constraints on children, depending on the label of the father node. It turns out that the corresponding tree automata have the same expressiveness as the corresponding PMSO logic and that the definable unary queries can be evaluated in linear time.

Tree Queries

To the best of our knowledge, none of the querying approaches mentioned so far, with the exception of grammar queries (available in Fxgrep), have been implemented in practical XML querying languages.

A practically available system for XML querying is X^2 proposed by Meuss *et al.* [MSW$^+$05]. The basic capabilities of the query language of X^2 allow it to specify and relate query nodes via child and descendant relations, labels of nodes or tokens occurring in text nodes. Supplementary constraints make it possible to specify immediate vicinity or relative order of siblings and also to mark nodes as leftmost or rightmost children. Formally, the tree queries expressible by X^2 are *conjunctive queries over trees*. The nodes identifiable via these queries are subsumed by Core XPath (as shown by Gottlob *et al.* in [GKS04]) and therefore also by Fxgrep. Nevertheless, in contrast to XPath, X^2 is able to express k-ary queries, since one answer to a X^2 query is a variable assignment mapping each node in the query to a node in the input (hence, k is the number of nodes in the query). Because the number of answer mappings can be exponentially large, a novel data structure, the complete answer aggregate (CAA) is introduced by Meuss and Schulz [MS01]. CAAs represent the answer space in a condensed form and allow its visual exploration, while guaranteeing that all single answers can be reconstructed. The computation of a CAA for a query is in $O(n \cdot log(n) \cdot h \cdot m)$, where n and h are the size and the maximal depth of the input, respectively, and m is the size of the query.

A main concern of X^2 is supporting the specification of queries and navigation within the answer space via a graphical user interface. Among other visual interfaces to XML query languages which have been proposed are XML-GL [CCD$^+$99], BBQ [MP00], Xing [Erw03], or visXcerpt [BBS03].

There are a few more practical XML tools built upon recent research work. Most of them are strictly speaking not query languages but transformation languages, and we therfore defer addressing them until in Section 12, after presenting our transformation tool based on Fxgrep. Anyhow, we can anticipate that most of the available tools use XPath as a query language. Furthermore, a few more tools emerging from research on XML stream processing are mentioned in Section 7.3.

Chapter 6

Grammar Queries with Boolean Connectives

The grammar formalism and its implementation with pushdown automata introduced so far allow the location of matches fulfilling one structural and one contextual condition. In practice however, queries often need to specify several conditions which are independent (at least conceptually) of each other and which must be simultaneously fulfilled by matches. As an example consider the pattern `//sec[subsec]/title` in which the `sec` nodes have to fulfill two structural constraints, as they are required to have both a `subsec` and a `title` child. Similarly, the pattern `((//king//) & (//queen//)) person` locates `person` nodes which simultaneously fulfill two contextual conditions: they have both a `king` and a `queen` ancestor. Sometimes one also needs to specify that a match must not fulfill a certain condition. Both the grammar formalism and its automata-based implementation can be naturally extended in order to express the above.

We start in Section 6.1 by considering the extensions needed in order to conveniently express boolean combinations of structural conditions. Boolean combinations of contextual conditions are addressed in Section 6.2.

6.1 Conjunctions and Negations of Structural Conditions

The grammar queries introduced so far can already implicitly specify more than one structural condition for a single node as shown in the next example.

Example 6.1: Consider specifying that a document with root element `book` contains at least two `sec` *and* one `cont` element (for contents) as with the Fxgrep pattern `/book[sec _ sec][cont]`. This might be achieved via grammar $G = (R, x_{book})$, where R is the following set of productions:

$$
\begin{aligned}
x_{book} &\to book\langle\, \_\, x_{sec}\, \_\, x_{sec}\, \_\, x_{cont}\, \_ \mid \_\, x_{sec}\, \_\, x_{cont}\, \_\, x_{sec}\, \_ \mid \_\, x_{cont}\, \_\, x_{sec}\, \_\, x_{sec}\, \_\,\rangle \\
x_{sec} &\to sec\langle\, \_\,\rangle \\
x_{cont} &\to cont\langle\, \_\,\rangle
\end{aligned}
$$

Observe that the production for the `book` element has to account for all three different relative positions of the `sec` and `cont` elements. The specification rapidly grows complicated when more elaborate content models have to be simultaneously checked.

□

In general we want to require that a given element simultaneously fulfills two content models r_1 and r_2. One way to achieve this is by reducing it to the problem of specifying a regular expression r whose regular language is exactly the intersection of the regular languages of r_1 and r_2. Placing this burden on the user is however neither reasonable nor necessary.

Instead, we allow one to directly specify that an element has to simultaneously satisfy two given content models, by extending the grammar formalism presented in Section 4.1.1. Furthermore, we allow one to specify that an element must not fulfill a given content model.

6.1.1 Extended Forest Grammars

Extended forest grammars have been introduced in [Neu00]. In the following we briefly review them and mention how they can be implemented.

A *boolean content model* e over a set of non-terminals X is a conjunction of possibly negated regular expressions over X, defined by the following grammar (with non-terminals e and cm):

$$\begin{array}{lcll} e & ::= & cm & \mid cm \,\&\, e \\ cm & ::= & r & \mid\ !r, \text{ where } r \in \mathcal{R}_X \end{array}$$

The set of positive content models $pos(e)$ in a boolean content model e are defined in the obvious way as:

$$pos(cm) \quad = \quad \begin{cases} \{r\}, & \text{if } cm \ = r \\ \emptyset \ \ , & \text{if } cm \ = !r \end{cases}$$

$$pos(e) \quad = \quad \begin{cases} pos(cm) & , \quad \text{if } e \ = \ cm \\ pos(cm) \cup pos(e'), & \text{if } e \ = \ cm \,\&\, e' \end{cases}$$

The set of negative content models $neg(e)$ in a boolean content model e is analogously defined in the obvious way.

An *extended forest grammar* (EFG) over Σ is a tuple $G = (\Sigma, X, R, E_0)$ where R is a finite set of *(extended) productions* and E_0 is a set of boolean content models over X called *start expressions*. The rules in R have the form $x \to a\langle e\rangle$ with $x \in X$, $a \in \Sigma$ and e a boolean content model over X. Again, we omit Σ and X from the grammar when these are obvious from the context.

To distinguish between the original forest grammars (and their productions) as defined in Section 4.1.1 and their extended counterparts we refer to the former as *simple forest grammars* (SFG) (and as *simple productions*). Note that an EFG has a set of start expressions rather than a start expression as in the case of an SFG. This is necessary in order to allow the specification of an alternative of boolean content models for the top level. While in the case of SFG an alternative of content models can be specified with a single content model containing the choice operator |, this is not possible in the case of boolean content models.

Example 6.2: The `book` elements as required by Example 6.1 can be specified by the extended grammar $G = (R, \{x_{book}\})$, where R is the following set of productions:

$$\begin{array}{lll} x_{book} & \rightarrow & book\langle \_\, x_{sec} \_\, x_{sec} \_ \;\; \& \;\; \_\, x_{cont} \_ \rangle \\ x_{sec} & \rightarrow & sec\langle \_ \rangle \\ x_{cont} & \rightarrow & cont\langle \_ \rangle \end{array}$$

To additionally require that the `book` has no `index` element we write:

$$\begin{array}{lll} x_{book} & \rightarrow & book\langle \_\, x_{sec} \_\, x_{sec} \_ \;\; \& \_\, x_{cont} \_ \;\; \& \,!\, \_\, x_{index} \_ \rangle \\ x_{sec} & \rightarrow & sec\langle \_ \rangle \\ x_{cont} & \rightarrow & cont\langle \_ \rangle \\ x_{index} & \rightarrow & index\langle \_ \rangle \end{array}$$

□

As in the case of simple rules, a set of extended rules R together with a boolean content model e or with a non-terminal x specifies a set of forests $[\![R]\!]\, e$ or a set of trees $[\![R]\!]\, x$, respectively, as follows:

$$\begin{array}{rcl} f \in [\![R]\!]\, e & \text{iff} & f \in [\![R]\!]\, r \text{ for all } r \in pos(e) \text{ and} \\ & & f \notin [\![R]\!]\, r \text{ for all } r \in neg(e) \\ t_1 \ldots t_n \in [\![R]\!]\, r & \text{iff} & \text{there is } x_1 \ldots x_n \in [\![r]\!], \text{ with } t_i \in [\![R]\!]\, x_i \\ & & \text{for all } i = 1, \ldots, n \\ a\langle f \rangle \in [\![R]\!]\, x & \text{iff} & f \in [\![R]\!]\, e \text{ for some } e \text{ with } x \rightarrow a\langle e \rangle \in R \end{array}$$

The *language of an extended forest grammar* $G = (R, E_0)$, denoted $\mathcal{L}_G$, is defined as $\mathcal{L}_G = \bigcup_{e \in E_0} [\![R]\!]\, e$.

Recognizing Languages of Extended Forest Grammars

The language of an EFG can be recognized by a pushdown forest automaton constructed in a similar way to the pushdown automaton recognizing the language of an SFG (presented in Section 4.2.2 on page 37). The construction of the recognizing pushdown forest automaton has been presented and its correctness has been proven in [Neu00]. We briefly summarize the construction below.

Basically, when computing the transitions of the automata, for a production containing a boolean content model rather then a simple content model, the fulfillment of all content models in the boolean content model have to be simultaneously considered. Obviously, a boolean content model as defined above is fulfilled when all its positive content models are fulfilled and none of its negative content models is fulfilled.

Given an EFG G we construct a pushdown automaton $A_G^{\vec{}}$ accepting exactly $\mathcal{L}_G$. As opposed to the automaton construction for a SFG, the tree states of $A_G^{\vec{}}$ contain production numbers rather than non-terminals. This is not necessarily needed for accepting $\mathcal{L}_G$, but provides information which will be needed to answer EFG queries, introduced in the next section.

Let $G = (\Sigma, X, R, E_0)$ with $R = \{R_1, \ldots, R_h\}$, such that each production R_k has the form $x_k \rightarrow a_k\langle e_k \rangle$ for $k = 1, \ldots, h$. Furthermore, let $H = \{1, \ldots, h\}$ and let $\{r_1, \dots, r_l\}$ be the set of occurrences of regular expressions in G. For each $j = 1, \dots, l$, let $(Y_j, y_{0,j}, F_j, \delta_j) = \mathit{Berry}(r_j)$ such that $Y_i \cap Y_j = \emptyset$ for $i \neq j$, and let $Y = Y_1 \cup \dots \cup Y_l$ and $\delta = \delta_1 \cup \dots \cup \delta_l$. The automaton $A_{\vec{G}}$ is defined as $(2^H, 2^Y, q_0, F, \mathit{Down}, \mathit{Up}, \mathit{Side})$ where:

$$\begin{aligned}
q_0 &= \{y_{0,j} \mid r_j \in \mathit{pos}(e) \cup \mathit{neg}(e) \text{ for some } e \in E_0\} \\
F &= \{q \mid \text{there is an } e_0 \in E_0 \text{ such that} \\
&\qquad q \cap F_j \neq \emptyset \text{ for all } j \text{ with } r_j \in \mathit{pos}(e_0) \text{ and} \\
&\qquad q \cap F_j = \emptyset \text{ for all } j \text{ with } r_j \in \mathit{neg}(e_0)\} \\
\mathit{Down}(a, q) &= \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle e\rangle, r_j \in \mathit{pos}(e) \cup \mathit{neg}(e)\} \\
\mathit{Up}(a, q) &= \{k \in H \mid a = a_k,\ q \cap F_j \neq \emptyset \text{ for all } j \text{ with } r_j \in \mathit{pos}(e_k), \\
&\qquad \text{and } q \cap F_j = \emptyset \text{ for all } j \text{ with } r_j \in \mathit{neg}(e_k)\} \\
\mathit{Side}(q, p) &= \{y_1 \mid y \in q,\ k \in p, \text{ and } (y, x_k, y_1) \in \delta\}
\end{aligned}$$

6.1.2 Derivations of Extended Forest Grammars

As in the case of simple grammar queries, extended grammar queries will be defined by using the concept of *derivations*, since this allows an easy generalization from the unary to the k-ary case. In this section we introduce the concept of derivations adapted in order to account for boolean content models.

Let $G = (\Sigma, X, R, E_0)$ be an EFG. A boolean content model e defines a *derivation relation* $\mathcal{D}\mathit{eriv}_e \subseteq \mathcal{F}_\Sigma \times (\mathcal{F}_{X \cup \{\&\}} \cup \{\circ\})$ (where "&" and "∘" are two new, reserved symbols) as follows:

$$\begin{aligned}
(f, \&\langle f_1\rangle \dots \&\langle f_k\rangle) \in \mathcal{D}\mathit{eriv}_e \text{ iff } & f \in [\![R]\!]\, e,\ \mathit{pos}(e) = \{r_1, \dots, r_k\} \text{ and} \\
& (f, f_i) \in \mathcal{D}\mathit{eriv}^1_{r_i} \text{ for all } i = 1, \dots, k \\
(f, \circ) \in \mathcal{D}\mathit{eriv}_e \qquad \text{ iff } & f \in [\![R]\!]\, e \text{ and } \mathit{pos}(e) = \emptyset
\end{aligned}$$

$$\begin{aligned}
&(a_1\langle f_1\rangle \dots a_n\langle f_n\rangle, x_1\langle f_1'\rangle \dots x_n\langle f_n'\rangle) \in \mathcal{D}\mathit{eriv}^1_r \text{ iff} \\
&\qquad x_1 \dots x_n \in [\![r]\!],\ x_i \to a_i\langle e_i\rangle \in R \text{ and} \\
&\qquad (f_i, f_i') \in \mathcal{D}\mathit{eriv}_{e_i} \text{ for all } i = 1, \dots, n \\
&(\varepsilon, \varepsilon) \in \mathcal{D}\mathit{eriv}^1_r \text{ iff } \lambda \in [\![r]\!]
\end{aligned}$$

As in the simple forest grammar case, $f \in [\![R]\!]\, e$ denotes that an input forest f conforms to a schema e. Given $(f, f') \in \mathcal{D}\mathit{eriv}_e$, the *derivation* f' might be seen as a proof of how the required *positive* content models are fulfilled by the sequences of sibling nodes in f. The purpose of the "&" symbol in f' is to provide the proof of the conformance of the corresponding sequence of siblings in f to each of the required positive content models in a boolean content model. The purpose of the "∘" symbol is to denote that the corresponding sequence of siblings in f was not required to fulfill any positive content model (implying that nothing below it is required to fulfill any positive content model; hence, "∘" symbols are leaves in derivations).

If $(f, f') \in \mathcal{D}\mathit{eriv}_e$ we say that f' is a *derivation* of f w.r.t. e. We say that f' is a *derivation* of f w.r.t. G iff $(f, f') \in \mathcal{D}\mathit{eriv}_e$ for some $e \in E_0$.

Example 6.3: Let $G = (R, \{x_\top^*(x_1|x_a)x_\top^*\})$ with R containing the following productions:

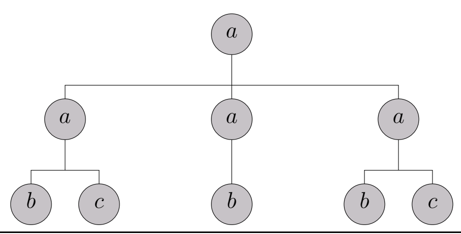


Figure 6.1: Input tree f

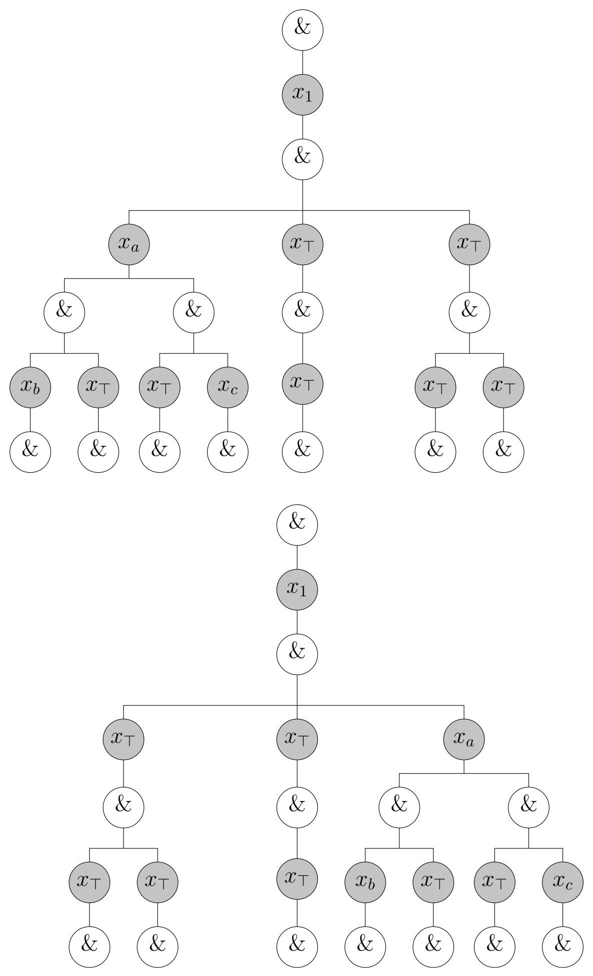


Figure 6.2: Possible derivations of f in Figure 6.1 w.r.t. G in Example 6.3

(1) $x_\top \to a\langle x_\top^* \rangle$	(4) $x_1 \to a\langle x_\top^*(x_1 \vert x_a) x_\top^* \rangle$	(6) $x_b \to b\langle x_\top^* \rangle$
(2) $x_\top \to b\langle x_\top^* \rangle$	(5) $x_a \to a\langle x_\top^* x_b x_\top^* \ \&\ x_\top^* x_c x_\top^* \rangle$	(7) $x_c \to c\langle x_\top^* \rangle$
(3) $x_\top \to c\langle x_\top^* \rangle$		

The language of G consists of all forests in which there is a node a having both a b and a c child and only a ancestors.

Consider as input f the tree depicted in Figure 6.1. Figure 6.2 depicts two possible derivations of f w.r.t. G. □

Every node in a derivation labeled with a non-terminal x has one or more children labeled "&", one for every positive content model of a boolean content model for x. Derivations defined as above have certain similarities with runs of *alternating tree automata* [CDG+05]. In alternating tree automata, a node in a run (corresponding to a derivation in our case) may have an arbitrary number of successor states (corresponding to non-terminals in our case) for the children of the node, s.t. a (positive) propositional formula over states at the children is fulfilled. Extended forest grammars containing only positive content models, can be seen as a restriction in which the formula is a conjunction of conjunctions of states, one state for each child. The correspondent of the formula is in our case a forest of & nodes, each of them having as children a conjunction of non-terminals. In this respect, extended forest grammars could be seen as *alternating forest automata*.

A node in some input f may correspond to many nodes in a derivation f'. Let $\pi \in N(f)$ be a node in f. The set of nodes in f' corresponding to π in f, denoted $\Pi_{\pi}^{f'}$, can be defined as:

$$\Pi_{n_1 \ldots n_k}^{f'} = \{\pi' \in N(f') \mid \pi' = n_1' n_1 n_2' n_2 \ldots n_k' n_k\}$$

That is, to get to a node π' corresponding to π, one arbitrarily descends to one of the "&" siblings at the first and every second step, and consecutively takes the child numbers denoted in π at the other steps.

6.1.3 Unary Extended Grammar Queries

Unary grammar queries based on extended grammar have been introduced in [Neu00]. We present here an equivalent definition based on derivations. The advantage of this definition is that it can be easily extended to capture k-ary queries.

An *extended unary query* Q is specified as in the case of simple unary queries as a distinguished set of non-terminals of an (extended) forest grammar, that is $Q = (G, T)$ with $G = (\Sigma, X, R, E_0)$ being some EFG and $T \subseteq X$. The *matches* of Q in an input forest f are given by the set $\mathcal{M}_{Q,f} \subseteq N(f)$ defined as follows:

$$\begin{array}{ll} \pi \in \mathcal{M}_{Q,f} \text{ iff} & \exists (f, f') \in \mathcal{D}eriv_e \text{ for some } e \in E_0 \text{ and} \\ & \exists \pi' \in \Pi_{\pi}^{f'} \text{ with } lab(f'[\pi']) = x \text{ for some } x \in T \end{array}$$

Example 6.4: Consider the query $Q = (G, \{x_a\})$ with G defined in Example 6.3. This query locates the a elements at arbitrary depths in the input which have only a ancestors and which contain both a b and a c child. Consider as input f the tree depicted in Figure 6.1. The leftmost and rightmost a nodes in the input are matches of Q as one can see in the derivations depicted in Figure 6.2. □

The definition of queries in the presence of conjunctions of positive content models should be quite intuitive. The definition of matches in the presence of negated content models requires some more attention. The intuitive meaning should become clear by reviewing the following few examples.

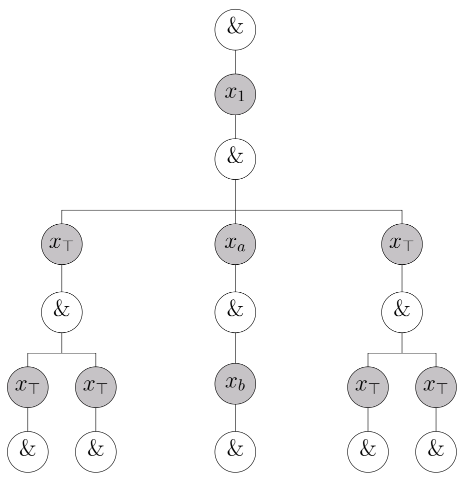


Figure 6.3: Derivation of f in Figure 6.1 w.r.t. G in Example 6.5

Queries with Negated Content Models

As already mentioned, derivations reflect only required positive content models, as shown in the following example:

Example 6.5:
Let $G = (R, \{x_\top^*(x_1|x_a)x_\top^*\})$ with R containing the following productions:

(1) $x_\top \to a\langle x_\top^*\rangle$	(4) $x_1 \to a\langle x_\top^*(x_1|x_a)x_\top^*\rangle$	(6) $x_b \to b\langle x_\top^*\rangle$
(2) $x_\top \to b\langle x_\top^*\rangle$	(5) $x_a \to a\langle x_\top^* x_b x_\top^* \ \&\ !x_\top^* x_c x_\top^*\rangle$	(7) $x_c \to c\langle x_\top^*\rangle$
(3) $x_\top \to c\langle x_\top^*\rangle$		

The query $(G, \{x_a\})$ locates a elements at arbitrary depths in the input which have only a ancestors and which contain a b but no c child. The only derivation of the input tree depicted in Figure 6.1 is shown in Figure 6.3. □

If a node is exclusively required not to fulfill some content models, then the corresponding node in a derivation has a unique child labeled with the "∘" symbol:

Example 6.6: Let $G = (R, \{x_1|x_a\})$ with R containing the following productions:

$$\begin{aligned} x_1 &\to *\langle x_1|x_a\rangle \\ x_a &\to a\langle !x_b\rangle \\ x_b &\to b\langle \_ \rangle \end{aligned}$$

The query $(G, \{x_a\})$ locates (tree representations of) substrings of a (tree-represented) input string which start with an a which is not immediately followed by a b. An input and two derivations are depicted in Figure 6.4.

□

In order for a node to be a match, the definition requires that all its ancestors must be defined positively, as the following example shows.

Example 6.7: Consider $G = (R, \{x_a\})$ with R containing the following productions:

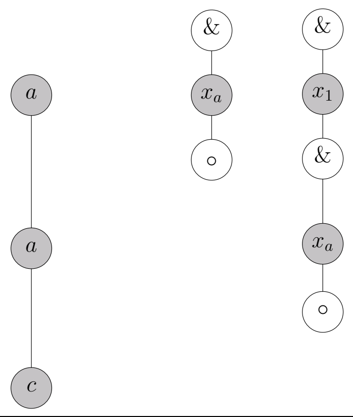


Figure 6.4: Input and derivations for Example 6.6

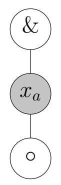


Figure 6.5: Derivation according to G in Example 6.7

$$\begin{aligned} x_a &\to a\langle !\_x_b\_\rangle \\ x_b &\to b\langle !\_x_c\_\rangle \\ x_c &\to c\langle \_\rangle \end{aligned}$$

A tree is in the language $\mathcal{L}_G$ if its root is labeled a and it has no child labeled b which does not contain a child labeled c. It is not quite obvious what are the matches of the query $Q = (G, \{x_b\})$. These should be those b children of a nodes which do not contain a c. But these b nodes are exactly those whose inexistence we require with G. Thus we can never locate such b nodes. Indeed, since x_b only occurs in a negated content model, Q does not ever locate any node even in inputs conforming to G. This should become clear after noting that the only possible derivation of any input is depicted in Figure 6.5 which does not contain any node labeled x_b. Similar considerations can be made for the query $Q = (G, \{x_c\})$. □

6.1.4 K-ary Extended Grammar Queries

As in the case of simple grammar queries, going from the unary to the k-ary case is straightforward by using the concept of derivations. An *extended k-ary query* is a pair $Q = (G, T)$ where $G = (\Sigma, X, R, E_0)$ is an EFG and $T \subseteq X^k$. The *matches* of Q in an input forest f are given by the k-ary relation $\mathcal{M}_{Q,f} \subseteq N(f)^k$:

$$(\pi_1, \ldots, \pi_k) \in \mathcal{M}_{Q,f} \text{ iff } \begin{array}{l} \exists (f, f') \in \mathcal{D}eriv_e \text{ for some } e \in E_0 \text{ and} \\ \exists (x_1, \ldots, x_k) \in T \text{ and} \\ \exists \pi'_i \in \Pi^{f'}_{\pi_i} \text{ with } lab(f'[\pi'_i]) = x_i \text{ for } i = 1, \ldots, k \end{array}$$

Example 6.8: Consider grammar G from Example 6.3 (on page 70) and $Q = (G, \{(x_b, x_c)\})$. The query locates a b and a c sibling which have only a ancestors regardless of their relative order. In the input f depicted Figure 6.1 these are the left-most and right-most b and c, respectively. This can be seen in the derivations of f w.r.t. G depicted in Figure 6.2 (on page 71).

□

6.1.5 Locating Unary Matches

As in the case of unary queries specified by SFGs, the construction for answering unary queries specified via EFGs requires annotating the input by the DLPA recognizing $\mathcal{L}_G$, $A_{\vec{G}}$, followed by the running of a DRPA $B_{\overleftarrow{G}}$ which uses the annotation to recognize matches.

Let f be an input forest and $\vec{f}$ be the $A_{\vec{G}}$-annotation of f (defined as in Section 5.2.2 on page 45). Furthermore, let $q_F = \delta_{\mathcal{F}}(q_0, f)$ be the output state of $A_{\vec{G}}$ for f. The DRPA $B_{\overleftarrow{G}}$ runs over $\vec{f}$, i.e. over $\Sigma \times P \times Q$, where P and Q are the set of tree and forest states of $A_{\vec{G}}$, and is defined as $(2^X, 2^Y, \{\overleftarrow{q}_0\}, \emptyset, Down^{\leftarrow}, Up^{\leftarrow}, Side^{\leftarrow})$, where:

$$\begin{aligned}
\overleftarrow{q}_0 &= \{y \mid y \in F_j \text{ for some } e \in E_0 \text{ with } r_j \in pos(e), \\
&\qquad F_i \cap q_F \neq \emptyset \text{ for all } i \text{ with } r_i \in pos(e) \text{ and} \\
&\qquad F_i \cap q_F = \emptyset \text{ for all } i \text{ with } r_i \in neg(e)\} \\
Down^{\leftarrow}(q, (a, \vec{p}, \vec{q})) &= \{y_2 \mid y \in q \cap \vec{q},\ k \in \vec{p},\ (y_1, x, y) \in \delta \text{ for some } y_1, \\
&\qquad R_k \equiv x \longrightarrow a\langle e\rangle \text{ and } y_2 \in F_j \text{ for some } r_j \in pos(e)\} \\
Up^{\leftarrow}(q, (a, \vec{p}, \vec{q})) &= \{x_k \mid k \in \vec{p}\} \\
Side^{\leftarrow}(q, p, (a, \vec{p}, \vec{q})) &= \{y_1 \mid (y_1, x, y) \in \delta, y \in q \cap \vec{q}, x \in p\}
\end{aligned}$$

where we also provide the $Side^{\leftarrow}$ transition with the label $(a, \vec{p}, \vec{q})$ of the node over which it is executed.

Note how the information in tree states $\vec{p}$ is used in $Down^{\leftarrow}$ transitions. When proceeding to the children forest, we want to consider all positive content models occurring in boolean content models, which are fulfilled by the children forest. Which boolean content models e were fulfilled is told by the production numbers k in $\vec{p}$.

Using the $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$ pushdown forest automata constructed as above, matches can be located as stated in the following theorem. We use the same notations as in the simple case: for a node πi, $\vec{q}_{\pi(i-1)}$ is the forest state in which $A_{\vec{G}}$ reaches the node, $\vec{p}_{\pi i}$ the tree state synthesized for the node, and $\vec{q}_{\pi i}$ the forest state in which $A_{\vec{G}}$ leaves the node; $q_{\pi i}$ and $q_{\pi(i-1)}$ are the states in which $B_{\overleftarrow{G}}$ reaches and leaves the node, respectively .

Theorem 6.1: Let $Q = (G, T)$ be a unary extended grammar query and $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$ be as above. Then

$$\pi \in \mathcal{M}_{Q,f} \text{ iff } y \in q_\pi \cap \vec{q}_\pi \text{ and } (y_1, x, y) \in \delta \text{ for some } y, y_1 \in Y \text{ and } x \in T.$$

This theorem is proven in [Neu00] as Theorem 7.3. □

6.1.6 Locating Binary Matches

Let $Q = (G, \{(x^1, x^2)\})$ be a binary query where G is an EFG and let f be some input forest. The construction for locating binary matches of EFG queries is similar to that for SFG queries presented in Section 5.3.1, but uses the different transitions of the pushdown forest automata $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$. The main difference is the way in which binary matches are recognized, which is presented below. Also, the generalization to the case in which $Q = (G, T)$ with $T \subseteq X$ is straightforward, as in the SFG case.

The $B_{\overleftarrow{G}}$ automaton works over the annotation $\vec{f}$ of the input performed by $A_{\vec{G}}$ as presented in Section 6.1.3. As in the case of simple binary queries, we call unary matches of $(G, \{x^1\})$ and $(G, \{x^2\})$, *primary* and *secondary* matches, respectively. Primary and secondary matches encountered during the run of $B_{\overleftarrow{G}}$, are accumulated in set attributes l_1 and l_2 with which we equip each element of the tree and forest states of $B_{\overleftarrow{G}}$.

The values of the attributes l_1 and l_2 are computed by a set of local rules (similarly to the rules in Section 5.3.1.1 on page 48):

✧ If $y \in F_0$ or $y \in Down^{\leftarrow}(q, (a, \vec{p}, \vec{q}))$ then $y.l_1 = \emptyset$, $y.l_2 = \emptyset$.

✧ If $x \in Up^{\leftarrow}(q, (a, \vec{p}, \vec{q}))$ at node π then

$$x.l_1 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \to a\langle e\rangle, r_j \in pos(e)\}, & \text{if } x = x^1 \\ \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \to a\langle e\rangle, r_j \in pos(e)\} & \text{, otherwise} \end{cases}$$

$$x.l_2 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \to a\langle e\rangle, r_j \in pos(e)\}, & \text{if } x = x^2 \\ \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \to a\langle e\rangle, r_j \in pos(e)\} & \text{, otherwise} \end{cases}$$

✧ If $y \in Side^{\leftarrow}(q, p, (a, \vec{p}, \vec{q}))$ then

$$y.l_1 = \bigcup\{y_1.l_1 \cup x.l_1 \mid (y, x, y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\}$$
$$y.l_2 = \bigcup\{y_1.l_2 \cup x.l_2 \mid (y, x, y_1) \in \delta, y_1 \in q \cap \vec{q}, x \in p\}$$

To find matches of binary EFG queries we proceed similarly to the simple case. Matches are detected, on the one hand, at $Side^{\leftarrow}$ transitions completely analogously to the simple case. Let $x \in p_{\pi i}$, $y \in \vec{q}_{\pi i} \cap q_{\pi i}$. Furthermore, let $(y', x, y) \in \delta$, $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$. Then:

(a) Every pair (π_1, π_2) with $\pi_1 \in x.l_1$, $\pi_2 \in y.l_2$ is a binary match;

(b) Every pair (π_1, π_2) with $\pi_1 \in y.l_1$, $\pi_2 \in x.l_2$ is a binary match;

(c) If $x = x^1 = x^2$ then $(\pi i, \pi i)$ is a binary match.

(d) If $x = x^1$ every pair $(\pi i, \pi_2)$ with $\pi_2 \in x.l_2$ is a binary match.

(e) If $x = x^2$ every pair $(\pi_1, \pi i)$ with $\pi_1 \in x.l_1$ is a binary match.

In the extended case, on the other hand, one additionally has to account for match pairs in which one element of the match pair is defined conforming to a content model and the other with a different content model which has to be simultaneously fulfilled (as specified via a conjunction). These matches are detected when $B_{\overleftarrow{G}}$ finishes visiting a sequence of siblings. Let $q_{\pi 0}$ be the forest state in which $B_{\overleftarrow{G}}$ leaves the forest located at path π. This forest is either the sequence of children of the node π, if $\pi \neq \lambda$, or the top level sequence, if $\pi = \lambda$.

If $\pi \neq \lambda$, let p_π be the tree state synthesized for node π by $A_{\vec{G}}$. For all $y_{0,i}, y_{0,j} \in q_{\pi 0}$, $i \neq j$ and $k \in p_\pi$ with $R_k \equiv x \to a\langle e\rangle, r_i, r_j \in pos(e)$:

(f) Every pair (π_1, π_2) with $\pi_1 \in y_{0,i}.l_1$, $\pi_2 \in y_{0,j}.l_2$ is a binary match.

If $\pi = \lambda$, then (f) holds for all $y_{0,i}, y_{0,j} \in q_{\pi 0}$, $i \neq j$ and $r_i, r_j \in pos(e)$, for some $e \in E_0$.

The above is summarized by the following theorem:

Theorem 6.2: A pair (π_1, π_2) is a binary match iff
(A) there is $\pi \in N(f)$, $x \in p_\pi$, $y \in q_\pi \cap \vec{q}_\pi$, $(y', x, y) \in \delta$ and either:

(a) $\pi_1 \in x.l_1$, $\pi_2 \in y.l_2$ or

(b) $\pi_1 \in y.l_1$, $\pi_2 \in x.l_2$ or

(c) $\pi_1 = \pi_2 = \pi$, $x = x^1 = x^2$ or

(d) $\pi_1 = \pi$, $x = x^1$, $\pi_2 \in x.l_2$ or

(e) $\pi_2 = \pi$, $x = x^2$, $\pi_1 \in x.l_1$.

or
(B) there are $y_{0,i}, y_{0,j} \in q_{\pi 0}$ with $i \neq j$, $r_i, r_j \in pos(e)$, where $e \in E_0$ or $x \rightarrow a\langle e \rangle \in R$, and:

(f) $\pi_1 \in y_{0,i}.l_1$, $\pi_2 \in y_{0,j}.l_2$

We only sketch the proof here. The more detailed proof is given in Appendix A.3.

The construction ensures two invariants (similar to the case of binary queries defined via SFGs), namely:

(i_1) A node π_1 is propagated in the attribute $x.l_1$ (or $x.l_2$) in the tree state synthesized at a node π exactly when there is an (extended) derivation f' w.r.t. G with a node corresponding to π labeled with x and having a descendant corresponding to π_1 labeled with x^1 (or x^2 respectively).

(i_2) A node π_2 is propagated in the attribute $y.l_2$ (or $y.l_1$) in the state in which a node π is reached exactly when there is an (extended) derivation f'' w.r.t. G with a node π' corresponding to π in it labeled with x, the incoming transition into y, and a descendant of a right sibling of π' labeled with x^2 (or x^1 respectively).

Soundness Let a primary match π_1 and a secondary match π_2 meet as in one of the cases (a) to (e). Then, there exist two derivations f' and f'' as in (i_1) and (i_2). A derivation which defines (π_1, π_2) as a binary match can be built by replacing the content of the node corresponding to the meeting point π' from f'' with that from f'.

Let a primary match π_1 and a secondary match π_2 meet as in case (f). Then, by considering (i_1), (i_2) and the last executed $Side^{\leftharpoonup}$ transition it follows that there exist two derivations f' and f'' defining π_1 as a primary match and π_2 as a secondary match in which the content model of the current element conforms to a content model r_i and r_j, respectively. A derivation which defines (π_1, π_2) as a binary match can be built by replacing in f' the content corresponding to r_j with that from f''.

Completeness Let (π_1, π_2) be a a binary match. By definition, there is a derivation f''' w.r.t. G in which two nodes π_1' and π_2' corresponding to π_1 and π_2 are labeled with x^1 and x^2, respectively. By the definition of extended derivations, the nodes π_1' and π_2' necessarily have a nearest common ancestor in f''' which is labeled either with some non-terminal x or with the symbol "&".

Let us consider first that the nearest common ancestor is a node labeled "&". If (π_1, π_2) is a match then (i_1) and (i_2) ensure that this is detected as stated in one of the cases (a) to (e), depending on the relative position of π_1 and π_2 (as in Figure 5.4 and Figure 5.5 on page 50).

If the nearest common ancestor of π_1' and π_2' is a node labeled with some non-terminal x then π_1' and π_2' have to be located within two different &-labeled children node of the common ancestor. If (π_1, π_2) is a match then (i_1) and (i_2) ensure that this is detected as in case (f).

□

6.2 Conjunctions and Negations of Contextual Conditions

The extension presented in the previous paragraph allows one to specify that a node of interest simultaneously satisfies several content models or that it must not satisfy a content model. Besides that, we sometimes need to specify that a node of interest simultaneously occurs in several different contexts or that it must not occur in a specific context.

Example 6.9: As a motivating example consider locating in some XML document storing a genealogical tree, `person` elements having an ancestor labeled `king` and no ancestor labeled `duke`, expressible in Fxgrep via the pattern `((//king//)& !(//duke//)) person`. This cannot be expressed with the querying formalism introduced so far. The extension of the formalism allowing the formulation of this query is presented below. □

In order to deal with negations in context specifications conveniently, we extend the notion of queries in the following way: a query is conceptually now defined by one or more context queries (with vacuous structural constraints) and one or more structure queries (with vacuous contextual constraints). A context query locates nodes of arbitrary structure which find themselves in the specified context, while a structure query locates trees with the specified structure. Thus, now a node is a match if it is simultaneously located by all its structure and the context queries. Context or structure constraints might also occur negated, meaning that they must not be fulfilled by the match nodes.

Example 6.10: By allowing arbitrary boolean combinations of both contextual and structural constraints, the matches as in Example 6.9 can be specified by $Q_1 \wedge Q_2 \wedge \neg Q_3$ where $Q_1 = ((R_{person}, x_0), \{x_{person}\})$ (the structure query), $Q_2 = ((R_{king}, x_1), \{x_2\})$ and $Q_3 = ((R_{duke}, x_3), \{x_4\})$ (the context queries) with the productions:

R_{person} : (x_0-root, x_{person}-arbitrarily deep situated element labeled *person*)

$$\begin{array}{lll} x_0 & \rightarrow & *\langle\_x_0 \mid x_{person}\_\rangle \\ x_{person} & \rightarrow & person\langle\_\rangle \end{array}$$

R_{king} : (x_1-root, x_2-an arbitrary element having an ancestor labeled *king*)

$$\begin{array}{lll} x_1 & \rightarrow & *\langle\_x_1 \mid x_{king}\_\rangle \\ x_{king} & \rightarrow & king\langle\_x_2\_\rangle \\ x_2 & \rightarrow & *\langle\_x_2\_\rangle \end{array}$$

R_{duke} : (x_3-root, x_4-an arbitrary element having an ancestor labeled *duke*)

$$\begin{array}{lll} x_3 & \rightarrow & *\langle\_x_3 \mid x_{duke}\_\rangle \\ x_{duke} & \rightarrow & duke\langle\_x_4\_\rangle \\ x_4 & \rightarrow & *\langle\_x_4\_\rangle \end{array}$$

□

6.2.1 Unary Formula Queries

In fact, it is not necessary to have different grammars for different portions of the composite query as suggested above. Instead, we find it convenient to use just one grammar but allow boolean combinations of target non-terminals.

The set $Form_X$ of boolean formulas h over a set of non-terminals X is defined as:

$$\begin{array}{ll} h \in Form_X \text{ iff} & h = x \text{ and } x \in X, \text{ or} \\ & h = h_1 \vee h_2 \text{ and } h_1, h_2 \in Form_X, \text{ or} \\ & h = h_1 \wedge h_2 \text{ and } h_1, h_2 \in Form_X, \text{ or} \\ & h = \neg h_1 \text{ and } h_1 \in Form_X \end{array}$$

Let $\mathcal{B}$ be the set of boolean values $\mathcal{B} = \{true, false\}$. Given a variable environment $\sigma : X \rightarrow \mathcal{B}$, the meaning $[\![h]\!] : \sigma \rightarrow \mathcal{B}$ of a formula h is defined in the obvious way, inductively on the structure of formulas:

$$\begin{array}{lll} [\![x]\!]\,\sigma & = & \sigma\, x \\ [\![h_1 \vee h_2]\!]\,\sigma & = & [\![h_1]\!]\,\sigma \text{ or } [\![h_2]\!]\,\sigma \\ [\![h_1 \wedge h_2]\!]\,\sigma & = & [\![h_1]\!]\,\sigma \text{ and } [\![h_2]\!]\,\sigma \\ [\![\neg h_1]\!]\,\sigma & = & \text{not } [\![h_1]\!]\,\sigma \end{array}$$

A *unary formula query* is a tuple (G, h) where G is a (simple or extended) forest grammar with a set of non-terminals X and $h \in Form_X$. The matches of a unary formula query (G, h) in an input forest f are given as the set $\mathcal{M}_h \subseteq N(f)$ recursively defined by:

$$\begin{array}{ll} \pi \in \mathcal{M}_x & \text{iff } \pi \in \mathcal{M}_{Q,f} \\ \pi \in \mathcal{M}_{h_1 \vee h_2} & \text{iff } \pi \in \mathcal{M}_{h_1} \text{ or } \pi \in \mathcal{M}_{h_2} \\ \pi \in \mathcal{M}_{h_1 \wedge h_2} & \text{iff } \pi \in \mathcal{M}_{h_1} \text{ and } \pi \in \mathcal{M}_{h_2} \\ \pi \in \mathcal{M}_{\neg h} & \text{iff } \pi \notin \mathcal{M}_h \end{array}$$

where $Q = (G, \{x\})$ is a unary (SFG or EFG) query.

Example 6.11: Given the set of productions $R = R_{person} \cup R_{king} \cup R_{duke}$ from Example 6.10, the query in Example 6.10 can be expressed as $((R, x_0 \mid x_1 \mid x_3), x_{person} \wedge x_2 \wedge \neg x_4))$. □

Relation to Unary Extended Grammar Queries

Unary SFG formula queries can be used to express many unary EFG queries (introduced in Section 6.1.3). This is always possible when the conjunctions and negations of structural conditions to be expressed concern only the match node.

Example 6.12: Locating `book` elements which have at least two `sec` elements, one `cont` element and no `index` element can be achieved by $Q = (G, \{x_{book}\})$, with $G = (R, x_{book})$ being the EFG in Example 6.2 the productions of which are reproduced for convenience below:

$$\begin{array}{ll} x_{book} & \rightarrow book\langle \_x_{sec}\_x_{sec}\_ \ \&\_x_{cont}\_ \ \& \, !\_x_{index}\_\rangle \\ x_{sec} & \rightarrow sec\langle \_\rangle \\ x_{cont} & \rightarrow cont\langle \_\rangle \\ x_{index} & \rightarrow index\langle \_\rangle \end{array}$$

The same can be expressed with the formula query $((R, x_{book_1}|x_{book_2}|x_{book_3}), x_{book_1} \wedge x_{book_2} \wedge \neg x_{book_3}))$ where R is the following set of productions:

$$\begin{array}{ll} x_{book_1} & \rightarrow book\langle \_x_{sec}\_x_{sec}\_\rangle \\ x_{book_2} & \rightarrow book\langle \_x_{cont}\_\rangle \\ x_{book_3} & \rightarrow book\langle \_x_{index}\_\rangle \\ x_{sec} & \rightarrow sec\langle \_\rangle \\ x_{cont} & \rightarrow cont\langle \_\rangle \\ x_{index} & \rightarrow index\langle \_\rangle \end{array}$$

□

In general, however, SFG formula queries do not capture all EFG queries. This is the case when one needs to express a conjunction of structural conditions on nodes other than the match node. For instance, one might have to locate footnotes in sections containing at least four subsections. The corresponding production for these sections has to specify that their content include a footnote and the four subsections, via a boolean content model.

Therefore, to allow as general queries as possible, formula queries can be defined using both using simple and extended forest grammars. Evaluating these queries is uniformally performed in both cases as presented next.

Locating Matches

The pushdown automata construction can be extended in a straightforward way in order to implement this generalized form of queries. Rather than reporting as match a node from the input which *corresponds* to one distinguished non-terminal of the query, one reports a match if the boolean formula of the query evaluates to *true* in a variable environment defined by the node. The variable environment binds every non-terminal to *true* or *false* depending on whether the node can be derived from the non-terminal or not, respectively.

Let f be an input forest. We use the notations from Section 5.2. The variable environment σ_π defined for every node $\pi \in N(f)$ by the runs of $A_G^{\leftarrow}$ and $B_G^{\leftarrow}$ (constructed as in Section 5.2 or Section 6.1.5, depending on whether G is a SFG or EFG, respectively) is given by:

$$\sigma_\pi\ x = \begin{cases} \mathit{true}, \text{ if } \exists y_1 \in q_\pi \cap \vec{q}_\pi \text{ and } (y, x, y_1) \in \delta \text{ for some } y, y_1 \in Y \\ \mathit{false}, \text{ otherwise} \end{cases}$$

Theorem 6.3: Let $Q = (G, h)$ be a unary formula query and $f \in \mathcal{L}_G$. With $A_G^{\vec{}}$, $B_G^{\overleftarrow{}}$ and σ_π as above, $\pi \in \mathcal{M}_h$ iff $[\![h]\!]\ \sigma_\pi$.

The proof is straightforward by structural induction on h (i.e. induction on the number of operators $\vee$, $\wedge$ and $\neg$ in h).

Let $h = x$ for some $x \in X$, where X is the set of non-terminals in G. We have that $\pi \in \mathcal{M}_x$ iff $\pi \in \mathcal{M}_{Q,f}$ where $Q = (G, \{x\})$. Depending on whether G is an SFG or EFG, it follows by Theorem 5.1 or Theorem 6.1, respectively, that $\pi \in \mathcal{M}_{Q,f}$ iff $y_1 \in q_\pi \cap \vec{q}_\pi$ and $(y, x, y_1) \in \delta$ for some $y, y_1 \in Y$ which is equivalent to $[\![x]\!]\ \sigma_\pi$.

If $h = h_1 \vee h_2$ then $\pi \in \mathcal{M}_h$ iff $\pi \in \mathcal{M}_{h_1}$ or $\pi \in \mathcal{M}_{h_2}$. This is equivalent by the hypothesis induction to $[\![h_1]\!]\ \sigma_\pi$ or $[\![h_2]\!]\ \sigma_\pi$, i.e. $[\![h]\!]\ \sigma_\pi$.

The case $h = h_1 \wedge h_2$ is completely similar.

If $h = \neg h_1$ then $\pi \in \mathcal{M}_h$ iff not $\pi \in \mathcal{M}_{h_1}$, i.e., by the hypothesis induction iff not $[\![h_1]\!]\ \sigma_\pi$ and by definition iff $[\![h]\!]\ \sigma_\pi$.

□

6.2.2 Binary Formula Queries

In this section we extend the formalism for expressing queries via formulas also to binary queries.

Example 6.13: Binary formula queries are needed to locate for example `person` elements having both an ancestor `king` and an ancestor `duke`, together with their first child element. □

A *binary formula query* is a tuple $Q = (Q_2, (h_1, h_2))$, with a binary query $Q_2 = (G, \{(x^1, x^2)\})$ and $h_1, h_2 \in Form_X$, where X is the set of non-terminals in G. The matches of Q in a forest f are given as the set $\mathcal{M}_{Q,f} \subseteq N(f)$ defined by:

$$(\pi_1, \pi_2) \in \mathcal{M}_{Q,f} \text{ iff } (\pi_1, \pi_2) \in \mathcal{M}_{Q_2,f}, \pi_1 \in \mathcal{M}_{h_1} \text{ and } \pi_2 \in \mathcal{M}_{h_2}$$

That is, a binary formula query is a binary query, where the primary and secondary match are additionally required to conform to a formula over non-terminals.

Example 6.14: The query in Example 6.13 can be expressed as $(Q_2, (x_{person} \wedge x_2 \wedge x_4, x_{first}))$ where $Q_2 = ((R, x_0|x_1|x_3), \{(x_{person}, x_{first})\})$ and R is the following set of productions:

$$\begin{array}{lll}
x_0 & \rightarrow & *\langle \_\, x_0 \mid x_{person}\, \_ \rangle \\
x_{person} & \rightarrow & person\langle x_{first}\, \_ \rangle \\
x_{first} & \rightarrow & *\langle \_ \rangle \\
x_1 & \rightarrow & *\langle \_\, x_1 \mid x_{king}\, \_ \rangle \\
x_{king} & \rightarrow & king\langle \_\, x_2\, \_ \rangle \\
x_2 & \rightarrow & *\langle \_\, x_2\, \_ \rangle \\
x_3 & \rightarrow & *\langle \_\, x_3 \mid x_{duke}\, \_ \rangle \\
x_{duke} & \rightarrow & duke\langle \_\, x_4\, \_ \rangle \\
x_4 & \rightarrow & *\langle \_\, x_4\, \_ \rangle
\end{array}$$

□

Binary Formula Queries with Simple Forest Grammars

For simplicity, we first consider the case in which G is a simple forest grammar. The case in which G is an extended forest grammar is completely analogous and is addressed at the end of this section.

Matches of binary formula queries are located by a construction similar to that for binary queries (as introduced in Section 5.3.1). The $A_{\vec{G}}$ and $B_{\overleftarrow{G}}$ automata are exactly the same, but instead of l_1 and l_2, two other attributes l'_1 and l'_2 are computed. The attributes l'_1 and l'_2 are computed using the same rules as for l_1 and l_2 except for the rule computing the attributes of the x elements in tree states as follows.

If $x \in Up^{\leftarrow}(q, (a, \vec{p}, \vec{q}))$ then:

$$x.l'_1 = \begin{cases} \{\pi\} \cup \bigcup\{y.l'_1 \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\}, & \text{if } x = x^1 \text{ and } [\![h_1]\!]\ \sigma_\pi \\ \bigcup\{y.l'_1 \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\} \quad , & \text{otherwise} \end{cases}$$

$$x.l'_2 = \begin{cases} \{\pi\} \cup \bigcup\{y.l'_2 \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\}, & \text{if } x = x^2 \text{ and } [\![h_2]\!]\ \sigma_\pi \\ \bigcup\{y.l'_2 \mid y \in q, y = y_{0,j}, x \to a\langle r_j\rangle\} \quad , & \text{otherwise} \end{cases}$$

Using this new construction, binary formula matches are located similarly to binary matches, as stated by the following theorem (analogous to Theorem 5.4):

Theorem 6.4: $(\pi_1, \pi_2) \in \mathcal{M}_{Q,f}$ iff there is $\pi i \in N(f)$, $x \in p_{\pi i}$, $y \in q_{\pi i} \cap \vec{q}_{\pi i}$, $(y', x, y) \in \delta$ and either:

(a) $\pi_1 \in x.l'_1$, $\pi_2 \in y.l'_2$ or

(b) $\pi_1 \in y.l'_1$, $\pi_2 \in x.l'_2$ or

(c) $\pi_1 = \pi_2 = \pi i$, $x = x^1 = x^2$, $[\![h_1 \wedge h_2]\!]\ \sigma_{\pi i}$ or

(d) $\pi_1 = \pi i$, $x = x^1$, $[\![h_1]\!]\ \sigma_{\pi i}$, $\pi_2 \in x.l'_2$ or

(e) $\pi_2 = \pi i$, $x = x^2$, $[\![h_2]\!]\ \sigma_{\pi i}$, $\pi_1 \in x.l'_1$.

Let us consider case (a). Obviously, by comparing the construction with that in Section 5.3.1, we have that $\pi_1 \in x.l'_1$ iff $\pi_1 \in x.l_1$ and $[\![h_1]\!]\ \sigma_{\pi_1}$, and $\pi_2 \in y.l'_2$ iff $\pi_2 \in y.l_2$ and $[\![h_2]\!]\ \sigma_{\pi_2}$. By Theorem 5.4 (for binary matches for SFGs, on page 53), $\pi_1 \in x.l_1$ and $\pi_2 \in y.l_2$ is equivalent to $(\pi_1, \pi_2) \in \mathcal{M}_{Q_2,f}$. Also, by Theorem 6.3 (for formula queries, on page 81), $[\![h_1]\!]\ \sigma_{\pi_1}$ and $[\![h_2]\!]\ \sigma_{\pi_2}$ iff $\pi_1 \in \mathcal{M}_{h_1}$ and $\pi_2 \in \mathcal{M}_{h_2}$. Thus, $\pi_1 \in x.l'_1$ and $\pi_2 \in y.l'_2$ iff $(\pi_1, \pi_2) \in \mathcal{M}_{Q_2,f}$, $\pi_1 \in \mathcal{M}_{h_1}$ and $\pi_2 \in \mathcal{M}_{h_2}$, i.e. iff $(\pi_1, \pi_2) \in \mathcal{M}_{Q,f}$.

Case (b) follows completely similarly. In case (c), we have by Theorem 5.4 that $\pi_1 = \pi_2 = \pi i$, $x = x^1 = x^2$ iff $(\pi_1, \pi_2) \in \mathcal{M}_{Q_2,f}$. Also, by definition $[\![h_1 \wedge h_2]\!]\ \sigma_{\pi i}$ iff $[\![h_1]\!]\ \sigma_{\pi i} \wedge [\![h_2]\!]\ \sigma_{\pi i}$. From the last two equivalences it follows that (c) holds.

Cases (d) and (e) follow similarly. □

Binary Formula Queries with Extended Forest Grammars

The case in which a formula query is defined via an extended forest grammar $G = (R, E_0)$, can be handled completely similarly, by extending the construction for locating binary matches for EFG queries presented in Section 6.1.6, rather than that for locating binary matches of SFG queries as before. The $A_G^{\vec{}}$ and $B_G^{\overleftarrow{}}$ automata are exactly the same as in Section 6.1.6, but instead of l_1 and l_2, two other attributes l_1' and l_2' are computed. The attributes l_1' and l_2' are computed using the same rules as for l_1 and l_2 except for the rule computing the attributes of the x elements in tree states as follows.

If $x \in Up^{\leftarrow}(q,(a,\vec{p},\vec{q}))$ at node π then

$$x.l_1 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \to a\langle e\rangle, r_j \in pos(e)\}, & \text{if } x = x^1 \text{ and} \\ & [\![h_1]\!]\ \sigma_\pi \\ \bigcup\{y.l_1 \mid y \in q, y = y_{0,j}, x \to a\langle e\rangle, r_j \in pos(e)\} \qquad, & \text{otherwise} \end{cases}$$

$$x.l_2 = \begin{cases} \{\pi\} \cup \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \to a\langle e\rangle, r_j \in pos(e)\}, & \text{if } x = x^2 \text{ and} \\ & [\![h_2]\!]\ \sigma_\pi \\ \bigcup\{y.l_2 \mid y \in q, y = y_{0,j}, x \to a\langle e\rangle, r_j \in pos(e)\} \qquad, & \text{otherwise} \end{cases}$$

Matches are located as stated by the following theorem (analogous to Theorem 6.2)

Theorem 6.5: A pair (π_1, π_2) is a binary match iff
(A) there is $\pi \in N(f)$, $x \in p_\pi$, $y \in q_\pi \cap \vec{q}_\pi$, $(y', x, y) \in \delta$ and either:

(a) $\pi_1 \in x.l_1$, $\pi_2 \in y.l_2$ or

(b) $\pi_1 \in y.l_1$, $\pi_2 \in x.l_2$ or

(c) $\pi_1 = \pi_2 = \pi$, $x = x^1 = x^2$, $[\![h_1 \wedge h_2]\!]\ \sigma_{\pi i}$ or

(d) $\pi_1 = \pi$, $x = x^1$, $[\![h_1]\!]\ \sigma_{\pi i}$, $\pi_2 \in x.l_2$ or

(e) $\pi_2 = \pi$, $x = x^2$, $[\![h_2]\!]\ \sigma_{\pi i}$, $\pi_1 \in x.l_1$.

or
(B) there is $y_{0,i}, y_{0,j} \in q_{\pi 0}$ with $i \neq j$, $r_i, r_j \in pos(e)$, where $e \in E_0$ or $x \to a\langle e\rangle \in R$, and:

(f) $\pi_1 \in y_{0,i}.l_1$, $\pi_2 \in y_{0,j}.l_2$

The proof is completely analogous to that of Theorem 6.4 while using Theorem 6.2 instead of Theorem 5.4. □

6.3 Bibliographic Notes

Neumann handled unary queries extended with boolean connectives of structural conditions via extended grammar queries in [Neu00]. His definition of unary matches is given, as in the case of simple unary queries, recursively in terms of the queries matched by the father node and the structural constraints fulfilled by the sibling nodes. This local type of definition is not suitable for an extension to k-ary queries. Our definition based on derivations is equivalent to the original definition for the unary case and can be straightforwardly extended to the k-ary case. The algorithm for locating unary matches is introduced in [Neu00]. The algorithm for locating binary matches of EFG queries is a new contribution.

The contribution regarding boolean connectives for contextual conditions is essentially new. Universal unary queries defined by selecting automata (as mentioned in Section 5.6) are the special case of unary formula queries in which the formula is restricted to a conjunction of non-terminals. In contrast, the definition of universal *k*-ary queries in [NPTT05], by which a tuple of nodes is a match if for every tuple of states specified by a query there is a run visiting the nodes in these states, is not comparable with our definition for the binary case. Nevertheless, the result of [NPTT05], by which universal and existential queries expressed by selecting automata are equally expressive, implies that our unary formula queries consisting only of conjunctions of non-terminals are equally expressive as the simple grammar queries.

Chapter 7

Querying XML Streams

XML processing can be classified into two main categories, which correspond to the two main approaches to XML parsing, DOM [DOM98] and SAX [SAX98]. In the first approach, used by most existing XML processors, the tree which is textually represented by the XML input is effectively constructed in memory and subsequently used by the XML application.

In the second approach, the XML input is transformed into a stream of events which are transmitted to a listening application. An event contains a small piece of information linearly read from the input, e.g. a *start-tag* or an *end-tag*. The order of the events in the stream corresponds to the document order of the input, i.e. to the sequential order in which the information is read from the input. It is up to the listening application to decide how it processes the stream of events. In particular, it can construct the XML tree in memory and subsequently process it, as in the first approach, being thus at least as expressive.

The advantage of the event-based approach is that it allows one to buffer only the relevant parts of the input, thereby saving time and memory. The increased flexibility allows the handling of very large documents, the size of which would be prohibitive if the XML input tree was to be entirely built in memory. Also, the event-based processing naturally captures real-life applications in which the document is received linearly via some communication channel, rather than being completely available in advance.

The research interest in querying XML streams has been very vivid recently and there is a very rich literature on this topic. The related work is reviewed in Section 7.3. The proposed query languages are generally able to implement different subsets of XPath . Most of them are subsumed by *Core XPath* .

Our contribution is a novel solution for efficient event-based evaluation of queries which go beyond the capabilities of many languages for which this problem was previously addressed. Most of these languages can be expressed using first-order logic (FO) possibly extended with regular expressions on vertical paths, but are less expressive than monadic second order logic (MSO). In contrast, our solution evaluates *grammar queries*, which are equivalent to MSO queries as mentioned in Section 5.1.3.

Grammar queries can be implemented using pushdown forest automata as presented in Section 5.2. The original construction as introduced by Neumann and Seidl [NS98a] generally requires the construction of the whole input tree in memory and the execution of two traversals of it. A one-pass query eval-

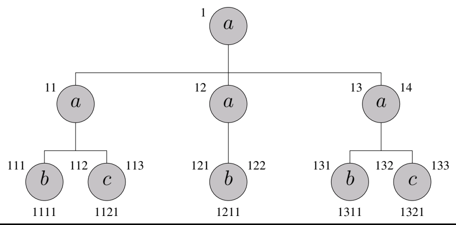


Figure 7.1: Input tree

uation, suitable for an event-based implementation, is addressed only for a very restricted class of queries. These are the so called *right-ignoring* queries for which all the information needed to decide whether a node is a match has been seen by the time the end-tag of the node is encountered. The term right-ignoring is coined by the fact that all the nodes to the right of the match node in the tree representation are irrelevant for the match.

In this chapter we lift this restriction. Rather than a-priori (i.e. statically) handling only a restricted subset of queries, we show here how *arbitrary* grammar queries can be evaluated on XML streams using pushdown forest automata.

Example 7.1: Consider an XML document, the tree representation of which is depicted in Figure 7.1. Each location in the tree corresponds to an event in the corresponding stream of events. The stream of events together with the corresponding locations are denoted below. Nodes too can be identified by the location corresponding to their start-tag.

```
 1
<a>
        11  111 1111  112  1121  113
        <a> <b> </b>  <c>  </c>  </a>
        12  121 1211  122
        <a> <b> </b>  </a>
        13  131 1311  132  1321  133
        <a> <b> </b>  <c>  </c>  </a>
 14
</a>
```

It should be clear that the amount of memory necessary to answer an arbitrary query inherently depends on the query and on the input document at hand. Consider for example the (XPath or Fxgrep) pattern `//a/b` locating `b` nodes which have as father an `a` node. The node 111 is a match in our input. This can be detected as early as at the location 111, as the events following 111 cannot change the fact of 111 being a match.

The pattern `//a[c]/b` locates `b` nodes which have a node `a` as father and a `c` sibling. The node 111 is again a match but this becomes clear only after seeing that the `a` parent has also a child `c` at location 112. One has thus to remember 111 as a potential match between the events 111 and 112. As the events to the right of 112 cannot change the fact of 111 being a match, 111 can be reported and discarded at 112.

Finally, as an extreme case consider the (MSO expressible) XPath pattern `/*[not(d)]//*` locating all descendant nodes of the root element if this has no child node d. Any node in the input is a potential match until seeing the last child of the root element. In our example all nodes have to be remembered as potential matches up to the last event 14. Note thus, that any algorithm evaluating a query needs in the worst case an amount of space linear in the input size. However, most of the practical queries require a quite small amount of memory as compared to the size of the input. □

The contributions of this chapter are as follows.

- ✧ We introduce a way of defining the earliest detection location of a match for some given query and input tree.
- ✧ The main contribution is proving that matches of grammar queries are recognizable at their earliest detection location and hereby proving the following theorem:

  Theorem 7.1: Matches of MSO definable queries are recognizable at their earliest detection location.

- ✧ Based on the construction used for proving Theorem 7.1 we give an efficient algorithm for grammar query evaluation, which reports matches at their earliest detection point. As a consequence potential matches are remembered only as long as necessary, meaning that our construction implicitly adapts its memory consumption to the strict requirements of the query on the input at hand.
- ✧ The algorithm has been completely implemented in Fxgrep . We provide experimental evidence of the practical performance of the algorithm.

This chapter is organized as follows. In section Section 7.1 we briefly present how a pushdown automaton can be used to answer right-ignoring queries on streams. The main contribution on query evaluation for XML streams is given in Section 7.2 where the algorithm is presented and its correctness, optimality, complexity and performance are addressed. Related work is discussed in Section 7.3.

In the following let $Q = (G, T)$ be an arbitrary query on input f_1 with $G = (\Sigma, X, R, r_0)$. Further, let $A_{\vec{G}}$ be the LPA accepting $\mathcal{L}_G$ constructed as in Section 4.2.2.

7.1 Right-ignoring Queries

In this section we briefly present the ideas [NS98a, Neu00] which allow the evaluation of a right-ignoring query $Q = (G, T)$ using the $A_{\vec{G}}$ LPA. Let us investigate what are the requirements for Q to be right-ignoring. Consider a match node π of Q as depicted in Figure 7.2. Since the query is right-ignoring, all the nodes from the right-context are irrelevant for the decision as to whether π is a match. That is, π is a match however the right-siblings of π and of every ancestor of π might look like.

Let us consider that π is the k-th out of m siblings, with $m \geq k$. Since π is a match, according to the definition, there is a derivation labeling the sequence of siblings containing π with $x_1 \ldots x_k \ldots x_m$ and $x_k \in T$. There is thus a content

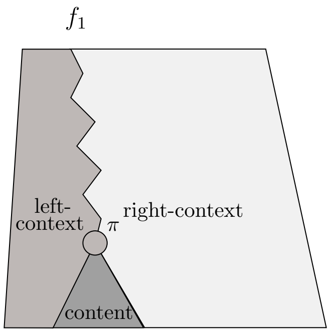


Figure 7.2: The context and the content of a match

model r_j s.t. $x_1 \dots x_k \dots x_m \in [\![r_j]\!]$. The fact that the right siblings of π might be any trees implies that x_i must be able to derive any tree, i.e. $[\![R]\!]\ x_i = \mathcal{T}_\Sigma$ for all $i = k+1, \dots, m$. Also, as the number of right-siblings might be arbitrary, all of the above must hold for all $m \in \mathbb{N}$ with $m \geq k$.

To ensure the above there must exist an NFA state $y_k \in Y_j$ reached after seeing the left siblings of π with $y_k \in F_j$ and s.t. for all $p \in \mathbb{N}$, there are $y_{k+1}, \dots, y_p \in F_j$ with $(y_i, x_\top, y_{i+1}) \in \delta_j$ for $i = k+1, \dots, p$, where $x_\top \in X$ and $[\![R]\!]\ x_\top = \mathcal{T}_\Sigma$. We call such a y_k a *right-ignoring NFA state*. The non-terminal $x_\top$ is to be seen as a wild-card non-terminal which can derive any tree and which is made available in any forest grammar. The necessity of the above follows from the fact that no other non-terminal x in the grammar can be s.t. $[\![R]\!]\ x = \mathcal{T}_\Sigma$, as in general the alphabet Σ is neither finite nor known in advance[1].

Given an NFA state y, we use the predicate $rightIgn(y)$ to test whether y is a right ignoring state. Testing whether $rightIgn(y)$ holds, can be done statically by checking in the NFA whether there are cycles visiting y and consisting only of $x_\top$ edges, needing thus time linear in the size of the NFA.

Similar considerations have to be made due to the right-ignorance for all the nodes lying on the path from the root to π. Therefore we need to consider all the non-terminals with which a derivation defining a match may label the nodes lying on the path from the match to the root. These are the so-called *match-relevant* non-terminals, defined by:

x is match-relevant iff
  $x \in T$ or $x \to a\langle r_j \rangle, (y_1, x_1, y) \in \delta_j$ and x_1 is match-relevant

We call a query Q *right-ignoring* iff all $y \in Y$ with $(y_1, x, y) \in \delta$ and x match-relevant are right-ignoring. As presented above, testing whether a query is right ignoring can be done completely statically.

The right-ignorance of Q ensures thus that if the left-context of a match is fulfilled, then the right-context is also always satisfied. Hence, to check whether a node is a match, it suffices to look into the forest state in which $A_{\vec{G}}$ leaves a node, which synthesizes the information gained after visiting the left-context and the content of the node, depicted in dark grey in Figure 7.2:

Theorem 7.2: Let q_π be the forest state in which $A_{\vec{G}}$ leaves a node π. If Q is right-ignoring then $\pi \in \mathcal{M}_{Q,f}$ iff $y \in q_\pi$, $(y_1, x, y) \in \delta$ for some $y, y_1 \in Y$ and $x \in T$.

[1]We do not consider optimizations possible when the schema of the XML data is available.

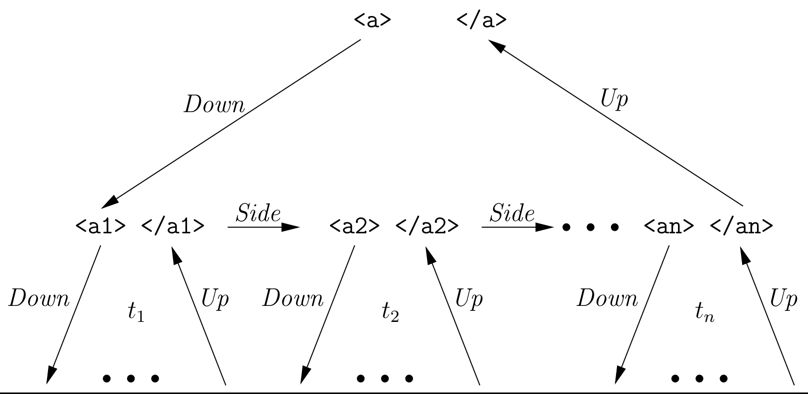


Figure 7.3: Event-driven run of a pushdown forest automaton

The theorem is proven as Theorem 7.4 in [Neu00]. □

To answer queries on XML streams without building the document tree in memory, it remains to show how a left-to-right pushdown automaton can be implemented in an event-based manner.

Event-driven Runs of Pushdown Forest Automata

Consider an LPA $A_{\vec{G}}$ =$(2^X, 2^Y, \{q_0\}, F, Down, Up, Side)$ as defined in Section 4.2.1 and its processing model as depicted in Figure 4.3 (on page 36). The order in which $A_{\vec{G}}$ visits the nodes of the input is the order of a depth-first, left-to-right search, which corresponds exactly to the document-order.

Compare Figure 4.3 (on page 36) and Figure 7.3. At every node π, $A_{\vec{G}}$ executes one *Down* transition at the moment when it proceeds to the content of π and one *Up* followed by one *Side* transitions at the moment when it finishes visiting the content of π. These moments correspond to the start and end tags, respectively, of the node π. The algorithm implementing the event-driven run of $A_{\vec{G}}$ is depicted in Listing 7.1.

We handle the following events:

1. `startDoc`, which is triggered before starting reading the stream;
2. `endDoc`, which is triggered after finishing reading the stream;
3. `enterNode`, which is triggered when a start-tag is read;
4. `leaveNode`, which is triggered when an end-tag is read.

Listing 7.1: Skeleton for the event-driven run of a pushdown forest automaton

```
Stack s;
ForestState q;

startDocHandler(){
  q := q0;
}

enterNodeHandler(Label a){
```

```
9    s.push(q);
10   q := Down(q,a) ;
11 }
12
13 leaveNodeHandler(Label a){
14   TreeState p = Up(q,a) ;
15   q := Side(s.pop(),a) ;
16 }
17
18 endDocHandler(){
19   if q ∈ F then output("Input accepted.")
20   else output("Input rejected.");
21 }
```

The stack declared in line 1 is needed in order to remember the forest states used for the traversal of the content of the elements opened and not yet closed. The variable q declared in line 2 stores the current forest state during the traversal of the document.

At the beginning, `startDocHandler` is called and it sets the current state to the initial state of the automaton (line 5). A start tag triggers a call of `enterNodeHandler` which remembers the current state on the stack (line 9) and updates the current state as result of executing the *Down* transition (line 10). An end-tag triggers the corresponding *Up* transition (line 14), followed by the *Side* transition which uses as forest state the last remembered state on the stack, i.e. the forest state before entering the element now ending (line 15).

The number of elements on the stack always equals the depth of the XML element currently handled. Hence the maximal height of the stack is the maximal depth of the handled XML document, which is in general rather small, even for very large documents.

Depending on the purpose of the pushdown automaton, other actions can be performed in the events handler. For the purpose of validation, it must be checked at the end of the document whether the current state is a final state (line 18).

For the purpose of answering right-ignoring queries it must be checked whether the forest state obtained after the side transition has the property stated in Proposition 7.2. Using the above presented implementation, $A_{\vec{G}}$ is thus able to answer right-ignoring queries on XML streams.

7.2 Arbitrary Queries

The previous section only shows how right-ignoring queries can be answered on XML streams. In this section we lift this limitation by showing how *arbitrary* queries can be answered on XML streams.

In the case of non right-ignoring queries, the decision as to whether a node is a match cannot be taken locally, i.e. at the time the node is left, because there is still match-relevant information in the part of the input not yet visited. The decision can only be taken after seeing all of the match-relevant information.

The general situation is depicted schematically in f_1 in Figure 7.4 (i). The node π is a potential match considering its left context and its content (depicted in dark grey) which can be checked by the time the end-tag of the node is seen.

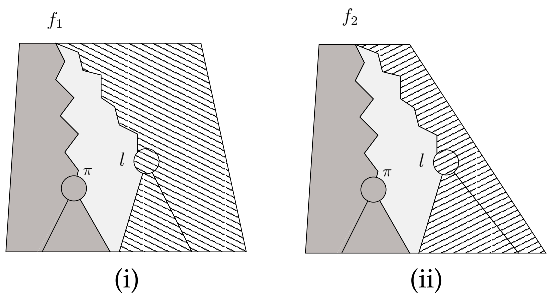


Figure 7.4: Right completion of a forest

The decision as to whether this is indeed a match node must be postponed until seeing the relevant part of the right context (depicted in light grey), which was empty in the particular case of right-ignoring queries. The location which must be reached in order to recognize π as a match is denoted as l. We call such a location l, *earliest detection location* of the match π, formally defined below.

Earliest Detection Locations

A forest f_2 is a *right-completion* of a forest f_1 at location $l \in L(f_1)$ iff f_1 and f_2 consist of the same events until l. (The tree representation of f_1 and f_2 are depicted in Figure 7.4). Formally:

$$f_2 \in \mathcal{R}ightCompl_{f_1,l} \text{ iff } \; prec_{f_1}(l) = prec_{f_2}(l) \text{ and } lab(f_2[\pi']) = lab(f_1[\pi']) \\ \text{for all } \pi' \in prec_{f_1}(l).$$

with $prec_f(l)$ denoting the *preceding nodes* of a location $l \in L(f)$ in a forest f, defined as $prec_f(l) = \{\pi \mid \pi \in N(f), \pi < l\}$, where "<" denotes lexicographical comparison.

A location l is an *early detection location* of a match node π for a query Q in input f_1 iff $\pi \in \mathcal{M}_{Q,f_2}$ for all right-completions f_2 of f_1 at l.

A location l is the *earliest detection location* of a match node π iff l is the smallest early detection location of π in lexicographic order.

Example 7.2: Reconsider Example 7.1 and the accompanying input depicted in Figure 7.1 (on page 86). Given the query `//a/b`, the earliest detection location of node 111 is 111. As for the query `//a[c]/b` the earliest detection location of node 111 is 112. Finally, for the query `/*[not(d)]//*`, there is no early detection location for any of the match nodes. These matches cannot be detected until the last location in the input has been reached. □

7.2.1 Idea

We proceed now to the description of the computation performed by our algorithm for evaluating grammar queries on XML streams. This can be seen as a run of a pushdown automaton changing its state on every XML event.

For the purpose of evaluation we use the stack to remember the following information for a location l at some nesting level :

1. q, denoting the progress within the content models to be considered on the level containing l. This is exactly the forest state in which $A_{\vec{G}}$ (the DLPA accepting $\mathcal{L}_G$) reaches l;
2. ri, needed for the early detection of matches as presented below;
3. m, storing potential matches which might be confirmed on the current level, as well as the potential matches accumulated while traversing the current level up to l.

For 1, we remember the states of the finite automata corresponding to the content models which are reached considering the content seen so far on the current level. They are obtained as by performing the transitions of $A_{\vec{G}}$.

For 2, we need to know which of the content models considered on the current level occur in right-ignoring contexts. A content model for an element e occurs in a right ignoring context iff there is no content model of an enclosing element whose fulfillment depends on the right context of e. We call such content models *right-ignoring content models*.

For 3, we associate potential matches with NFA states from q. A potential match is associated with a state y at location l iff the match may be defined w.r.t. derivations in which the word of non-terminals on the current level is accepted by an NFA run reaching l in state y. The information m can be thus represented as a partial mapping from NFA states y to the corresponding potential matches $m(y)$.

Consider our query $Q = (G, T)$ with a forest grammar $G = (R, r_0)$. Let $r_1, \ldots, r_p$ be the regular expressions occurring on the right-hand sides in the productions R, where p is the number of productions. For $0 \leq j \leq p$, let $A_j = (Y_j, y_{0,j}, F_j, \delta_j)$ be the NFA accepting the regular string language defined by r_j as obtained by the Berry-Sethi construction. By possibly renaming the NFA states we can always ensure that $Y_i \cap Y_j = \emptyset$ for $i \neq j$. Let $Y = Y_0 \cup \ldots \cup Y_p$ and $\delta = \delta_0 \cup \ldots \cup \delta_p$.

Initial State

Initially, we start with the NFA start state $y_{0,0}$ of the start content model r_0. The content model r_0 is right-ignoring as there are no enclosing elements. Also, there are no potential matches detected yet, thus the information initially remembered on the stack consists of:

$$q_0 = \{y_{0,0}\},\ ri_0 = \{r_0\},\ m_0 = \emptyset$$

Start-Tag Transitions

On a start-tag event `<a>` at location l, new information (q_1, ri_1, m_1) is pushed on the stack, depending on the information in the current top of the stack (q, ri, m) as follows.

The possible content models of the current element are computed from the content models in which the element may occur (as in the case of a *Down* transition in $A_{\vec{G}}$). Before seeing any of the children of the current element we are in the initial NFA state of these content models:

$$q_1 = \{y_{0,j} \mid y \in q,\ (y, x, y_1) \in \delta,\ x \to a\langle r_j \rangle\}$$

A content model r_j considered for the current element l is right-ignoring if (1) the surrounding content model r_k is right ignoring and (2) r_k is fulfilled independently of how the right siblings of l might look like. Condition (1) can be looked up in ri. To ensure (2) there must be a right-ignoring NFA state y_1 reachable after seeing the left siblings of l. Thus:

$$ri_1 = \{r_j \mid y \in q,\ (y, x, y_1) \in \delta_k,\ x \rightarrow a\langle r_j\rangle, r_k \in ri, rightIgn(y_1)\}$$

As for the potential matches which might be confirmed while visiting the content of l, these are the matches propagated so far for which the content of the current level is fulfilled whatever follows after l. We add l to these potential matches if it can be derived from a target non-terminal considering its left-context, and its right-context is irrelevant (that is l's confirmation as a match depends thus only on its content).

$$\begin{aligned} m_1(y_{0,j}) =& \bigcup\{m(y) \mid y \in q,\ (y, x, y_1) \in \delta_k,\ x \rightarrow a\langle r_j\rangle, r_k \in ri, rightIgn(y_1)\} \\ & \cup \\ & \{l \mid y \in q,\ (y, x, y_1) \in \delta_k,\ x \rightarrow a\langle r_j\rangle, r_k \in ri, rightIgn(y_1), x \in T\} \end{aligned} \tag{7.1}$$

End-Tag Transitions

An end-tag event `</a>` at location $\pi i(n+1)$ signals that the processing of the sequence of children $\pi i1, \ldots, \pi in$ is completed and the computation has to return to the nesting level and advance over the father node πi. The top two elements on the stack at this moment: (q, ri, m) and (q_1, ri_1, m_1) store the state of the computation after seeing the children and the left siblings of πi, respectively. The elements (q, ri, m) and (q_1, ri_1, m_1) are consumed from the stack and used to compute the new top of the stack (q_2, ri_2, m_2), reflecting the state after finishing seeing πi, as follows.

A content model r_j is fulfilled by the children of πi iff there is some $y_2 \in q \cap F_j$, i.e. an NFA final state for r_j is reached after traversing them. It follows that πi can be derived from symbols x for which there is a production $x \rightarrow a\langle r_j\rangle$. The advance in the content models on the level of πi, after seeing πi is obtained by considering NFA transitions with symbols x from which πi may be derived. This is completely similar to an *Up* transition followed by a *Side* transition in $A_{\vec{G}}$ and is summarized by:

$$q_2 = \{y_1 \mid y_2 \in q \cap F_j, x \rightarrow a\langle r_j\rangle, y \in q_1, (y, x, y_1) \in \delta\}$$

As the set of right ignoring content models only depends on the surrounding content models, it remains unchanged for a whole nesting level, that is :

$$ri_2 = ri_1$$

As for the potential matches, we have to aggregate the potential matches from the left-context of πi with those from its content. More precisely, potential matches defined by an NFA run on the children level are joined with potential matches from the left context associated with NFA states which are reached in nesting NFA runs after seeing the father node. The father node, πi is added as a potential match if it can be derived from a target non-terminal:

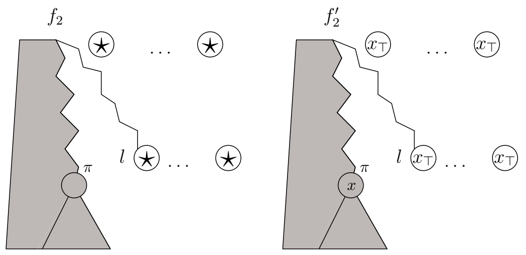


Figure 7.5: Right completion at l and corresponding derivation

$$\begin{aligned} m_2(y_1) = \{m(y_2) \cup m_1(y) \mid y_2 \in q \cap F_j, x \to a\langle r_j \rangle, y \in q_1, (y, x, y_1) \in \delta\} \cup \\ \{\pi i \mid y_2 \in q \cap F_j, x \to a\langle r_j \rangle, y \in q_1, (y, x, y_1) \in \delta, x \in T\} \end{aligned} \tag{7.2}$$

7.2.2 Recognizing Matches

The construction above allows the location of matches as stated by the following theorem:

Theorem 7.3: A location l is an early detection location for a match node π iff $\pi \in m(y)$, $r_j \in ri$ and $rightIgn(y)$ for some $y \in q \cap Y_j$ with (q, m, ri) being the top of the stack at event l.

The complete proof is given in Appendix A.4. The idea of the proof is described next.

Let l be an early detection location for a match node π. Let f_2 be a right-completion obtained from f_1 by adding on every level from the root to l an arbitrary number of right siblings $\star\langle\rangle$ (as depicted in Figure 7.5), where $\star$ is a symbol not occurring in any of the rules in the grammar. By the definition of early detection locations there is a derivation f_2' of f_2 in which π is labeled x for some $x \in T$. Also, since $\star$ does not occur in any rule, f_2' must label all the $\star$ nodes with $x_\top$. The y with the properties as required by this theorem is the NFA state in which the location l is reached within the NFA accepting run corresponding to f_2'.

Conversely, let (q, m, ri) be the top of the stack at event l and let $\pi \in m(y)$, $r_j \in ri$ and $rightIgn(y)$ for some $y \in q \cap Y_j$. From $\pi \in m(y)$ it follows that there is a relabeling of the nodes visited so far in which π is labeled with some $x \in T$ and which might be completed to a whole derivation according to the grammar G using $x_\top$ symbols for the not yet visited nodes. The existence of the completion on the current level follows from $rightIgn(y)$, while the existence of the completions on the enclosing levels is ensured by the condition $r_j \in ri$.

$\square$

As locations are visited in lexicographic order, testing the condition in Theorem 7.3 ensures that every match π is detected when reaching its earliest detection location. This proves Theorem 7.1.

7.2.3 Implementation

The algorithm implementing the event-driven evaluation of the queries as above is given in Listing 7.2. We assume that `enterNodeHandler` and `leaveNodeHandler` receive as an argument, besides the label of the currently read node, also the currently reached location. For the case in which the current location is not provided by the event-based parser, note that it can be easily propagated along the event handlers in the internal parse-state. The algorithm basically follows the computation rules given above while sharing the commonalities in the rules for q, ri and m. As an abbreviation we use the operator $\oplus$ to add a new entry or update an existing set entry in a mapping m via set union.

Listing 7.2: Algorithm for event-driven query answering

```
Stack s;

enterNodeHandler(Location $l$, Label a){
  ($q$,$ri$,$m$) := s.top();
  $q_1$ := $ri_1$ := $m_1$ := $\emptyset$;

  forall $y \in q$ with $(y,x,y_1) \in \delta_k$ and $x \to a\langle r_j \rangle$
    $q_1$ := $q_1 \cup \{y_{0,j}\}$
    if $rightIgn(y_1)$ and $r_k \in ri$ then
      $ri_1$ := $ri_1 \cup \{r_j\}$;
      if $rightIgn(y_{0,j})$ then
        reportMatches($m(y)$);
        if $x \in T$ then reportMatches($\{l\}$) endif
      else
        $m_1$ := $m_1 \oplus \{y_{0,j} \mapsto m(y)\}$;
        if $x \in T$ then $m_1$ := $m_1 \oplus \{y_{0,j} \mapsto \{l\}\}$ endif
      endif
    endif
  endfor

  s.push(($q_1$,$ri_1$,$m_1$));
}

leaveNodeHandler(Location $ln$, Label a){
  ($q$,$ri$,$m$) := s.pop();
  ($q_1$,$ri_1$,$m_1$) := s.pop();
  $q_2$ := $m_2$ := $\emptyset$;
  $ri_2$ := $ri_1$;

  forall $y \in q_1$,$y_2 \in q$,$y_2 \in F_j$,$x \to a\langle r_j \rangle$ and $(y,x,y_1) \in \delta_k$
    $q_2$ := $q_2 \oplus \{y_1\}$;
    if $r_j \in ri$ then reportMatches($m(y_2)$)
    else
      $m_2$ := $m_2 \oplus \{y_1 \mapsto m(y_2)\}$;
      $m_2$ := $m_2 \oplus \{y_1 \mapsto m(y)\}$;
      if $x \in T$ then $m_2$ := $m_2 \oplus \{y_1 \mapsto \{l\}\}$ endif
    endif
  endfor

  s.push(($q_2$,$ri_2$,$m_2$));
}

startDocHandler(){s.push(($q_0$,$\{r_0\}$,$\emptyset$));}
```

```

endDocHandler(){
  $(q,ri,m)$ := s.pop();

  forall $y \in q \cap F_0$
    reportMatches($m(y)$);
  endfor
}
```

Matches are detected when their earliest detection location is reached, i.e. at the event-handler executed at the immediately preceding location. This might be the case either at start or at end tags. At start tags (line 12) we report potential matches for which we know that (a) the right siblings at the current level are irrelevant (condition $rightIgn(y_1)$ tested in line 9); (b) the right siblings of the ancestors are irrelevant (condition $r_k \in ri$ tested in line 9) and (c) the content is irrelevant (condition $rightIgn(y_{0,j})$ in line 11).

At end tags (line 32) we report potential matches for which the content was fulfilled (condition $y_2 \in F_j$ in line 30) and the upper-right context is irrelevant (condition $r_j \in ri$).

Note that there is no need to propagate a confirmed match π beyond its earliest detection location l where it is reported. (see tests in lines 11 and 32).

Also, potential matches are discarded implicitly precisely as soon as enough information is seen in order to reject them. Potential matches $m(y)$ at a location l are no longer propagated when y is not involved in the NFA transitions. This happens at end tag events if there is no transition (y, x, y_1) in any of the possible content models. Also at end tag events, potential matches in $m(y)$ are discarded if y is not a final state in any of the considered content-models on the finished level. Thereby matches are remembered only as long as the strictly necessary portion of the input has been seen in order to confirm them.

Finally, at the end of the input potential matches not yet confirmed and conforming to the top-level content model (condition $y \in q \cap F_0$ in line 48) are reported as matches in line 49.

7.2.4 Complexity

Let $|D|$ be the size of the input data, i.e. the number of nodes in it. The size of a query Q can be estimated as the number of NFA states $|Y|$ plus the number of non-terminals $|X|$. Let pot_{max} be the maximum number of potential match nodes at any given time during the traversal.

For every node in the input `enterNodeHandler` and `leaveNodeHandler` is called once. In `enterNodeHandler` at πi, the loop starting at line 7 is executed for every $y \in q$, for every outgoing NFA transition (y, x, y_1) and for all content models r_j for x. The size of q is in $O(|q_{max}|)$, where q_{max} is the forest state q with the maximum number of elements. Let cm_{max} be the maximum number of content models considered on a level and let out_{max} be the maximum number of outgoing NFA transitions from an NFA state. The loop is executed thus up to $|q_{max}| \cdot out_{max} \cdot cm_{max}$ times.

The set union in line 8 can be computed in time $O(|q_{max}|)$. The set union in line 10 needs time $O(cm_{max})$. Reporting the confirmed matches additionally requires time $O(pot_{max})$. Finally the set unions in lines 15 and 16 necessitate again $O(pot_{max})$ time. A call to `enterNodeHandler` amounts thus to $O(|q_{max}| \cdot out_{max} \cdot cm_{max} \cdot (|q_{max}| + cm_{max} + pot_{max}))$ time.

In `leaveNodeHandler`, the loop starting at line 30 is executed in the worst case, for every $y \in q_1$ and every $y_2 \in q$, i.e. up to $|q_{max}|^2$ times. The set union in line 31 is computed in time $O(|q_{max}|)$. Reporting the confirmed matches possibly adds $O(pot_{max})$ time. The set unions in lines 34, 35 and 36 need $O(pot_{max})$ time. A call to `leaveNodeHandler` amounts thus to $O(|q_{max}|^2 \cdot (|q_{max}| + pot_{max}))$ time.

As `leaveNodeHandler` and `enterNodeHandler` are called each once for every node, the overall time complexity of event driven evaluation of queries is thus in $O(|D| \cdot (|q_{max}| \cdot out_{max} \cdot cm_{max} \cdot (|q_{max}| + cm_{max} + pot_{max}) + |q_{max}|^2 \cdot (|q_{max}| + pot_{max})))$. The values of $|q_{max}|$, out_{max} and cm_{max} are bounded by values which do not depend on $|D|$. Experimental evidence show them to be small, and correspondingly the algorithm scales well with the size of the query as presented in the next section.

The worst complexity in the size of the document is obtained for $pot_{max} = |D|$, in the case where all the nodes are potential matches until the end of the document. In general, however, the number of potential matches is much less than the total number of nodes ($pot_{max} \ll |D|$) and can be assimilated with a constant. In this case we obtain a time linear in the size of the document, as suggested by our experimental results.

As for the space complexity, let d be the depth of the input document. During the scan of the document we store at each location the (q, ri, m) tuples for all ancestor locations up to the root, which correspond to the opened and not yet closed elements at the current location. For every level, q has up to $|q_{max}|$ elements, m stores up to $|q_{max}| \cdot pot_{max}$ locations and ri up to cm_{max} content models. As all these elements can be stored in constant space and the height of the stack is at most d, we obtain the worst case space complexity $O(d \cdot (|q_{max}| + |q_{max}| \cdot pot_{max} + cm_{max}))$. Most of the practical queries need only a small amount of memory, as the information relevant to whether a node is a match is typically located in the relative proximity of the node (that is pot_{max} is small).

7.2.5 Experimental Results

The algorithm presented here has been completely implemented in Fxgrep. Even though many proposals for evaluating XML queries on streams exist, there are surprisingly few tools publicly available. Furthermore, most of the proposals for which public implementations exist impose serious limitations on Core XPath (see Section 7.3 on related work). A more mature implementation we were able to experiment with was SPEX [OFB04] which covers a large subset of XPath. As a reference for the in-memory DOM approach we used Xalan-Java 2.6.0 [Pro05] one of the most popular XSLT processors, which also provides a command line XPath processor.

We used for our experiments the Protein Sequence Database [PSD], an XML document of over 700 MB size, containing around 25 million nodes with a maximal depth of 7 and an average depth of approximately 5. The experiments were performed on an Athlon XP 3000+ with 1GB of memory running under Linux (kernel version 2.6.8).

Even though the querying capabilities of Fxgrep go beyond those of Core

	Fxgrep			Spex			Xalan		
	T	P	R	T	P	R	T	P	R
Q_1	29.8	8.7	3.4	9.1	4.1	2.2	15.8	2.5	6.3
Q_2	31.9	8.7	3.6	21.2	4.1	5.1	22.0	2.5	8.8
Q_3	33.5	8.7	3.8	50.9	4.1	12.4	178.8	2.5	71

Table 7.1: Evaluation times (in seconds) for increasingly complex queries

XPath , for the comparison with other query tools for XML streams we had to limit ourselves to queries expressible in Core XPath. We used the following queries:

Q_1: As a representative of simple queries, specifying only the path to the matches, we chose Q_1 looking for protein entries containing a reference with an author element. The query is expressible with the XPath pattern `//ProteinEntry//refinfo/author`.

Q_2: A slightly more complex query, containing simple structure qualifiers is Q_2. The query locates authors of entries, the description of which contains the word "iron" and s.t. the year "2000" is mentioned among its references, i.e. the XPath pattern

```
//ProteinEntry[//description[contains(.,'iron')]]
  //refinfo[//year[contains(.,'2000')]]//author.
```

Q_3: Finally we use a more complex query Q_3 locating authors of proteins containing a reference to the year "2000", which are followed by two protein entries, the descriptions of which contain the word "iron".

Table 7.1 presents the evaluation times for Q_1, Q_2 and Q_3 on a fragment of the database of 16 MB size. Absolute times are difficult to compare in the case of tools implemented in different programming languages. Fxgrep and its underlying parser Fxp are written in SML, while SPEX and Xalan are written in Java. The SML parser is significantly slower than the Java parsers in the case of large documents as in our experiment. That is, the events are delivered at different paces by the parsers used in the three applications. It is a situation similar to comparing absolute time measurements obtained using CPUs with different tact frequencies. A sensible way to account for this is to divide the absolute times by the frequency. Therefore, besides the total time in column T, we list the parsing time in column P, as well as the relative speed in column R obtained by dividing the total time by the parsing time.

The absolute times are depicted in Figure 7.6. One remarkable feature of Fxgrep is that the evaluation time does not significantly depend on the complexity of the query as opposed to the other tools. The reason is the expressiveness of the underlying grammar formalism in which adding supplementary contextual conditions does not significantly change the size of the underlying grammar. Interestingly enough, Fxgrep performs better even in absolute terms as the query complexity increases.

The relative times are depicted in Figure 7.7. The numbers denote how many times slower the query evaluation is as compared to the generation of the event stream. In Fxgrep around 30 percent of the evaluation time is needed for generating the stream of events. Fxgrep's throughput is comparable on

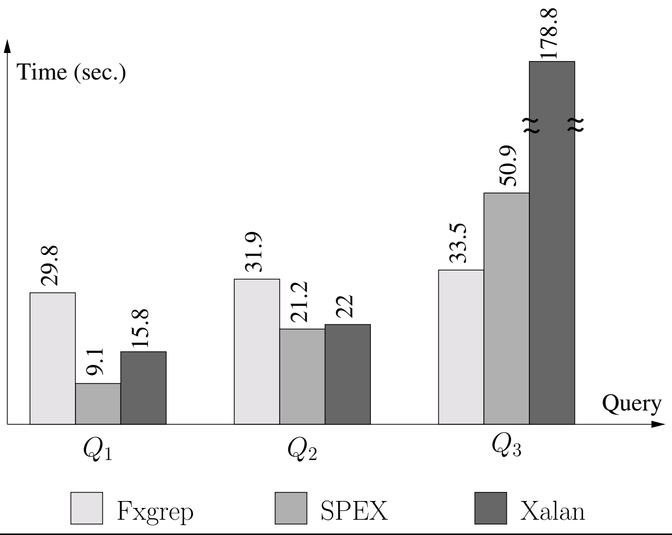


Figure 7.6: Absolute evaluation times for increasingly complex queries

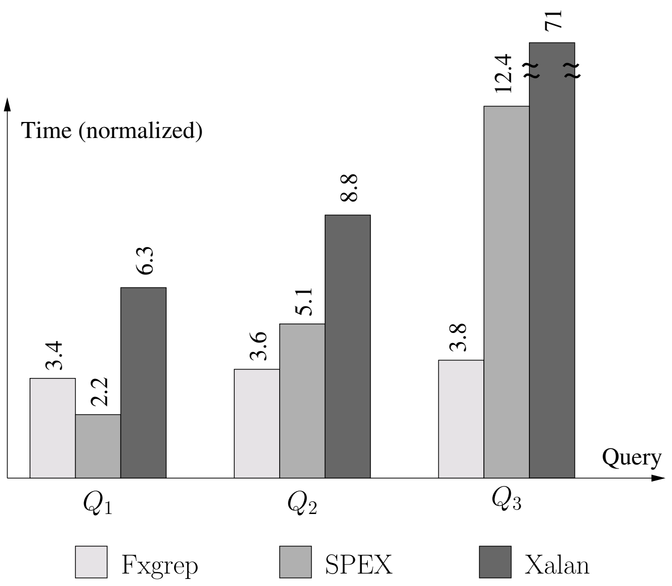


Figure 7.7: Relative evaluation times for increasingly complex queries

	Fxgrep	Spex	Xalan
3 MB	9.8	10.2	10.1
16 MB	33.5	50.9	178.8
32 MB	61.8	96.8	614.6
159 MB	338	446	n/a

Table 7.2: Evaluation times (in seconds) for Q_3 for increasing document sizes

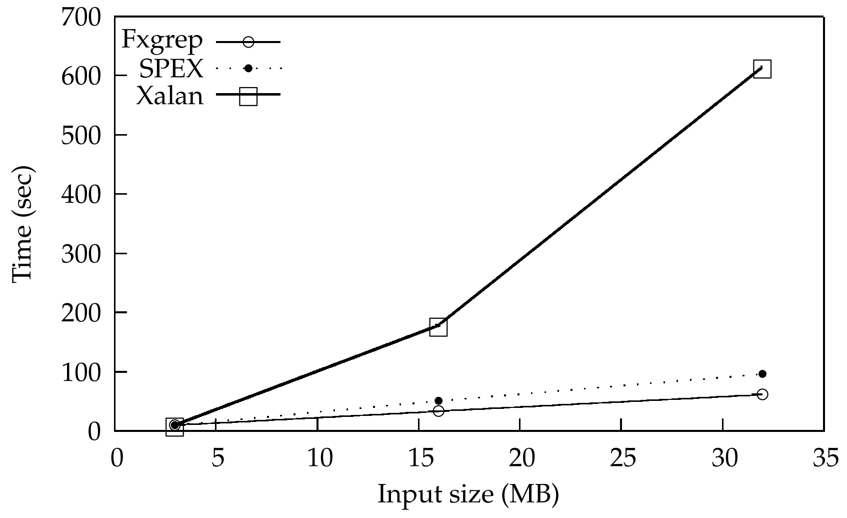


Figure 7.8: Scalability with the input size

average with the relative throughput of SPEX and better than what is achieved in Xalan.

Table 7.2 presents the dependency of the evaluation times on the size of the document. We chose fragments of the input of increasing sizes and evaluated query Q_3. As depicted in Figure 7.8 (containing the results for 3MB, 16MB and 32 MB size, respectively) the evaluation time increases linearly for Fxgrep and SPEX, as opposed to Xalan. This shows that the event-based processing mode of Fxgrep scales well with the input size, as presented in the complexity assessment presented in Section 7.2.4.

As for the memory usage, Fxgrep and SPEX need a constant space for all runs of up to 10 and 15 MB, respectively, including the SML runtime system and the Java Virtual Machine. As opposed to this, Xalan needs a multiple of the size of the handled input document. Even a memory space of 1 GB was not enough for Xalan in order to process the 159 MB large input.

7.3 Bibliographical Notes

A basic task in XML processing is XML validation. The problem of validating XML streams is addressed by Segoufin and Vianu in [SV02] and Chitic and Rosu in [CR04]. As mentioned in Section 4.1.3, XML schema languages are basically regular forest languages, hence conformance to such a schema can be checked by a pushdown forest automaton. As presented in this chapter this can be performed efficiently on XML streams in the event-based manner.

Many research works deal with querying of XML streams. Most of them consider subsets of XPath. Some of them deal with XQuery, which in fact is more than a querying language as it allows the transformation of the input. In the following we are mainly interested in the querying capabilities of the considered languages.

Conventional attribute grammars (AG) and compositions thereof are proposed by Nakano and Nishimura in [NN01] as a means of specifying tree transformations. An algorithm is presented which allows an event-driven evaluation of attribute values. Specifying transformations, or in particular queries, using AG is however quite elaborate even for simple context-dependent queries and AG are restricted to use attributes of non-terminal symbols at most once in a rule. Also as no stack is used input trees have to be restricted to a maximum nesting depth.

More suited for XML are attribute grammars based on forest grammars as considered in XML Stream Attribute Grammars (XSAGs) [KS03] and TransformX [SK05][2]. A restricted form of attribute forest grammars is considered which allows the evaluation of attributes on XML streams. The attribute grammars have to be L-attributed, i.e. to allow their evaluation in a single pass in document-order. Another necessary restriction is that the regular expressions in productions are *unambiguous*, as in the case of DTDs. This ensures that every parsed element corresponds to exactly one symbol in the content model of the corresponding production, which allows the unambiguous specification and evaluation of attributes. While XSAGs are targeted at ensuring scalability and have the expressiveness of deterministic pushdown transducers, the TransformX AGs allow the specification of the attribution functions in a Turing-complete programming language (Java). In both cases, for the evaluation of the attribute grammars pushdown transducers are used. The pushdown transducers used in TransformX [SK05] validate the input according to the grammar in a similar manner to the pushdown forest automata. Additionally, a sequence of attribution functions is generated as specified by the attribute grammar. A second transducer uses this sequence and performs the specified computation. For the identification of the non-terminals from which nodes are derived in the (unique) parse tree, as needed for the evaluation of the AGs in [KS03, SK05], pushdown forest automata can be used. The unambiguousness restriction of the attribute forest grammars allows one to proceed as in the case of right-ignoring queries presented in Section 7.1. That is, the non-terminal corresponding to the current node can be directly determined from the (single) NFA state in the current forest state, as it does not depend on the events after the current one.

A number of approaches handle the problem of querying XML streams in the context of selective dissemination of information (SDI), also known as XML message brokering [CDTW00, AF00, DFFT02, DF03, ACGG$^+$02, GMOS03, GS03, CFGR02]. In this scenario a large number of users subscribe to a dissemination system by specifying a query which acts like a filter for the documents of interest. Given an input document, the system simultaneously evaluates all user queries and distributes it to the users whose queries lead to at least one

[2]In these works forest grammars are called extended regular tree grammars.

match. Strictly speaking, the queries are not answered. The documents which contain matches are dispatched but the location of the matches is not reported. XFilter [AF00] handle simple XPath patterns, i.e. without nested XPath patterns as filters. These can be expressed with regular expressions, hence they are implemented using finite string automata. YFilter [DFFT02] improves on XFilter by eliminating redundant processing by sharing common paths in expressions. More recently, in [DF03], the querying capabilities are extended to handle filters comparing attributes or text data of elements with constants and nested path expressions are allowed to occur basically only for the last location step. Green *et al.* [GMOS03] consider regular path expressions without filters. It is shown that a lazy construction of the DFA resulting from multiple XPath expressions can avoid the exponential blow-up in the number of states for a large number of queries. XPush [GS03] also handles nested path expressions and addresses the problem of sharing both path navigation and predicate evaluation among multiple patterns. XTrie [CFGR02] considers a query language which allows the specification of nested path expressions and, besides, an order in which they are to be satisfied. Even though Fxgrep is not targeted at SDI, note that it basically exceeds the essential capabilities of all previously mentioned query languages.

There are a number of approaches in which queries on XML streams are answered by constructing a network of transducers [LMP02, PC03, OFB04, Feg04]. A query is there compiled into a number of interconnected transducers, each of them taking as input one or more streams and producing one or more output streams by possibly using a local buffer. The XML input is delivered to one start transducer and the matches are collected from one output transducer. The query language of XSM [LMP02] handles only XPath patterns, without filters and deep matching (//), but allows instead value-based joins. XSQ [PC03] deals with XPath patterns in which at most one filter can be specified for a node and filters cannot occur inside another filter. The filters only allow the comparison of the text content of a child element or an attribute with a constant. SPEX [OFB04] basically covers Core XPath. Each transducer in the network processes the input stream and transmits it augmented with computed information to its successors. The number of transducers is linear in the query size. The complexity of answering queries depends on whether filters are allowed and is polynomial in both the size of the query and of the input. XStreamQuery [Feg04] is an XQuery engine based on a pipeline of SAX-like event handlers augmented with the possibility of returning feedback to the producer. The strengths of this construction are its simplicity and the ability to ignore irrelevant events as soon as possible. However, the approach only handles the child and descendant axes as yet.

FluXQuery [KSSS04] extends a subset of XQuery with constructs which guide an event-based processing of the queries using the DTD of the input. FluXQuery is used within the StreamGlobe project which is concerned with query evaluation on data streams in distributed, heterogeneous environments [SKK04]. STX [Bec03] is basically a restriction of the XSLT transformation language to what can be handled locally by considering only the visited part of the tree and selecting nodes from the remaining part of the tree. Sequential XPath [Des01] presents a quite restricted subset of XPath, handling only right-ignoring XPath patterns, which can be implemented without the need of any buffering. TurboXPath [JFB05] introduces an algorithm for answering XPath queries containing both arithmetic and structural predicates and which is neither directly based on finite automata nor on transducer networks. The dynamic data structure WA (*work array*), used to match the document nodes has certain similarities with our construction. Entries are added in the WA upon each start-tag event for each sub-pattern to which the children must conform, which roughly correspond to a *Down* transition of the LPA. Matches of the sub-patterns are detected upon end-tag events by AND-ing the fulfillment of the sub-patterns by the children, similarly to an *Up* transition. *Side* transitions are not needed as the pattern language does not impose any order on the children nodes. In this perspective the context information is optimally used, as in our case, by a combination of top-down and bottom-up transitions. Recent work by Bar-Yossef *et al.* [BYFJ04], indicates that the space requirement for the TurboXPath approach is near the theoretical optimum for XPath queries.

Part II

An XML Transformation Language

Chapter 8

Introduction

One of the main advantages of encoding information in XML format is that it offers a platform-independent representation suitable for information exchange between applications. XML is often deemed the "de facto standard for data exchange in Internet". XML documents used in information exchange are intermediate representations which have to be transformed by the communicating entities into their internal formats. Transforming XML documents is therefore a task to be accomplished by any application which uses an XML communication interface.

Also, XML documents used as a permanent rather than temporary representation are frequently transformed to other different formats. In this usage scenario, an XML document typically stores some information content, which is transformed to different layout formats for different presentation purposes, e.g. HTML for presence on the Internet, PDF for printed media, WML for mobile telephones, and so on. This usage scenario has coined the term "stylesheet" for XML transformations, as these can be considered style indications to be followed in order to present the informational content in the specified layout.

Transforming is thus a task inherently related with XML processing, both for XML as transient and as permanent information representation. Correspondingly, many proposals exist as to how this could be best accomplished in different usage scenarios.

A straightforward possibility is to use already available general purpose programming languages to perform the desired transformation tasks. As every general purpose programming language can be used to perform any computable transformation, a programmer can choose in principle his favorite language. As the programmer has total control over the computation to be performed, this approach has the advantage that the application can be tailored for optimally performing the task at hand.

Many libraries exist to support XML processing in most of the programming languages, e.g. Java, C, C#, SML, Caml, Haskell and many others. Moreover, in order to support the seamless integration and interchangeability of XML libraries, efforts have been made to standardize their functionality, for example SAX [SAX98] and XMLPull [XML98b] for parsing, or DOM [DOM98] for representing and accessing XML documents. The predetermined syntax of the general purpose programming languages however makes even simple transformations tedious to write or understand. Moreover, non-programmers are precluded from specifying even very simple transformations using this approach. This is a serious limitation as XML processing has to be performed by increasingly many non-programmers.

Rather than using general purpose programming languages, it is thus preferable to have languages dedicated to XML processing, in particular for XML transformations. They typically offer the convenience of an XML suitable syntax which allow for more concise and, for the most part, more intuitive specifications. Ideally, they should allow the convenient specification of simple transformations, while offering the whole power of a general purpose programming language for more elaborate transformations.

The most prominent transformation languages dedicated to XML available at this time are XSLT [XSL99] and XQuery [XQu05a], both proposed by the W3C consortium. Rather than imperative, as are the most used programming languages, XSLT and XQuery are intended to be declarative languages in order to provide a more intuitive way of specifying transformations. XSLT is a W3C Recommendation released in 1999 and as a rule-based transformation language is accessible even to non-programmers. Support for XSLT is already available in most modern Web browsers, where it may be used for formating Web pages stored as XML documents. XQuery, at the time of writing still a W3C Working Draft, is similar in syntax and style to the relational query language for databases SQL, and is targeted more at programmers. While XQuery continuously increases its importance, it will probably not replace the need for rule-based stylesheet languages like XSLT.

XSLT's and XQuery's goal of declarativeness is however only partially achieved due to their dependence on XPath as a sub-language for addressing nodes of interests. Rather than specifying *what* are the properties of the nodes to be selected, XPath specifies *how* to navigate within the input in order to reach them.

We therefore chose to develop a new XML transformation language, Fxt , which uses the declarative querying capabilities presented in the previous part. The contributions presented in this second part are as follows.

Fxt: a declarative, rule-based XML transformation language In the construction of Fxt we build upon the Fxgrep pattern language introduced in Chapter 3. Fxt uses the expressiveness of Fxgrep patterns to allow for intuitive rule-based transformations. Simultaneously it allows the usage of the full power of a general purpose programming language, SML [MTHM97, SML05], by making possible the integration of SML code into Fxt transformations.

Binary queries for rule-based transformations Querying is, as mentioned, at the base of transformation languages. Traditionally, transformation languages use only unary queries, as for example XPath patterns in XQuery and XSLT. The nodes identified by a unary query are obtained by evaluating the query in the context of a given node. This context node is often in turn identified by another unary query. For example, in rule-based languages a match pattern identifies the nodes on which a rule is applicable. When the rule is applied on such a node, a select pattern is typically used to locate nodes of interest in the context of this node. The current node together with each selected node are elements of a binary relation. This relation can be specified in the global context of the root node via a binary query. This is easily possible, because the context nodes are always identified in the global context of the root. As Fxgrep is able to express binary queries, we may use its binary patterns as a a means of selection in Fxt. Rather than explicitely specifying how to navigate from the context node in order to reach the nodes to be selected, as for example with XPath select patterns, we indicate where are these nodes located in the global context. This is in our view another improvement in terms of declarativeness. Moreover, since binary queries can be efficiently implemented as presented in Section 5.3, it turns out that using binary queries can mean an important gain of efficiency, as shown in Fxt.

Binary queries for other approaches to XML transformations While binary queries are especially suited as a means of selecting in rule-base transformations, they can be also very useful in the implementation of other approaches to XML transformations. A typical usage of queries, for instance, is to iterate over the set of nodes located by one unary query q_1 and use a second unary query q_2 to associate each node n_1 selected by q_1 with the set of nodes selected by q_2 in the context of n_1. This is for example the case in XQuery where nodes from the input XML data are selected via an XPath pattern in the context of nodes previously selected by another XPath pattern. Given the similarities with the use of match and select patterns in rule-based transformations, we suggest how binary queries can be similarly employed in these cases to support efficient implementations.

This part is organized as follows. The basic capabilities of Fxt are introduced in Chapter 9. In Chapter 10 we show by means of Fxt how binary queries can be used for efficient and declarative rule-based XML transformations. Implementation issues for Fxt are discussed in Chapter 11. Also, we present experimental results assessing Fxt's performance as compared to established XML transformation tools. In Chapter 12 we relate Fxt to the other transformation languages proposed for XML, including the standardized languages XSLT and XQuery. We compare them with Fxt in terms of expressiveness and present how the ideas used in Fxt can be put to work in their implementation.

Chapter 9

A Primer for Fxt

In this chapter we introduce Fxt [1], a rule-based XML transformation language constructed on top of the querying capabilities presented in the first part of the work. On the one hand, we want to provide the non-specialist user with a tool by which transformational ideas can be expressed conveniently, i.e., without resorting to a general-purpose programming language. Therefore we choose the intuitive and comfortable rule-based approach to XML transformations. On the other hand, however, we also want to support XML processing for functional programmers. For this we allow the formulation of more elaborate transformations to be flexibly achieved by hooks provided to the full functionality of the SML programming language.

Our three main design goals are:

- to allow the specification of the intended transformations as declarative as possible;
- to provide only primitives, in particular for pattern matching, which can be implemented efficiently;
- to allow maximal flexibility by fully integrating an external programming interface.

In the rest of this chapter we present the basic capabilities of Fxt. It is mainly meant as a primer to the specification language of Fxt XML transformations, which we call again Fxt. For convenience, we use in the following the term Fxt transformation or stylesheet to denote the specification of an XML transformation in Fxt.

9.1 The Transformation Model of Fxt

XML transformation tools typically expect a transformation specification and some XML input, and produce the specified output as depicted in Figure 9.1. Fxt can be used in this manner as exemplified below.

[1]Fxt is an acronym for "the functional XML transformer", where "functional" denotes that Fxt is written in a functional programming language (SML).

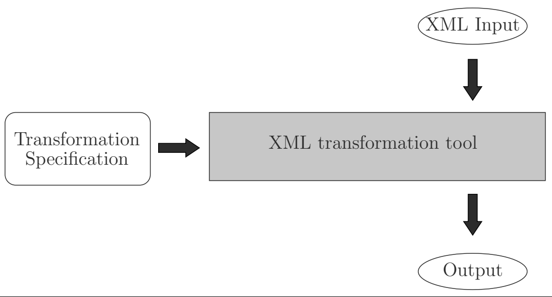


Figure 9.1: The interpreter approach to XML transformations

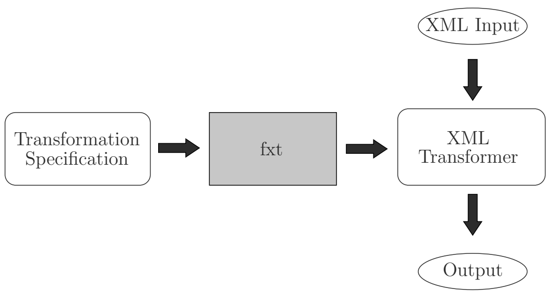


Figure 9.2: Fxt as compiler of XML transformations

Example 9.1:
An XML input file `input.xml` is transformed by Fxt into an output file `output.xml` as specified in a stylesheet located in a file `transform.tf` by the following command:

```
> fxt -i input.xml -o output.xml transform.tf
```

□

Fxt achieves this by compiling the specification of the XML transformation into an XML transformer as depicted in Figure 9.2. The generated transformer can be subsequently used to transform an XML input document as specified. The output of a transformation typically is an XML document, but in general may be some arbitrary content.

Obviously, the advantages of this approach are those of a compiler over an interpreter, namely that the specification is processed only once, independently of how many times the transformation is applied on different input documents. It is convenient to proceed in this manner especially in XML processing where typically the same transformation is meant for a whole class of documents rather than for a single one.

Example 9.2: Given an Fxt transformation specified in a file `transform.tf`, the compilation of the transformation is achieved by executing:

```
> fxt transform.tf
```

As a result, a transformation program is generated and compiled to machine code. The specified transformation can be subsequently achieved by executing

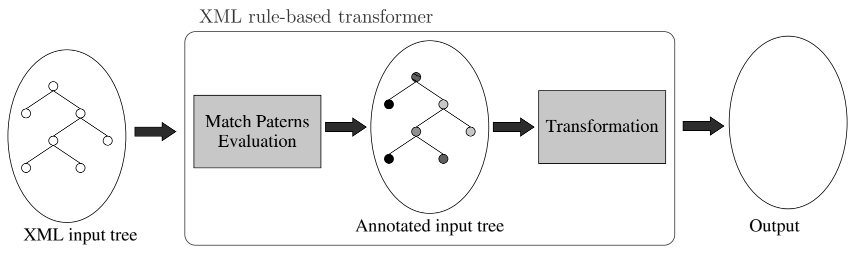


Figure 9.3: Phases of an XML transformation

the generated code on some XML input data. For instance, applying the transformation on an XML input file `input1.xml` to produce output file `output1.xml` is performed by:

```
> transform.sh -i input1.xml -o output1.xml
```

The same transformation can be invoked repeatedly for example to transform a file `input2.xml` to a file `output2.xml` by:

```
> transform.sh -i input2.xml -o output2.xml
```

□

9.2 Rule-based Transformations

In the rule-based approach, a transformation is specified by a set of rules. Each rule consists of a *match pattern* (the *where*) identifying subtrees from the input to which the rule is potentially applicable and a corresponding action specifying how to transform these subtrees (the *what*). An action constructs a piece of XML content – typically by using parts of the matching subtree and by selecting further subtrees from the input and recursively applying the transformation rules on them. We use the term *current tree* to denote the subtree on which a rule is applied.

The result of a transformation is given by executing the action associated with the first match pattern in the specification fulfilled by the root of the input document.

It is worth emphasizing that the applicable rules for a subtree do not depend on the rule in which the subtree was selected for recursive processing, but only on the match-pattern fulfilled by the subtree in the global context. Consequently, the applicable rule for a subtree is determined solely by the match patterns fulfilled by the subtree in the input tree. Correspondingly, finding out which are the applicable rules can be done before actually performing the transformation by evaluating the match patterns on the input tree as depicted in Figure 9.3.

We may therefore think of such a transformation as consisting of two phases (see Figure 9.3). In the first phase, the pattern matching phase, a pattern matcher annotates each node of the input tree with the fulfilled matched patterns. In the second phase, the transformation phase, the annotations are used as a guide to what actions are to be taken when the subtrees are to be transformed.

Fxt uses as match patterns the Fxgrep patterns introduced in Section 3.

Example 9.3: As a simple example of a rule-based transformation, consider the following Fxt transformation, which given an XML document, produces an HTML list of titles of the sections in the document. Note that the specification of the XML transformation to be generated by Fxt is itself given as an XML document.

Fxt Example 1

```
<fxt:spec>
  <fxt:pat>/*</fxt:pat>
    <ul>
      <fxt:apply/>
    </ul>

  <fxt:pat>//section/title/""</fxt:pat>
    <li>
      <fxt:current/>
    </li>

  <fxt:pat>default</fxt:pat>
    <fxt:apply/>
</fxt:spec>
```

Here, the elements `fxt:pat` contain the match patterns of the rules, whereas the action parts span from their triggering pattern to the following `fxt:pat` element (or the end). In the given example, the topmost element of every document to be transformed matches the first pattern, i.e., `/*`. The corresponding action specifies that the result must be an element of type `ul` (which stands for "unordered list"), the content of which is given by recursively transforming the content of the topmost element. The second rule says, that whenever text is found inside the `title` element of a `section`, a new `li` (which stands for "list item") element should be created the content of which is the matched text. The rule for the *default pattern* says that otherwise, the transformation should simply proceed to the subtrees in the content of the current subtree. □

9.3 Fxt Basic Constructs

Many usual transformations can be achieved by using a small number of primitive constructs that we introduce in this section.

9.3.1 Literal XML Output

Choosing an XML syntax for specifications has the advantage that XML elements which are to appear literally in the output, as well as the nested structure of the XML output to be produced, are directly reflected in the Fxt specification.

For that matter, anything appearing in the action part of a rule which is not an element prefixed by `fxt:` is to be exactly reproduced in the output of the transformation. Elements prefixed by `fxt:` are reserved for specifying other specific Fxt actions.

Example 9.4: Consider the following transformation:

Fxt Example 2

```
<fxt:spec>
  <fxt:pat>/*</fxt:pat>
    <html>
      <head>
        <title>Example</title>
      </head>
      <body>
      </body>
    </html>
</fxt:spec>
```

The generated transformation produces for any XML document (whose root is matched by the pattern `/*`) an empty HTML document as below:

XML Example 14

```
    <html>
      <head>
        <title>Example</title>
      </head>
      <body>
      </body>
    </html>
```

□

The previous example transformation is not especially useful, as the input does not influence how output is produced. For a reasonable transformation we typically will need to copy portions of the input document or recursively apply the transformation rules on subtrees of the input document.

9.3.2 The Default Action

If a default pattern is not specified, an action is added by default. The default action is to be performed when the transformation is applied on a subtree that does not match any specified pattern. The default action produces a subtree that has the root of the matched subtree as its root and the result of applying the transformation on the content of the matched subtree as its content. This ensures that a node from the input which conforms to a match pattern of a rule is processed by the action of that rule (even) if its ancestors do not conform to any match pattern.

Example 9.5: The following specification uses an explicit action and the default action to generate a transformation where all `italic` elements are replaced by `i` elements containing the text “Hello!”:

Fxt Example 3

```
<fxt:spec>
  <fxt:pat>//italic</fxt:pat>
    <i>Hello!</i>
</fxt:spec>
```

For the following XML document:

XML Example 15

```
<doc>
```

```
  <italic>Some text written in italic</italic>
  <bold>
    <italic>Some text written in bold-italic</italic>
  </bold>
</doc>
```

the generated transformation produces:

XML Example 16

```
<doc>
  <i>Hello!</i>
  <bold>
    <i>Hello!</i>
  </bold>
</doc>
```

□

9.3.3 Explicitly Applying the Transformation Rules

One of the fundamental constructs of rule-based transformation is recursive transformation. Recursive application of the transformation allows one to specify within a rule that the output to be produced when processing a node by that rule is obtained by applying the transformation rules on a set of nodes selected in the context of the processed node.

This is achieved in Fxt by the `fxt:apply` element. `fxt:apply` produces a sequence of subtrees by concatenating the results of recursively applying the transformation on the children of the current subtree.

Example 9.6: The transformation specified in the previous subsection replaced `italic` elements by `i` elements with some fixed content. Now we want to have the `italic` elements replaced by `i` and `bold` elements replaced by `b`. Rather than some fixed content, we want the content of the `i` and `b` elements to be obtained by applying the transformation rules on the content of the corresponding `italic` and `bold` elements respectively. This is specified as follows:

Fxt Example 4

```
<fxt:spec>
  <fxt:pat>//italic</fxt:pat>
    <i><fxt:apply/></i>

  <fxt:pat>//bold</fxt:pat>
    <b><fxt:apply/></b>
</fxt:spec>
```

The same input file in XML Example 15 is now transformed to:

XML Example 17

```
<doc>
  <i>Some text written in italic</i>
  <b>
    <i>Some text written in bold-italic</i>
  </b>
</doc>
```

The transformation starts at the root element `doc`. As `doc` does not match any of the specified match patterns, the default rule produces a `doc` element, the content of which is obtained by applying the transformation rules on its content. Transforming the first `italic` child which is matched in the original input by the pattern `//italic` produces an `i` element, the content of which is obtained by applying the transformation rule on the content of the `italic` node, i.e. on the text node "`Some text written in italic`". This text node does not match any match pattern and thus the result of transforming it by the default rule is a copy of it. The `bold` child of `doc` matches the pattern `//bold` and is thus transformed to a `b`, the content of which is given by applying the transformation rules on the content of the `bold` element. The content of the `bold` element is an `italic` element which is transformed similarly to the first `italic` element. □

Note again that the way a node from the input is transformed by the transformation rules depends only on the match pattern that this node matches when considered in the input, i.e., does not depend on the location where the node was selected for recursive transformation.

9.3.4 Explicitly Selecting Nodes for Recursive Transformation

The set of nodes on which a transformation is to be recursively applied as specified by an element `fxt:apply` is by default formed from the children of the current element. In general, however, it is possible to explicitly select the set of nodes by specifying a *select pattern*. The set of nodes selected is the set of nodes located by the pattern in the current subtree. The selected subtrees are processed in the order in which they appear in the input. A select pattern can be specified via a `select` attribute of the `fxt:apply` element. Select patterns can be arbitrary Fxgrep patterns.

Example 9.7: Consider the following transformation:

Fxt Example 5

```
<fxt:spec>
  <fxt:pat>//section</fxt:pat>
    <ol>
      <fxt:apply select="//example"/>
    </ol>

  <fxt:pat>//example</fxt:pat>
    <li>
      <fxt:apply/>
    </li>
</fxt:spec>
```

This transformation produces for each section an HTML ordered list of the examples occurring in that section. The `fxt:apply` with `select` attribute in the first rule tells that when a `section` element is processed, an `ol` (which stands for "ordered list") element must be produced. The content of the `ol` element is obtained by applying the transformation on the `example` descendants of the current `section` element. When such a descendant is transformed, as specified by the second rule, an `li` element is to be produced. The content of the `li` element is given by recursively transforming the children of the `example` element.
□

It is worth noting that, while *match* patterns are evaluated once and for all in the input tree, select patterns are to be evaluated in the current subtree.

9.3.5 Copying Nodes from the Input

Observe that, by the default rule, when a node is transformed and none of its descendants are matched by some match pattern, the outcome in the output is a copy of the node. This is the case for instance in Fxt Example 5 with `example` elements (assuming that examples themselves do not contain sections or other examples). Example elements could thus be directly copied rather than recursively transformed as specified by `fxt:apply`. The content of an element can be copied by using the element `fxt:copyContent` as below:

Example 9.8:

Fxt Example 6

```
<fxt:spec>
  <fxt:pat>//section</fxt:pat>
    <ol>
      <fxt:apply select="//example"/>
    </ol>

  <fxt:pat>//example</fxt:pat>
    <li>
      <fxt:copyContent/>
    </li>
</fxt:spec>
```

□

By default, `fxt:copyContent` copies the children of the current node. Similarly as with `fxt:apply`, one can explicitly specify other descendants to be copied by providing a select pattern as the value of the attribute `select`.

Example 9.9: The following transformation produces for any `section` element in the input the collection of all `example`-elements in the section grouped under an element `examples`:

Fxt Example 7

```
<fxt:spec>
  <fxt:pat>//section</fxt:pat>
    <examples>
      <fxt:copyContent select="//example"/>
    </examples>
</fxt:spec>
```

□

9.3.6 Decomposing and Building up

XML input content can be decomposed into its parts, e.g., an element into the element tag and element attributes, text content, and these parts can be independently used to build up the output.

When the current node is an element, `fxt:copyType` for example produces an element with the same tag as the current element and the content of which is given by the content of the `fxt:copyType` element.

Example 9.10: The following specification generates a transformation which embeds all elements having an attribute `importance` with a value of "great" in `bold` elements:

Fxt Example 8

```
<fxt:spec>
  <fxt:pat>//*[@importance="great"]</fxt:pat>
    <bold>
      <fxt:copyType>
        <fxt:apply/>
      </fxt:copyType>
    </bold>
</fxt:spec>
```

The following input:

XML Example 18

```
<manual>
  <ch>Unpacking</ch>
  <ch importance="great">Before you begin</ch>
  <ch>Troubleshooting</ch>
</manual>
```

is transformed by the generated transformation to:

XML Example 19

```
<manual>
  <ch>Unpacking</ch>
  <bold>
      <ch>Before you begin</ch>
  </bold>
  <ch>Troubleshooting</ch>
</manual>
```

□

An action `fxt:copyAttributes` copies the attribute of the current element into the set of attributes of the surrounding element in the output document. `fxt:addAttribute` adds an attribute to the set of attributes of the surrounding element in the output.

Example 9.11: The transformation:

Fxt Example 9

```
<fxt:spec>
  <fxt:pat>//a/b</fxt:pat>
     <B>
      <fxt:copyAttributes/>
      <fxt:addAttribute name="sonOf" val="a"/>
     </B>
</fxt:spec>
```

replaces `b` elements having an `a` father by `B` elements having the same set of attributes and a supplementary attribute `sonOf` with value "`a`". □

In addition to adding an attribute, it is possible to remove an attribute or change its value. Furthermore, a set of useful shortcuts are provided for groups of basic constructs which are frequently combined in the same way, as for example:

- ✧ `fxt:current` to get an exact copy of the current subtree;
- ✧ `fxt:copyTag` to produce an element with the same tag and the same attributes as the current element;
- ✧ `fxt:copyTagApply` to produce an element with the same tag, the same attributes as the current element and a content obtained as the concatenation of the subtrees resulting from recursively applying the transformation on the content of the current element.

9.4 Interfacing with the SML Programming Language

Many of the standard transformation tasks should be expressible by the constructs as introduced so far. However, not all potential needs, especially of more elaborated XML processing, are foreseeable. In order to meet them as well, we embedded into Fxt an escape mechanism into the full programming language SML. In order to do so in a clean way, an interface is needed which abstracts from the implementation details of documents.

As mentioned in Chapter 7, there are two main types of interfaces: tree-based and event-based. While event-based interfaces are well-suited for one-pass applications, tree-based interfaces also support applications that need multiple passes over the input, as generally needed for XML transformations. In the tree-based case, an abstract data type is specified. The most commonly used is the Document Object Model (DOM) [DOM98]. Though claiming to be designed for any programming language, the DOM is committed to the object-oriented paradigm: it defines class interfaces for accessing XML documents. Also, it views the document tree as a graph within which arbitrary navigation is possible. Different interfaces are needed for XML processing in functional programming languages. We briefly present one in the next section.

9.4.1 The Functional Document Model

For SML and its specific use with Fxt, we provide a programming interface called FDM (Functional Document Model).

One practical problem of SML here is that XML documents may contain any legal Unicode character [Con96], while SML supports only 8-bit characters and has no notion of Unicode. Therefore a Unicode library is provided inside FDM declaring types for the Unicode characters and strings, along with basic functions for manipulating them, as well as conversion functions from and to SML strings.

The types Tree and Forest are provided as abstractions of XML trees and forests, respectively. Functions are provided for testing the type or content of a node, for accessing its constitutive parts and for constructing different node types. Basic functionality is supplied for transforming forests, e.g., for mapping, successive composition (folding), filtering, sorting or outputting trees or forests. The functional concept of higher-order functions makes it possible to elegantly obtain complex processing from combining basic functions provided by the FDM.

The FDM is completely presented in Appendix B. An excerpt from the FDM interface is presented below:

Example 9.4.1:

```
signature FDM =
  sig
    type Tree
    type Forest =  Tree vector
    ...

    (* testing type and content *)
    val isElement      : Tree -> bool
    val isText         : Tree -> bool
    val hasElementType : Unicode.Vector -> Tree -> bool
    val hasTextContent : Tree -> bool
    val hasAttribute   : Unicode.Vector -> Tree -> bool
    ...

    (* constructing node types *)
    val element        : Unicode.Vector -> Attribute list
                           -> Tree vector -> Tree
    val text           : Unicode.Vector -> Tree
    ...

    (* accessing constitutive parts *)
    val getElementType : Tree -> Unicode.Vector
    val getTextContent : Tree -> Unicode.Vector
    val getAttribute   : Unicode.Vector -> Tree
                           -> Unicode.Vector
    ...

    (* transforming forests *)
    val map            : (Tree -> 'a) -> Forest -> 'a vector
    val foldl          : (Tree * 'a -> 'a) -> 'a -> Forest
                           -> 'a
    val deleteAll      : (Tree -> bool) -> Forest -> Forest
    val deleteFirstN   : int -> (Tree -> bool) -> Forest
                           -> int * Forest
    val filterFirst    : (Tree -> bool) -> Forest -> Tree
    ...

    (* outputting *)
    val putTree        : Tree -> string -> string option
                           -> unit
    val putForest      : Forest -> string -> string option
                           -> unit

  end
```

The names of the FDM functions are mostly self-explanatory. `filterFirst`, for example, is a function which expects a predicate over trees as its first argument and a forest as the second argument. It returns the first tree in the forest satisfying the predicate. Consider the call `filterFirst hasTextContent` where `hasTextContent` tests whether a node has plain text content. Then a function is returned which takes a forest and returns the first tree in the forest having only text content.

In the call `filterFirst (hasElementType (String2Vector "alfa"))`, `(String2Vector "alfa")` returns the Unicode string `alfa`. The application of `hasElementType` to this Unicode string returns a predicate testing whether a tree has the specified type. The application of `filterFirst` returns thus a function which takes a forest and returns the first tree in the forest having element type `alfa`.

9.4.2 Embedding SML Code into Transformations

SML code is embedded into an Fxt specification via special attributes of Fxt actions. The values for these attributes are SML expressions. Their evaluation can provide XML content to be used in the output, predicates for filtering XML forests, or even functions for application onto document components. Access to the input is provided via the reserved name `current` which provides a handle to the node being currently transformed.

9.4.2.1 Computing Names

Rather than only fixed names or names available in the input, Fxt transformations offer the possibility of producing names with computed content. The computed content can be specified in general as an SML expression.

Example 9.12: The following transformation replaces every element name containing more than six characters by its first six characters:

Fxt Example 10

```
<fxt:spec>
  <fxt:pat>//*</fxt:pat>
    <fxt:tag
        fxt:name='
          let
            val name = getElementType current
          in
            if Vector.length name > 6 then Vector.extract (name,0,SOME 5)
            else name
          end'>
      <fxt:apply/>
    </fxt:tag>
</fxt:spec>
```

The `fxt:tag` outputs an element whose name is given as the value of an attribute `fxt:name`, which must be an SML expression evaluating to a Unicode string. The reserved name `current` always refers to the current subtree. Thus, `getElementType current` returns the name of the current element as a vector of characters. The content of the output element is given by the content of the `fxt:tag` element, which, in this case, applies the transformation on the children nodes. □

Furthermore, it is possible to output processing instructions with some computed names for the processor or the data parts, as well as text nodes with some computed value.

9.4.2.2 Conditional Processing

It is sometimes desirable that a rule produces some output only when the current node fulfills some condition which is not expressible by the match pattern. For this purpose, Fxt provides an `fxt:if` element which can be used if output is to be produced only when a condition is satisfied. The condition must be specified as the value of an attribute `test` and must evaluate to an SML boolean.

Example 9.13: A most superstitious user might want to avoid elements with a prime number of children by adding one extra `dummy` child. Then she could use, e.g., the following transformation:

Fxt Example 11

```
<fxt:spec>
  <fxt:pat>//*</fxt:pat>
    <fxt:copyTag>
      <fxt:apply/>
      <fxt:if
         test='
           let fun prime i = if i=0 then false
                             else List.all (fn j => i mod j >0)
                                  (List.tabulate (i div 2,fn x => x+2))
           in prime (Vector.length (sons current)) end'>
      <dummy/>
      </fxt:if>
    </fxt:copyTag>
</fxt:spec>
```

The boolean SML expression given as the value of the `test` attribute defines the auxiliary function prime, which is then used to check whether the number of children of the current element (`Vector.length (sons current)`) is prime.
□

9.5 Storing Intermediate Results: Variables

In order to access information obtained earlier during the transformation, it would be nice to have the possibility of simply storing certain data for later use. Therefore, we decided to include a notion of global variables into Fxt .

Example 9.14: As a simple example of the use of Fxt variables, consider an input document in which consecutive repetitions of the same XML content are avoided by using a special place-holder denoting previously defined content. Now, a transformation should process the XML input and replace the definition as well as the place-holder elements by the result of processing this XML content according to some specified rules. Suppose the XML content whose repetition is avoided is enclosed in an `def` element whereas the place-holder is the `lastdef` element.

Fxt remembers the result of processing the last `def` element in a variable with name "`res`". This value then just has to be looked up when transforming subsequent `lastdef` elements. A variable is declared using the `fxt:global` element before the first rule. The type of the variable (here: `Forest`) must be declared too. The result of outputting the value of a variable depends on its type.

Fxt Example 12

```
<fxt:spec>
  <fxt:global name="res" type="Forest"/>
  <fxt:push   name="res" val="emptyForest"/>

  <fxt:pat>//def</fxt:pat>
    <fxt:setForest name="res">
      <DEF><fxt:apply/></DEF>
    </fxt:setForest>
    <fxt:get name='res'/>

  <fxt:pat>//lastdef</fxt:pat>
    <fxt:get name='res'/>

</fxt:spec>
```

The variable is initialized before the transformation begins via the `fxt:push`[2] after its declaration. The value `emptyForest` is pre-defined in FDM for empty XML content. Every time a `def` element is processed, the value of the `def` variable is set to an element `DEF` whose content is obtained by recursively processing the content of the `def` element, as specified by the `fxt:setForest` element. The last stored value is output in the second rule via the `fxt:get` element which inserts the variable with the specified name in the result. Note that the transformation produces the intended result only under the assumption that `def` elements do not occur inside another `def` element. □

Nested Scopes Processing hierarchically nested elements incurs the need for introducing scopes for variables. One way of doing so is to allow local variables which are visible just during processing a specific XML element. Re-structuring transformations, however, ask for more flexible scoping rules. Therefore, we decided to organize every variable as a *stack* — meaning that we support push and pop operations on variables. A push operation introduces a new scope whereas a pop leaves this scope again. So, the single `fxt:push` element in the example above creates in fact a new scope of the variable `res`.

Besides for pushing and popping, Fxt actions are provided for setting or outputting the values of the topmost elements.

Example 9.15: The following specification generates a transformation which, given an XML document containing nested `li` elements (list items), adds before every `li` an integer representing the number of the list item on its nesting level:

Fxt Example 13

```
<fxt:spec>
  <fxt:global name="i" type="int"/>
  <fxt:push   name="i" val="1"/>
```

[2]The choice of the name `fxt:push` is shortly explained.

```
  <fxt:pat>//li</fxt:pat>
    <fxt:get name="i"/>:
      <li>
        <fxt:inc name="i"/>
        <fxt:push name="i" val="1"/>
        <fxt:apply/>
        <fxt:pop name="i"/>
      </li>

</fxt:spec>
```

The global variable `i` is declared to be of type `int`. Whenever an `li` element is transformed, the last value of `i` is output using `fxt:get`, and the `li` tag is copied to the output. Before proceeding with the transformation to the sons, the current value of `i` is incremented, and a new scope for the variable `i` is introduced by means of the `fxt:push` element. The new instance of the variable `i` which is initialized with the value `1`. After processing the contents of the `li` element, the last instance of `i` is popped off the stack using `fxt:pop`. The following input:

XML Example 20

```
<doc>
  <li>
    <li>
      <li>a</li>
      <li>b</li>
    </li>
    <li>
      <li>e</li>
    </li>
  </li>
</doc>
```

is transformed to:

XML Example 21

```
<doc>
  1:<li>
    1:<li>
      1:<li>a</li>
      2:<li>b</li>
    </li>
    2:<li>
      1:<li>e</li>
    </li>
  </li>
</doc>
```

□

Special actions are provided for the convenient use of `Tree` and `Forest` variables. For example, a `Forest` value can be constructed and pushed or set, using the actions `fxt:pushForest` or `fxt:setForest`, respectively. The `fxt:pushForest` and `fxt:setForest` elements must have an attribute `name` specifying the name of the global on top of which the forest value is to be pushed or set. The forest value is specified as the content of the elements `fxt:pushForest` or `fxt:setForest` which can consist of any sequence of Fxt actions.

Example 9.16: The following is the specification of a transformation which adds the list of links to the anchors within an XHTML document to the end of it:

Fxt Example 14

```
<fxt:spec>
  <fxt:global name="index" type="Forest"/>

  <fxt:pat>xhtml</fxt:pat>
    <fxt:push name="index" val="emptyForest"/>
    <fxt:copyTagApply/>
    <fxt:pop name="index"/>

  <fxt:pat>//a[@name]</fxt:pat>
    <fxt:setForest name="index">
      <fxt:get name="index"/>
      <a>
        <fxt:attribute name='name'/>
        <fxt:addAttribute name='href'
           valExp='concatVectors (String2Vector "#",
                   (getAttribute (String2Vector "name") current))'/>
      </a>
    </fxt:setForest>
    <fxt:copyTagApply/>

  <fxt:pat>//index</fxt:pat>
     <p>Index:</p>
     <p><fxt:get name='index'/></p>

</fxt:spec>
```

The following input document:

XML Example 22

```
<xhtml>
   <p>Text</p><a name="a1"/>
   <p>Text</p><a name="a2"/>
   <p>Text</p><a name="a3"/>
   <p>That was it.</p>
   <index/>
</xhtml>
```

is transformed to:

XML Example 23

```
<xhtml>
   <p>Text</p><a name='a1'></a>
   <p>Text</p><a name='a2'></a>
   <p>Text</p><a name='a3'></a>
   <p>That was it.</p>
   <p>Index:</p>
   <p><a href='#a1'>a1</a><a href='#a2'>a2</a><a href='#a3'>a3</a></p>
</xhtml>
```

The `Forest` variable `index` functions as an accumulating parameter in which links are collected when encountering anchors in the input document. □

9.6 Grouping

An important task in transformations is grouping and processing together elements having a certain common property.

Example 9.17: Consider the following XML document containing a list of books.

XML Example 24

```
<books>
  <book>
    <author>Stefan Zweig</author>
    <title>Maria Stuart</title>
  </book>
  <book>
    <author>Giuseppe Tomasi di Lampedusa</author>
    <title>Il Gatopardo</title>
  </book>
  <book>
    <author>Franz Kafka</author>
    <title>Amerika</title>
  </book>
  <book>
    <author>Stefan Zweig</author>
    <title>Sternstunden der Menschheit</title>
  </book>
  <book>
    <author>Franz Kafka</author>
    <title>Tagebücher</title>
  </book>
</books>
```

A classic task is to group together books written by the same author as below:

XML Example 25

```
<authors>
  <author>
    <name>Stefan Zweig</name>
    <title>Maria Stuart</title>
    <title>Sternstunden der Menschheit</title>
  </author>
  <author>
    <name>Giuseppe Tomasi di Lampedusa</name>
    <title>Il Gattopardo</title>
  </author>
  <author>
    <name>Franz Kafka</name>
    <title>Amerika</title>
    <title>Tagebücher</title>
  </author>
</authors>
```

□

The common property of the elements to be grouped is called *key*. Given an element, a key can be seen as an identifier telling to which group the element belongs. In the example above, given a `book` element its key is the text content of its `author` sub-element, i.e. the author's name. In general a key might be an arbitrary Unicode string.

Generally speaking, grouping is a partial mapping from a key to the set of nodes having this key in common. Correspondingly, this mapping can be stored as a table indexed by keys and whose entries are sets of nodes grouped together. In the following sections we call the partial mapping a *keyed table*. A key identifies the nodes belonging to the same group as the entry for the key.

9.6.1 Defining a Keyed Table

To define a keyed table one must specify which are the nodes to be stored in the table and, given such a node, what is its key. A keyed table is specified using an empty element `fxt:groupBy` before the first rule. The nodes of interest are specified using an Fxgrep pattern as the value of a `group` attribute. The key of a given node can be specified via a *select pattern* given as the value of an attribute `by`. The key is obtained as the text content of the nodes retrieved by the select pattern in the context of the given node. Alternatively, if an attribute `byExp` is present, it must specify an SML function which is to be applied on the node to obtain its key. As there may be more than one keyed table, an attribute `in` is used to identify the currently defined table.

Example 9.18: To group the books in XML Example 24 by their authors we write:

Fxt Example 15

```
<fxt:groupBy group="//book" by="author" in="booksByAuthor"/>
```

We group the `book` elements as specified by the pattern `//book`. A `book` element belongs to the group specified by the name of its author, obtained by evaluating the select pattern `author` in the context of the `book` element. □

9.6.2 Using a Keyed Table

Having defined a keyed table we are able to process every entry of it separately. Typically, there are two usage patterns of keyed tables as presented in the next two subsections. Firstly we want to be able to process all the entries one after another, when we want to handle each group of nodes sharing a common property in turn. An instance of this usage scenario is the transformation addressed in Example 9.17. Secondly we might be interested only in an entry for a given key, when we want to handle only nodes sharing a given common property.

9.6.2.1 Iterating over Table Entries

To iterate over the entries of a keyed table `fxt:forAllKeys` may be used. An attribute `in` specifies the identifier of the table as specified in its definition in the corresponding `fxt:groupBy` element. The content produced for each entry is specified by the content of the `fxt:forAllKeys` element, which may contain any Fxt action. An element `fxt:getKey` can be used to output the current key. The entire current entry can be output using `fxt:copyKey`. To recursively process the nodes in the current entry using the transformation rules, `fxt:applyKey` can be used.

Example 9.19: The solution to the transformation presented in Example 9.17 is presented in Fxt Example 16.

Fxt Example 16

```
<fxt:spec>
  <fxt:groupBy group="//book" by="author" in="booksByAuthor"/>

  <fxt:pat>/*</fxt:pat>
<authors>
  <fxt:forAllKeys in="booksByAuthor">
    <author>
      <name><fxt:getKey/></name>
      <fxt:applyKey/>
    </author>
  </fxt:forAllKeys>
</authors>

  <fxt:pat>//book<fxt:pat>
       <fxt:copyContent select="title"/>

</fxt:spec>
```

The keyed table is declared as presented in Example 9.18. The result is an `authors` element enclosing one `author` element for each entry in the table. The name of the author is the current key, output by `fxt:getKey`. The entry contains the `book` nodes in the input document written by the current author. These are recursively transformed by using the second rule which simply outputs their titles in order to produce the desired output. □

9.6.2.2 Selecting One Entry

Rather than being interested in processing all groups, we are sometimes interested only in a group with a certain property. The group of interest is selected via its key, which in general depends on information from the input document made available while evaluating a transformation rule on a current node. One can process a single entry in a table either by copying it with an `fxt:copyContent` element, or, more generally, by recursively applying the transformation rules on it using an `fxt:apply` element. The name of the table has to be specified as the value of an attribute `in`. The key is obtained by default as the text content of the current node. Alternatively, the key can be explicitly indicated either literally as the value of a `by` attribute or as an SML function taking the current node and producing the key.

Example 9.20: Consider the modified XML input in XML Example 24 where a list of the best authors has been added:

XML Example 26

```
<books>
  <book>
    <author>Stefan Zweig</author>
    <title>Maria Stuart</title>
  </book>
  <book>
    <author>Giuseppe Tomasi di Lampedusa</author>
    <title>Il Gattopardo</title>
  </book>
  <book>
    <author>Franz Kafka</author>
    <title>Amerika</title>
  </book>
  <book>
    <author>Stefan Zweig</author>
    <title>Sternstunden der Menschheit</title>
  </book>
  <book>
    <author>Franz Kafka</author>
    <title>Tagebücher</title>
  </book>

  <bestAuthors>
    <author>Giuseppe Tomasi di Lampedusa</author>
    <author>Franz Kafka</author>
  </bestAuthors>
</books>
```

We want to replace the `bestAuthors` element with a `bestBooks` containing the books written by the best authors as presented in XML Example 27:

XML Example 27

```
<books>
  <book>
    <author>Stefan Zweig</author>
    <title>Maria Stuart</title>
  </book>
  <book>
    <author>Giuseppe Tomasi di Lampedusa</author>
    <title>Il Gattopardo</title>
  </book>
  <book>
    <author>Franz Kafka</author>
    <title>Amerika</title>
  </book>
  <book>
    <author>Stefan Zweig</author>
    <title>Sternstunden der Menschheit</title>
  </book>
  <book>
    <author>Franz Kafka</author>
    <title>Tagebücher</title>
  </book>

  <bestBooks>
    <book>
      <author>Giuseppe Tomasi di Lampedusa</author>
      <title>Il Gattopardo</title>
    </book>
    <book>
      <author>Franz Kafka</author>
      <title>Amerika</title>
    </book>
    <book>
      <author>Franz Kafka</author>
      <title>Tagebücher</title>
    </book>
  </bestBooks>
</books>
```

The solution in Fxt is presented in Fxt Example 17. We again use a keyed table grouping `book` elements by their authors, as specified by the `fxt:groupBy` element. The first rule produces the `bestBooks` element by recursively applying the rules on the `author` sub-elements. These are processed by the second rule which uses the `fxt:copyContent` element to retrieve the entry stored under the key given by the author name obtained as the content of the current element.

Fxt Example 17

```
<fxt:spec>
  <fxt:groupBy group="//book" by="author" in="booksByAuthor"/>

  <fxt:pat>//bestAuthors<fxt:pat>
    <bestBooks>
      <fxt:apply/>
    </bestBooks>

  <fxt:pat>//bestAuhtors/author<fxt:pat>
       <fxt:copyContent in="booksByAuthor"/>

</fxt:spec>
```

□

9.6.3 Handling Cross References in XML Documents

In practice, not all the information which we would like to represent is strictly hierarchically structured. Consider for example the XML representation of an article in which references from one section are to be permitted to arbitrary sections. Therefore, the XML specification [XML98a, XML04] provides a way in which supplementary connections can be specified among the nodes in an XML document, allowing the representation of graph- rather than just tree-structured data.

For this purpose the DTD of an XML specifies a way of uniquely identifying and locating elements, via so-called *ID* and *IDREF* attribute types in elements. The uniqueness constraints and the validity of references can be verified by a validating parser.

Transformers must be correspondingly able to use these possibly additionally available cross references in XML documents. This can be handled in Fxt via keyed tables, as these are a generalization of the reference mechanism. Identifiers of nodes correspond to our keys, which can be located anywhere inside an element, rather than just as a value of some `ID` attribute. Similarly, a reference may be located anywhere, rather than just as a value of some `IDREF` attribute. Also, rather than just one node, a key may identify a group of nodes, allowing thereby the specification of one-to-many relations.

Example 9.21: As an example consider the following XML input containing a description of a recommended route.

XML Example 28

```
<routePlanner>
  <location id="1">München</location>
  <location id="2">Berlin</location>
  <location id="3">Köln</location>
  <location id="4">Bonn</location>
  <location id="5">Trier</location>

  <route>
    <step from="1" to="2">car</step>
    <step from="2" to="3">plane</step>
    <step from="3" to="4">train</step>
    <step from="4" to="5">bike</step>
    <step from="5" to="1">train</step>
  </route>
</routePlanner>
```

Suppose that we want to present the route in the following form:

XML Example 29

```
    From München to Berlin by car
    From Berlin to Köln by plane
    From Köln to Bonn by train
    From Bonn to Trier by bike
    From Trier to München by train
```

The Fxt transformation to achieve the above is presented below:

Fxt Example 18

```
<fxt:spec>
  <fxt:groupBy group="//location" byAtt="id" in="locationById"/>

  <fxt:pat>//step</fxt:pat>
    From <fxt:copyKey in="locationById" byAtt="from"/>
    to <fxt:copyKey in="locationById" byAtt="to"/>
    by <fxt:copyContent/>

   <fxt:pat>default</fxt:pat>
     <fxt:apply/>
</fxt:spec>
```

The mapping from locations to their identifiers is stored in the table `locationById` as declared by `fxt:groupBy`. The textual representation of a step is obtained in the first rule by outputting the entries keyed by the `from` and `to` attributes via `fxt:copyKey`. The second rule ensures that no other content is produced in the output.

□

9.7 Summary

This chapter has introduced the basic capabilities of Fxt via examples. For a complete reference to the constructs which have been presented, we refer to the user guide of Fxt [3]. The Fxt transformation language introduced so far, only makes use of the unary queries expressible by Fxgrep. This does not yet exploit all querying capabilities presented in the first part of this work, where we have seen that binary queries can be efficiently implemented. In the next chapter we present how binary queries can be used to increase the expressiveness and efficiency of rule-based transformations.

[3]Might be consulted online at `http://www2.informatik.tu-muenchen.de/~berlea/Fxt`

Chapter 10

Binary Queries in Transformations

The use of unary queries in rule-based transformations as presented in the previous chapter has a couple of problematic aspects. In this chapter we address them and show how they can be solved by using binary queries.

10.1 Limitations of Unary Queries

As presented so far, in Fxt nodes selected via a select pattern can be situated only below the current node, since Fxgrep unary patterns only locate nodes below the node in the context of which they are evaluated.

On the one side, this restriction ensures that transformations will terminate on any input. Also, one might argue that such uni-directional transformations are easier to comprehend than those performing a random walk over the tree.

On the other side, the select patterns have two inherent problems that we address in the rest of this section. The first problem is that nodes may need to be selected not only for the purpose of recursive processing but also for copying them into the output. Only allowing one to copy nodes which are below the current node may be too restrictive.

Example 10.1: Consider for example a document having `department` elements which contain an `id` element followed by a sequence of `employee` elements.

XML Example 30

```
<company>
  <department>
    <id>Public Relations</id>
    <employee>Jan Smith</employee>
    <employee>Meg Rush</employee>
  </department>
  <department>
    <id>Sales</id>
    <employee>David Hughes</employee>
    <employee>Angela Dimm</employee>
  </department>
</company>
```

Suppose we want to produce a list of employees, each of them containing the `id` of his department as below:

XML Example 31

```
<employee><dept>Public Relations</dept>Jan Smith</employee>
<employee><dept>Public Relations</dept>Meg Rush</employee>
<employee><dept>Sales</dept>Don Hughes</employee>
<employee><dept>Sales</dept>Angela Dimm</employee>
```

One straightforward solution would be to write a rule for processing `employee` elements in which the content of the `id` left sibling of the current element is selected. However, Fxt's select patterns introduced so far cannot refer to nodes other than below the current node. □

The variable mechanism introduced in Section 9.5 can be used as a workaround for this problem. Generally speaking, the nodes of interest can be stored in a variable in advance by selecting them at an ancestor node, rather than at the node where they are needed. All nodes are selectable in this way as they are all descendants of an ancestor of the current node (in the worst case they are descendants of the root node). The idea is exemplified below.

Example 10.2: The processing task described in Example 10.1 is achieved by the following Fxt transformation:

Fxt Example 19

```
<fxt:spec>
  <fxt:global name="deptName" type="Forest"/>
  <fxt:push name="deptName" val="emptyForest"/>

  <fxt:pat>//department</fxt:pat>
     <fxt:pushForest name="deptName">
       <fxt:copyContent select="id"/>
     </fxt:pushForest>
     <fxt:apply select="employee"/>
     <fxt:pop name="deptName"/>

  <fxt:pat>//department/employee</fxt:pat>
     <employee>
       <dept><fxt:get name="deptName"/></dept>
       <fxt:copyContent/>
     </employee>

  <fxt:pat>default</fxt:pat>
     <fxt:apply/>

</fxt:spec>
```

The `fxt:global` element declares a variable `deptName` which can store any XML fragment as an FDM forest. The value is remembered at department nodes as the content of the `id` sub-element in the first rule. The subsequent recursive processing of the `employee` child elements activated by the `fxt:apply` element can access the value of the variable, as specified in the second rule by the `fxt:get` element. The value is thrown away by `fxt:pop` on leaving the `department` node. □

Even though the presented workaround may always be used, one might argue that this workaround affects the intended declarativeness of rule-based transformation languages. Therefore, we are going to lift the limitation of Fxt select patterns s.t. that the selected nodes may be situated anywhere in the document, rather than only bellow the current node.

A second problematic aspect of select patterns is how to efficiently evaluate them. Every pattern evaluation (obviously) requires processing the tree in which the pattern is to be evaluated. Match patterns, as unary Fxgrep patterns, which are to be evaluated in the context of the root element of the input tree, require (as seen in Section 5.2) at most two traversals of the input. In contrast, a select pattern is to be evaluated on the subtree on which the rule containing the select pattern is currently applied. Its evaluation requires, at least in principle, to traverse this current sub-tree. As the number of rule applications is not fixed, this implies that some nodes may be visited for the purpose of pattern evaluation an unbounded (i.e. not a priori fixed) number of times. Navigation in the input tree, as necessary for pattern evaluation is nevertheless time-costly and should be reduced to a minimum in order to obtain efficient implementations of XML transformations. As we present next, by using binary patterns it is possible to ensure that two traversals of the input tree are enough for the evaluation of all (match and select) patterns in Fxt transformations.

10.2 Selecting via Binary Queries

The match pattern of a rule together with a select pattern occurring in the rule can be thought of as specifying a binary relation between nodes. The relation pairs a node fulfilling the match pattern with each of the nodes to be selected by the select pattern in the context of this node. Our key idea is that, rather than using a match and a select pattern, one could as well use a *binary query* to specify this binary relation.

As in the case of standalone Fxgrep patterns, we specify a binary relation by using the special symbol "`%`" in match patterns, such that the first node in a relation is the target node of the pattern and the second node is the node corresponding to the symbol preceded by "`%`".

The binary Fxgrep match patterns in Fxt have the following meaning. The primary node is the match node, i.e. it identifies nodes in the input to which the rule is applicable. Given a primary node, all the secondary nodes with which it forms a match pair can be selected together for further processing, similarly as with a unary select pattern. For example, a match pattern `//author[(//%book)][(name/"")]` in a rule specifies that the rule is applicable to `author` elements which have at least one `book` descendant and whose names have some text content. When applying the rule to such an author node it is possible to select all his `book`s descendants, as indicated by the second components of the binary matches denoted by the "`%`" symbol.

As with select patterns, we might want to select more than one set of nodes within a rule. Since each such set of selected nodes corresponds to a binary query, we allow the specification of several binary queries with a single pattern by denoting each binary relation with one "`%`" symbol in front of the corresponding node in a pattern. For example, by using the pattern `//author[(//%book)][(name/%"")]` we will be able to select the `book`s elements inside an `author` as well as the author's name. To refer to one of the set of selected nodes we simply specify which "`%`" symbol defines them, by giving its ordinal number in the pattern. Correspondingly, we allow the `select` attribute of the Fxt actions `fxt:apply` and `fxt:copyContent` for recursive applications and copying to be specified as a positive integer number.

Example 10.3: Consider the following XML input data:

XML Example 32

```
<authors>
  <author>
     <name>Gustave Flaubert</name>
     <book>Madame Bovary</book>
     <book>L'Education sentimentale</book>
  </author>
  <author>
     <name>Marcel Proust</name>
     <book>À la recherche du temps perdu</book>
     <book>Le temps retrouvé</book>
  </author>
</authors>
```

The following transformation produces for every author as above an unordered list (`ul`) of items (`li`) enclosing the content of the `book`s sub-elements.

Fxt Example 20

```
<fxt:spec>

  <fxt:pat>//author[(//%book)][(name/%"")]</fxt:pat>
    Books by <fxt:copyContent select="2"/>:
      <ul>
        <fxt:apply select="1"/>
      </ul>

  <fxt:pat>//book</fxt:pat>
        <li><fxt:copyContent/></li>

</fxt:spec>
```

The first rule handles the `author` elements of interest. The name of the author is copied by selecting the nodes specified by the second occurrence of the `%` symbol, as denoted by the value `2` of the `select` attribute in `fxt:copyContent`. The books are selected for recursive processing via the `select` attribute in `fxt:apply` which refer to the first occurrence of the `%` symbol in the match pattern.

The XML data in XML Example 32 is transformed to:

XML Example 33

```
<authors>
  Books by Gustave Flaubert:
      <ul>
        <li>Madame Bovary</li><li>L'Education sentimentale</li>
      </ul>
  Books by Marcel Proust:
      <ul>
        <li>À la recherche du temps perdu</li><li>Le temps retrouvé</li>
      </ul>
</authors>
```

□

The usage of binary queries in transformations solve the problematic efficiency and expressiveness issues previously mentioned, as we review in the next two sub-sections.

10.2.1 Efficiency

Note that the transformation performed in Example 10.3, could be as well achieved without using binary patterns by:

Fxt Example 21

```
<fxt:spec>

 <fxt:pat>//author[(//book)][(name/"")]</fxt:pat>
   Books by <fxt:copyContent select='name/""'/>:
     <ul>
       <fxt:apply select="//book"/>
     </ul>

 <fxt:pat>//book</fxt:pat>
       <li><fxt:copyContent/></li>

</fxt:spec>
```

We will see in Section 10.3 that every transformation using unary select patterns can be automatically rewritten such that it contains only binary match patterns.

There is however one important conceptual difference between Fxt Example 21 and Fxt Example 20 (on page 136). The unary select patterns in Fxt Example 21 are to be evaluated in the context of the current node, i.e., in the dynamic context of a node matched by the match pattern when this is reached by the transformation. Thus, select patterns must, at least conceptually, be evaluated during the transformation phase.

A binary pattern, like that in Fxt Example 20 on the other hand, simultaneously locates the primary match nodes, and, as secondaries, the sets of related nodes selected within the rule. No dynamic context is needed. Instead, all binary relations specified by binary patterns can be statically tabulated once before the actual transformation begins. Selections during transformation thus boil down to lookups of these tables. As presented in Section 5.3, evaluating *several* binary match patterns simultaneously requires at most two traversals of the input. In contrast, every dynamic evaluation of a pattern requires up to two supplementary traversals of the current subtree, thereby possibly being a major source of inefficiency.

Thus, instead of a potentially unbounded number of traversals, as needed to evaluate the select patterns, we only need two traversals for the evaluation of the binary patterns. This solves the efficiency problem addressed in Section 10.1.

10.2.2 Expressiveness

Moreover, binary queries also lift the expressiveness limitation addressed in Section 10.1. Indeed, rather than only below the current node, the nodes selected for further processing within a rule may be situated at arbitrary positions within the input tree.

Example 10.4: The solution for the transformation addressed in Example 10.1 (on page 133) is:

Fxt Example 22

```
<fxt:spec>
  <fxt:pat>//company</fxt:pat>
    <employees><fxt:apply/></employees>

  <fxt:pat>//department[%id]/employee</fxt:pat>
     <employee>
       <dept><fxt:copyContent name="1"/></dept>
       <fxt:copyContent/>
     </employee>

  <fxt:pat>default</fxt:pat>
     <fxt:apply/>

</fxt:spec>
```

Note how the `id` sibling of the currently transformed `employee` element is directly retrieved when needed. □

Generally speaking, in order to select nodes in a rule one has thus to put them in the context of the node being currently transformed by the rule. This possibly requires refining the match pattern of the rule if the nodes to be selected are not specified in it. One might argue that this makes transformations less error-prone by *requiring* one to think about the context, for instance here. In the example above, for instance, we wanted to provide `employee`s with the name of their department only if this has an `id` element.

Besides for copying, as in the example above, nodes can be selected also for recursive processing. This allows in principle proceeding to nodes others than descendants of the current node, which no longer ensures a strictly top-down transformation of the input and, consequently, no longer ensures termination. Such transformations require supplementary care to avoid infinite computations. Of course, binary queries can be also used to select descendants of the current node. We use this in the following to optimize transformations in terms of time consumption.

10.3 Removing Dynamic Select Patterns

As previously suggested, it is preferable from the point of view of efficiency, to use binary match patterns rather than unary select patterns. Rewriting transformations in order to eliminate unary select patterns is quite simple and can be performed automatically. The idea is straightforward, as presented in the following example.

Example 10.5: Suppose we have an XML document describing the output of some compiler:

XML Example 34

```
<program>
  <declarations>
    <warning>Variable i redeclared</warning>
  </declarations>
  <error>Unexpected assignment</error>
  <main>
     <error>Undeclared variable j</error>
     <warning>Deprecated method doAll</warning>
  </main>
</program>
```

If the compiler report contains both errors and warnings, we might want to output a list containing the errors in bold text followed by the warnings in italic as:

XML Example 35

```
<list>
   Errors:
     <b>Unexpected assignment</b><b>Undeclared variable j</b>
   Warnings:
     <i>Variable i redeclared</i><i>Deprecated method doAll</i>
</list>
```

This can be achieved by using unary select patterns with the following transformation:

Fxt Example 23

```
<fxt:spec>
  <fxt:pat>program</fxt:pat>
     <list>
       Errors:
         <fxt:apply select="//error"/>
       Warnings:
         <fxt:apply select="//warning"/>
     </list>

  <fxt:pat>//error</fxt:pat>
    <b><fxt:apply/></b>

  <fxt:pat>//warning</fxt:pat>
    <i><fxt:apply/></i>

</fxt:spec>
```

The same can be achieved by means of binary patterns as in the following transformation:

Fxt Example 24

```
<fxt:spec>
   <fxt:pat>program[(//%error)?][(//%warning)?]</fxt:pat>
     <list>
        Errors:
         <fxt:apply select="1"/>
       Warnings:
         <fxt:apply select="2"/>
     </list>

  <fxt:pat>//error</fxt:pat>
    <b><fxt:apply/></b>

  <fxt:pat>//warning</fxt:pat>
    <i><fxt:apply/></i>
```

```
</fxt:spec>
```

Here, the binary pattern in the first rule identifies `program` nodes together with their descendants of type `error` and `warning`, respectively, if any. When the rule is applied to a `program` node, the `error` and the `warning` descendants are retrieved via the ordinal reference to the secondary elements specified in the binary patterns and recursively transformed by the last two rules. □

Transformations as above are instances of the special case in which the rule with a match pattern p has an action containing a select pattern consisting of a path p_1. A binary pattern describing the necessary binary relation can be obtained by adding the qualifier $[(p_1')?]$ to the target node in p, where p_1' is obtained by preceding the target node of p_1 with the `%` symbol. The option `?` has to be specified in order to maintain the original applicability of the rule on a node, independently of whether p_1 leads to matches when evaluated for the node or not. Thus, in this case the elimination of the dynamic select pattern p_1 can be performed via syntactical transformations.

In general, however, the elimination can not be performed in a syntactical manner. The reason is that a select pattern can be a path p together with a boolean content model *bcm* for the top level evaluation context. For example, a select pattern `[theorem lemma]/proof`, selects the `proof` elements of the current node if this contains a `theorem` followed by a `lemma` element. Similarly, we need to deal with top level context qualifiers. For example, a select pattern `[theorem#lemma]/proof` selects the `proof` elements of the current node which are preceded by a `theorem` and followed by a `lemma`. The rewriting of the match pattern p of a rule has to capture in general the top-level constraint too. In the examples above, this would be `p[theorem lemma][%proof]` and `p[theorem %proof lemma]`, respectively.

In general, a select pattern can be removed as follows. Let $Q_{match} = ((R, E_0), T)$ be a an extended query as defined in Section 6.1.3 specifying a match pattern of a rule. Let $Q_{select} = ((R', E_0'), T')$ be an extended query specifying a select pattern within the rule. An extended binary query which subsumes the match and the select query as explained below can be constructed as $Q_{locate} = ((R'', E_0), T \times T')$, where :

$$R'' \quad = \quad R \cup R' \cup \{x \rightarrow a\langle bcm \ \& \ bcm' \rangle \mid x \rightarrow a\langle bcm \rangle \in R, x \in T, bcm' \in E_0'\}$$

The rules in the two queries are collected together in $R \cup R'$. Additionally, for any rule for a target in the match query, a new rule is added which requires that the target node fulfills, besides the original boolean content model *bcm*, one of the start content models of the select query. Keeping the rules in R ensures that the rule is applicable for exactly the same nodes as before. The new rules added relate the match node with the nodes to be selected. The conjunction of content models *bcm*&*bcm*$'$ ensures that the relation between the match and the select node is compatible with the original match pattern. One can see that R'' is a function depending on R, T, R' and E_0'. For later reference, we denote it as $R'' = combine((R, T), (R', E_0'))$.

The start content models for the XML input, E_0 remain unchanged. The targets of the match query and the targets of the select query become primary and secondary targets, respectively, in the constructed binary query.

Example 10.6: Consider the following Fxt rule:

Fxt Example 25

```
<fxt:pat>//section[(//corollary)]</fxt:pat>
   <fxt:copyContent select="//*[theorem#lemma]/proof"/>
```

which applied on a `section` containing a `corollary` copies the `proof`s descendants enclosed between a `theorem` and a `lemma`.
The match pattern is expressed by the query $((R, \{\_\ x_1|x_{section}\ \_\}), \{x_{section}\})$, where R is the following set of productions:

$$\begin{array}{lcl} x_1 & \to & *\langle\_\ x_1|x_{section}\ \_\rangle \\ x_{section} & \to & section\langle\_\ x_2|x_{corollary}\ \_\rangle \\ x_2 & \to & *\langle\_\ x_2|x_{corollary}\ \_\rangle \\ x_{corollary} & \to & corollary\langle\_\rangle \end{array}$$

The select pattern is expressed by the query $((R', \{\_\ x_3|x_4\ \_\}), \{x_{proof}\})$, where R' is the following set of productions:

$$\begin{array}{lcl} x_3 & \to & *\langle\_\ x_3|x_4\ \_\rangle \\ x_4 & \to & *\langle\_\ x_{theorem}\ x_{proof}\ x_{lemma}\ \_\rangle \\ x_{theorem} & \to & theorem\langle\_\rangle \\ x_{proof} & \to & proof\langle\_\rangle \\ x_{lemma} & \to & lemma\langle\_\rangle \end{array}$$

The corresponding binary query is $((R'', \{\_\ x_1|x_{section}\ \_\}), \{(x_{section}, x_{proof})\})$, where R'' is the union of rules in R, R' and the rule:

$$x_{section} \quad \to \quad section\langle\_\ x_2|x_{corollary}\ \_\ \&\ \_\ x_3|x_4\ \_\rangle$$

□

The efficiency gain obtained by automatically rewriting transformations containing unary select patterns into transformations using binary match patterns can be very large. This can be observed especially in applications using many select patterns. One such application, which is often given as a sample XML transformation in the literature, takes statistics about baseball players and produces HTML tables containing processed information about the players [Har99]. We wrote this transformation in Fxt using unary select patterns. The optimized version of this transformation, in which select patterns are automatically replaced by binary queries, is between 4 and 5 times faster than the un-optimized one. This great efficiency gain is due to the many select patterns used in the original transformation in this particular transformation. Nevertheless, as selection is a fundamental feature especially in rule-based transformation languages, this benefit is important for transformations in general.

10.4 Grouping via Binary Queries

Grouping nodes sharing a common key as presented in Section 9.6 is a special case of establishing a binary relation. Grouping can be namely considered as establishing a relation between a key and the set of nodes sharing that key. We present below by means of an example how binary queries can be used for grouping.

Reconsider the transformation in Example 9.17, which motivated the need for grouping in Section 9.6. The sample input and the desired output are reproduced for convenience below in XML Example 36 and XML Example 37, respectively.

XML Example 36

```
<books>
  <book>
    <author>Stefan Zweig</author>
    <title>Maria Stuart</title>
  </book>
  <book>
    <author>Giuseppe Tomasi di Lampedusa</author>
    <title>Il Gattopardo</title>
  </book>
  <book>
    <author>Franz Kafka</author>
    <title>Amerika</title>
  </book>
  <book>
    <author>Stefan Zweig</author>
    <title>Sternstunden der Menschheit</title>
  </book>
  <book>
    <author>Franz Kafka</author>
    <title>Tagebücher</title>
  </book>
</books>
```

XML Example 37

```
<authors>
  <author>
    <name>Stefan Zweig</name>
    <title>Maria Stuart</title>
    <title>Sternstunden der Menschheit</title>
  </author>
  <author>
    <name>Giuseppe Tomasi di Lampedusa</name>
    <title>Il Gattopardo</title>
  </author>
  <author>
    <name>Franz Kafka</name>
    <title>Amerika</title>
    <title>Tagebücher</title>
  </author>
</authors>
```

Grouping the books by their authors was achieved in Fxt Example 16 (on page 129) using the following declaration:

Fxt Example 26

```
  <fxt:groupBy group="//books" by="author" name="booksByAuthor"/>
```

Alternatively, key tables can be specified using a binary pattern as below:

Fxt Example 27

```
<fxt:groupBy group="//%book/author" name="booksByAuthor"/>
```

The primary matches of the pattern, in this case the `author` elements, are used to produce the key. Their text content is used by default. The entries in the key tables are the secondary matches of the binary query. A secondary match node belongs to the entry whose key is specified by its corresponding primary match.

As in the case of their usage as a means of selecting nodes, binary queries enlarge the expressiveness of grouping, as they allow the key and the nodes in an entry to be situated at arbitrary relative locations. As opposed to this, using unary patterns for grouping only allowed the key of a node to be retrieved from the content of the node.

For instance, it is for possible to group the `title` elements of `book`s by their `author`, independently of the relative positions of the `title` and `author` elements inside a `book` element. Thereby, the transformation above can be achieved by the following transformation:

Fxt Example 28

```
<fxt:spec>
  <fxt:groupBy group="//book[%title]/author" in="titlesByAuthor"/>

  <fxt:pat>/*</fxt:pat>
<authors>
  <fxt:forAllKeys in="titlesByAuthor">
    <author>
      <name><fxt:getKey/></name>
      <fxt:copyKey/>
    </author>
  </fxt:forAllKeys>
</authors>

</fxt:spec>
```

We iterate over the entries for each author and output the name as the value of the key via `fxt:copyKey` and the titles as the nodes stored in the entry via `fxt:copyKey`.

Compare Fxt Example 28 with the equivalent transformation in Fxt Example 16 (on page 129). There, it was not directly possible to key the titles under their author's name (because `author` is not a descendant of a `title` element). We instead had to group book elements by author and transformed the book elements to retrieve their titles. The new transformation appears to be more declarative and clean, due to the expressive power of binary queries.

10.5 Decoupling Navigation from Transformation

Finally, another advantage of using binary queries in rule-based transformation is a further step towards separation of concerns. Binary queries, namely, might help decoupling the task of locating matches of patterns from the task of constructing XML content in transformation rules. Consider an input document in which an `author` element contains all the `book`s written by the author. The following Fxt rule produces for each author a table row containing the `name` of the author and the books written by him:

Fxt Example 29

```
<fxt:pat>//author[(//%book)][%name]</fxt:pat>
  <tr>
    <td><fxt:copyContent select="2"/></td>
    <td><fxt:copyContent select="1"/></td>
  </tr>
```

If the structure of the input document changes s.t., for example, the books written by every author follow after the `author` element, only the match pattern has to be modified to `//*[#_ %book]/author[%name]` in order to account for the new relation between the author and his books, in order to achieve the same transformation.

Generally speaking, in this perspective, the action part uses the relation defined by the match pattern. The same informational relations might be encoded differently in different documents. However assuming that these relations can be extracted using match patterns, the action part is concerned only with building the desired layout from the extracted information. The action parts can be thus shared among transformations which use as input similar but differently structured data.

10.6 Conclusions

This chapter has shown that Fxt benefits from using binary queries in rule-based transformations in different respects, as follows.

Expressiveness Binary queries allow the selection of nodes situated at arbitrary positions relative to the node at which the selection takes place, rather than only below this node.

Declarative style Rather than explicitly indicating the navigation path from a node to a second one to be selected in its context (as if we were to use XPath select patterns) the relation between the two nodes is specified by refining the context of the first node s.t. it captures the second node.

Efficiency Locating binary matches in the static global context requires at most two traversals of the input XML data. In contrast, locating unary matches in the dynamic context of a node requires traversing the content of this node. This might result in repeatedly visiting the same node an unbounded number of times.

While exemplified using Fxt, the considerations made in this chapter might be also applied in other XML transformation tools, in particular in implementations of XSLT and XQuery. Suggestions in this direction will be presented in Chapter 12. Before that, in Chapter 11, we present the implementation of Fxt.

Chapter 11

Implementation of Fxt

The previous chapters introduced the specification language of Fxt transformations. This chapter provides an overview on the implementation of the Fxt language. As Fxt is a domain specific language (DSL) we start by presenting the DSL approach used. Then, we justify the choice of the implementation language. We describe the system architecture and conclude with experimental results which show the efficiency of Fxt.

11.1 Fxt as Pre-Processor

Rather than transforming XML data, Fxt generates XML transformers (as mentioned in Chapter 9). This is conceptually described in Figure 11.1. Given an Fxt transformation specification, the generator produces code in the Standard ML programming language (SML) [MTHM97]. The generated code can be subsequently either directly included in SML application programs, or compiled via an SML compiler and used as a stand-alone application.

This approach to implementing a DSL is not new. In fact, system design and implementation here follows the *pre-processing* paradigm for DSL implementation as sketched in [DKV00]. The advantage of such an approach is obvious: it clearly relieves us from re-implementing compiler support for standard programming language features. In particular,

- ✧ we participate in all general enhancements of compiler implementation for free;
- ✧ engineering, extending and re-engineering of our prototypical language

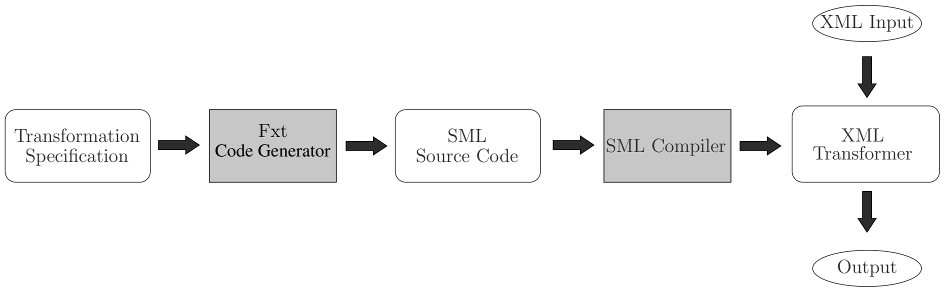


Figure 11.1: Fxt as pre-processor

design could very rapidly be implemented;

✧ embedding SML code into specifications (where needed) becomes essentially trivial.

The drawback of this approach, however, is also apparent: certain non-syntactical errors are currently caught not in the pre-processing phase, but only in the follow-up compilation phase — where the original source of malfunction is more difficult to track. This problem is partly alleviated in our system by generating comments which point back from the generated code to the corresponding Fxt source locations.

11.2 The Implementation Language

Standard ML

For the implementation of Fxt we chose the SML language. The decision was based on a couple of practical considerations as presented below. One of the main reasons to use SML was that the original implementation of the pattern language used by Fxt , Fxgrep , was also implemented in SML. Interfacing the two can be thus performed naturally via function calls in SML. Besides that, there are more advantages of using SML, some of them being already taken into consideration in the decision to use SML for Fxgrep, as follows:

Tree processing As a functional language, SML is a natural choice for processing tree structured data like XML documents. The declarative style of tree processing via recursive calls in functional languages is much clearer as compared to iterative approaches. The pattern matching primitives offered by SML also offer a good basis for extracting information from trees in a more declarative and less error-prone way as compared to explicitly using references to denote the edges in trees as in an imperative, object-oriented language.

Type system SML is a *strongly typed language*, meaning that an SML compiler can ensure that the accepted programs will execute without type errors. Besides ensuring robustness at the run time this allows many programming errors to be detected early. A comfortable feature is that types do not need (mostly) to be explicitely indicated, as they are automatically inferred by the compiler.

Parametric modules SML supports a modular development of programs via grouping of related types, values and functions inside *structures* and providing views thereof via *signatures*. Moreover, via *functors*, modules are parameterizable by taking as arguments signatures of other modules. For example, an event-based parser can be implemented as a functor parameterized with a structure which contains the event-handlers.

Polymorphism Another feature which supports reusing and maintaining the code is *polymorphism*. A polymorphic type is a type parameterized with other types. Polymorphic functions are functions which can handle arguments of polymorphic types without depending on the parameter types.

For example to compute the height of a tree one might use a polymorphic function working on trees, the nodes of which may be any fixed type, as the implementation does not use the information stored in the nodes. A single function declaration is enough to operate on trees of boolean, strings or whatever type as long as their structure is the same.

Imperative features SML is not a *pure functional language* meaning, that is, it also allows the use of side-effects as in imperative languages, in particular by providing updateable references. While this is to be avoided most of the time, there are situations where imperative updates are more natural and also efficient, as for example when updating entries in hash tables.

Standard ML of New Jersey

More implementations of SML exist. We chose to use Standard ML of New Jersey language (SML/NJ) [SML05] implementation. One of the main reasons is again the legacy of Fxgrep, which is also implemented in SML/NJ. There are also other important advantages as described in the following.

Compilation Manager A distinguishing feature of SML/NJ among other SML implementations is its compilation manager (CM) [Blu97]. CM helps maintaining large programs by automatically establishing the dependencies between the different modules in the program and re-compiling after a change to the source code only the modules influenced by the change. This has on the one hand the advantage that the re-compilation is generally performed very quickly and on the other hand relieves the programmer from the tedious task of manually maintaining the dependencies among the system units.

The Compiler Interface SML/NJ provides via its Compiler structure access to the SML/NJ compiler. The SML/NJ primitives for manipulating compile and run-time environments are designed to allow for *incremental compilation*. This basically means that compiling a source in an given environment results in a new environment obtained by adding the new bindings to those available in the input environment [AM94]. This allows in principle the implementation of a compilation manager like CM as a usual SML/NJ program [HLPR94]. Of particular importance for us, this compiler interface provides a convenient basis for *meta-programming*, i.e. for the synthesis and evaluation of ML code by user programs [AM94], as in the case of Fxt.

Exporting and Importing Heap Images A remarkable feature of SML/NJ runtime is its ability to save a frozen copy of the heap (called *exporting the heap image*) and use it to perform a computation in the same environment at a later moment. The SML/NJ heap not only contains the values dynamically created during the computation but also the code compiled up to the moment. Before exporting the heap a garbage collection is performed which gets rid of the unreachable objects and also only keeps the code necessary to run the exported functionality.

This feature is very convenient for sharing information between the generation and the execution phase of an Fxt transformation. For example,

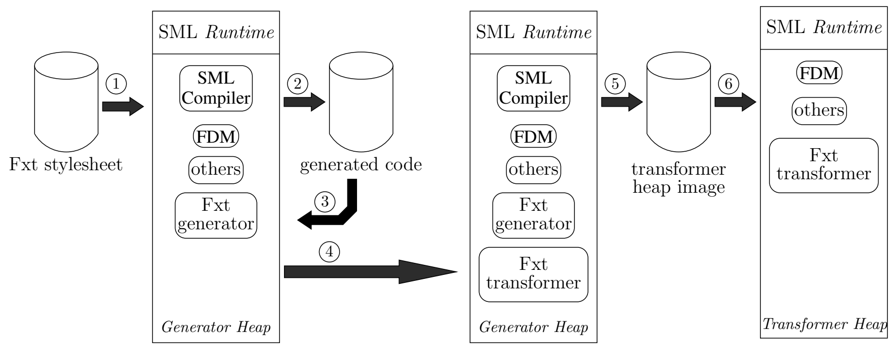


Figure 11.2: Fxt's compilation model

the match patterns are compiled to automata in the transformation phase, and the data structures containing the automata are exported in the heap image which will be imported by the transformation when this is called. Thereby, we are able to save the computational effort which would have otherwise been needed to serialize the shared data structures to SML code in the generating phase and re-compiling them in the compiling phase.

A few other ML dialects exist other than SML [LDG$^+$05, BRS$^+$05, CEKL05]. Better known among them is probably OCaml [LDG$^+$05] which can produce both bytecode and native code. The code generated by OCaml is generally known to run faster as compared to the code generated by SML/NJ. On the other side, however, it does not provide the convenience of a compilation manager as in SML/NJ. Neither is the comfort provided by other languages implementing SML, like MoscowML [Ses05], MLton [Wee05] or SML.NET [BKR05].

11.3 Compilation of Fxt Transformations

As previously mentioned, the Fxt generator of XML transformations uses the SML/NJ features of incremental compilation as well as exporting and importing of heap images. The steps of the code generation and compilation are sketched in Figure 11.2. The compiled code for Fxt together with the libraries it uses, like the Functional Document Model (FDM), as well as the SML compiler lie in the heap of the Fxt process executed via the SML runtime environment. Fxt receives the specification of a transformation as an Fxt stylesheet (step 1) and generates SML code to perform the required transformation (step 2). The same runtime is used to compile the generated code (step 3) and as result the heap is enriched with the compiled code needed to perform the transformation (step 4). The transformation is exported in step 5. Only the strictly necessary code (e.g. the FDM or library functions invoked by the user code in the stylesheet) and dynamic structures (e.g. the automata to perform the pattern matching) are automatically retained by the SML system upon exporting the transformation function. When the transformation is invoked via a new environment the previously exported heap image is imported and the transformation may directly proceed to its task (step 6).

11.4 System Architecture

The architecture of Fxt is depicted in Figure 11.3. The parser module handles both the XML input to be transformed and the Fxt stylesheet as this is also an XML document. We use the event-driven Fxp parser [Neu04]. The XML input is parsed via the tree event handlers which produce the forest representation of the XML input data. The Fxt front-end handlers basically deliver the rules specified in the transformation in the form of (match) patterns and their corresponding actions.

The actions are transformed into SML expressions of the FDM type Forest by the action generator module. The transformation generator uses the code generated for the actions to generate a transformation which basically is a function expecting an XML input annotated with match information and evaluating the corresponding action. The generated transformation code is subsequently compiled as described in Section 11.3.

The patterns are compiled by the pattern compiler module into a dynamic structure in the heap representing the pre-processed information needed by the automata implementing the pattern matching.

The compiled patterns and the generated transformation are to be used for the evaluation of the transformation performed by the evaluator module. The evaluation of the transformation does not have to be performed immediately, in the same runtime as the code generation and compilation. Alternatively, the heap image containing the transformation and the compiled patterns can be persistently saved by the exporter and may be subsequently restored by the importer module and delivered to the evaluator.

The input of the transformation is provided by the parser which delivers the input forest obtained by parsing the XML input. The pattern evaluator annotates the input with match information and passes it to the transformation evaluator which invokes the transformation function to produce the desired output.

11.5 System Evolution

The first release of Fxt contained the basic functionality of the rule-based transformer. Recursive application of the transformation was only allowed on children of the current element. It is often however useful to apply the transformation directly on some proper descendant, rather than on a direct child of the current node. This was possible by repeatedly selecting the child node via a default rule repeatedly applied until the desired descendant was reached and processed by the appropriate rule; but it sometimes made the transformation look unnecessarily complicated. Also, it required visiting all the descendants of the node up to the nodes of interest.

We therefore introduced select patterns which allow to directly select the nodes of interest in the context of the current node. As argued in Section 10.1, the evaluation of select patterns may be expensive in any rule-based language, since a direct implementation may require visiting some nodes in the input an unbounded number of times.

We removed this inconvenience later by automatically removing the select patterns via binary match patterns as presented in Chapter 10. This limits the

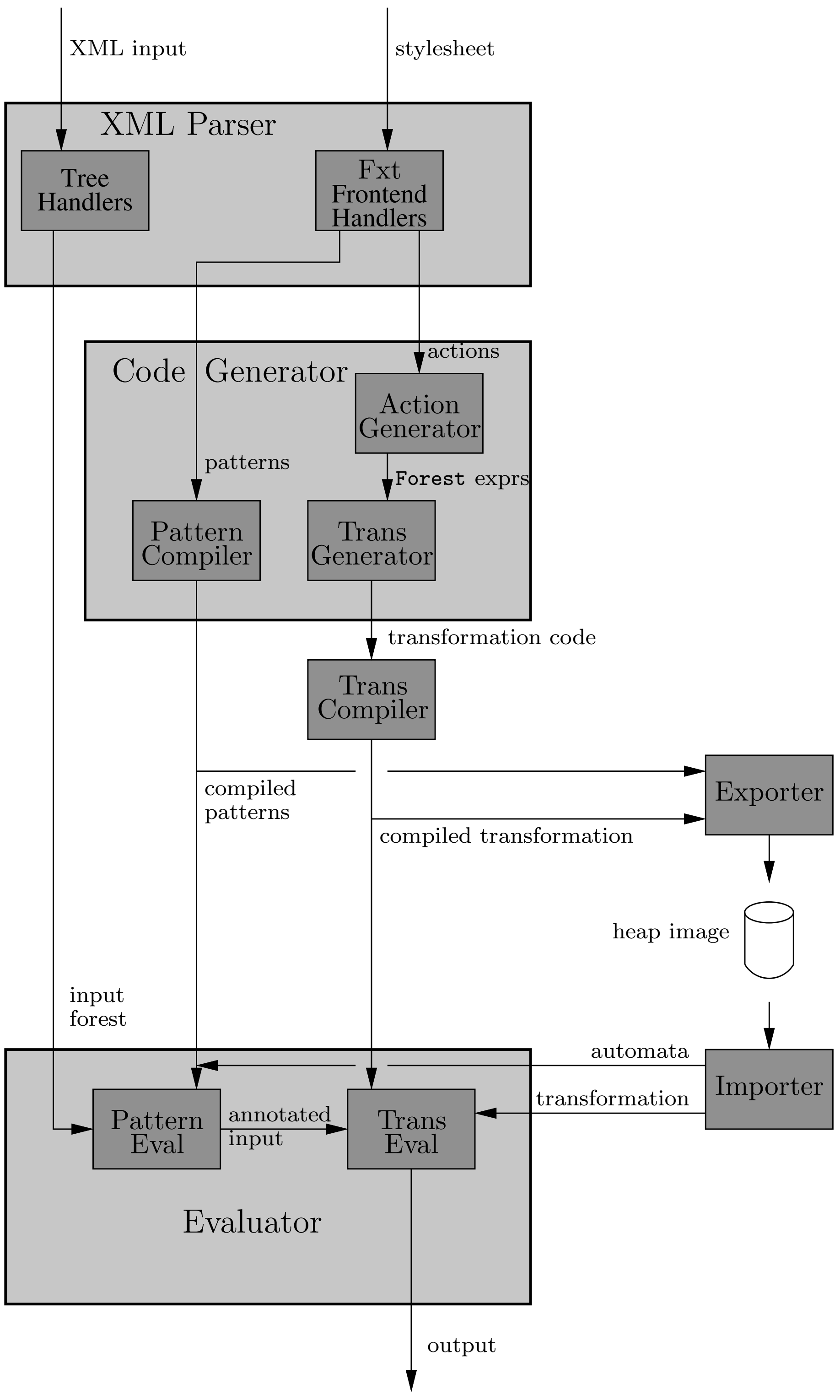


Figure 11.3: The system architecture of Fxt

number of traversals of the input tree to at most two. Also, selecting nodes via binary match patterns extended the expressiveness of Fxt by allowing the selection of nodes from everywhere around the current node, not only from below it. Binary patterns were further allowed as a means of specifying tables which can be mainly used for grouping purposes as presented in Section 10.4.

The Fxgrep pattern language used by Fxt evolved in parallel with Fxt. The possibility of expressing binary queries with Fxgrep initially arose from the select patterns necessities of Fxt. The other extensions of Fxgrep, such as conjunctions and negations of context as presented in Section 6.2 have also had to be accounted for in the interface of Fxt to Fxgrep.

A more detailed description of the evolution of Fxt during the research project can be found in the change log of the Fxt distribution.

11.6 Experimental Results

In order to assess the time performance of Fxt we compared it with two of the most popular XML processors. The first is the Xalan Java processor version 2.6.0 which is part of the Apache XML Project [Pro05]. As denoted by its name, this processor is written in Java. Xalan is currently one of the most used XSLT[1] processors, being part of the reference implementation of the standard Java Application Programming Interface for XML processing (JAXP [Mic05]) in the Java 2 Platform, Standard Edition 5.0. The second XSLT processor that we used is the open source Saxon XSLT processor [Kay05], version 6.5.3, also written in Java.

We considered the following benchmark transformations:

- Birds
  - XML Input (9.2 KB): Description of classes of birds
  - Output: Plain text file presenting the information in the input in an indented manner

  This transformation traverses the document in the implicit depth-first left-to-right manner. The text content of the nodes of interest is output using different indentations, depending on the type of the nodes. These nodes are identified by simple match patterns consisting only of the names of the nodes.

- Article
  - XML Input (80 KB): An article about Fxt [BS02] conforming to the DTD format for the Extreme Markup Language Conference Proceedings
  - Output: An HTML layout of the paper

  This is a typical stylesheet transformation. Most of the transformation rules relabel the nodes denoting the logical format of the paper with corresponding HTML annotations. The patterns used here are slightly more

[1] A comparison of Fxt with the XSLT language is given in Section 12.2.1.

elaborated than in the *Birds* application, as they sometimes need to distinguish among elements with the same name in different contexts. Additionally, bibliographical references, footnotes and cross references have to be handled. This is achieved in Fxt via keyed tables as presented in Section 9.6. Moreover the sections have to be numbered and an index has to be constructed. We perform this in Fxt by using variables as presented in Section 9.5.

✧ T1 and T2

- XML Input (205 KB): Shakespeare's "All's Well That Ends Well" play [Bos99]. A play is a sequence of `ACT`s, each of them containing a sequence of `SCENE`s. A `SCENE` has a sequence of `SPEECH`-es, each of them containing a `SPEAKER` and a sequence of `LINE`s containing plain text.
- Output: A list of paragraphs containing matches of patterns which are slightly more elaborated than the very simple patterns used in the previous transformations.

Transformation T_1 collects all lines in speeches of `Lafeu` appearing in scenes where `Bertram` is also present [2]. Transformation T_2 collects all speeches in scenes containing a line having the word "husband" and being in an act containing a line having the word "abundance"[3].

We wrote the stylesheets performing the above transformations in both Fxt and in XSLT and correspondingly used them as specifications for the different XML transformers. Every transformation was executed 10 times and the average execution time was computed. The measured times include the startup phase of the SML runtime and Java Virtual Machine (JVM), respectively.

The benchmarks were executed under Linux (kernel version 2.6.8) on a AMD Athlon XP 3000+ processor with 1 GB of memory. The JVM used to run the benchmarks with the Java XSLT implementations was the Sun JVM implementation Java version 1.5, which uses by default a *just-in-time* compiler (JIT). The SML version used for Fxt was 110.0.7.

We considered both the interpretive and the compilation approach, where applicable. In the compilation approach a stylesheet is compiled once, and subsequent executions of the same transformation may be performed by directly executing the compiled code, thereby saving the time needed to process the stylesheet. As presented in Chapter 9, Fxt always compiles the stylesheet. In its "interpretive" approach, Fxt immediately uses the compiled code in order to perform the required transformation on the presented XML input. In the compilation approach the compiled code is saved and can be used to directly perform the same transformation an arbitrary number of times. Xalan Java also includes a similar feature via its included XSLT compiler (XSLTC). Given an XSLT specification, XSLTC generates a Java class which can be subsequently used to perform the transformation. Saxon 6.5.3 (the last version implementing XSLT 1.0) only functions in the interpretive approach.

[2] Expressible with the XPath pattern `SCENE[.//SPEAKER="BERTRAM"]/SPEECH[SPEAKER="LAFEU"]/LINE`

[3] Expressible with the XPath pattern: `ACT[.//LINE[contains(.,"abundance")]]/SCENE[.//LINE[contains(.,"husband")]]/SPEECH`

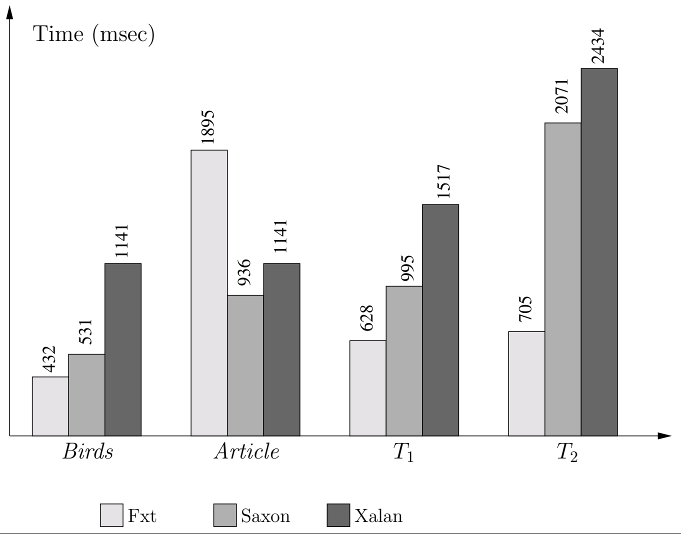


Figure 11.4: Transformation times in the interpretive approach

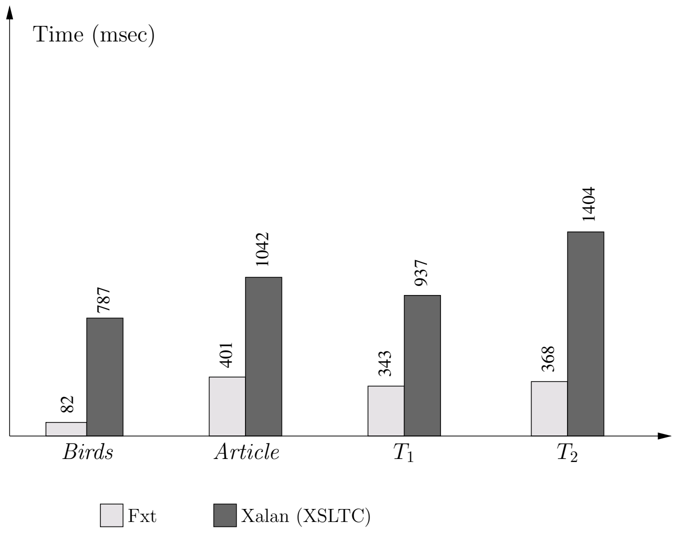


Figure 11.5: Transformation times in the compiling approach

The results for the interpretive and compilation approaches are graphically depicted in Figure 11.4 and Figure 11.5, respectively. The times indicated in the interpretive approach include the times needed to process the stylesheet. The times indicated in the compilation approach are the times needed to run the compiled code.

Fxt proved to be generally faster for most of the considered applications, both in the interpretive and in the compilation approach. The good performance of Fxt is especially visible in the compilation approach. In the case of the quite simple *Birds* transformation Fxt was almost ten times faster than the XSLT processor.

The more complex *Article* application clearly shows the advantage of compiling the stylesheet. Most of the time needed for executing this transformation in the interpretive approach by Fxt is taken by the compilation of the stylesheet which makes Fxt slower than the Java processors. Executing the same precompiled Fxt transformation however is significantly faster, even when compared with the equivalent compiled Xalan Java transformation.

A strength of Fxt is its pattern language, Fxgrep. While being quite expressive, Fxgrep can be also efficiently implemented. The advantages of using Fxgrep as a pattern language become visible as the complexity of the patterns used in the XML transformations increases. All transformations considered above use very simple patterns. In contrast, the transformations T_1 and T_2 contain more elaborate, yet completely meaningful patterns as mentioned above. One can see that the relative performance of Fxt was very good. Moreover, by comparing the execution times of Fxt for T_1 and the more complex T_2, one can note that the complexity of the patterns does not significantly influence Fxt, in contrast to the XSLT processors.

For T_1 and T_2 we also considered the dependency of the transformation time on the size of the input document. The input document was augmented by duplicating the `ACT`s of the play, which is doubling the breadth of the input tree. The expected effect on both the Fxt and the XSLT transformations is that of doubling of the transformation steps. The size-time dependency proved to be indeed linear.

11.7 Conclusions

As presented in the case of Fxt, most of the time spent during an XML transformation is likely to be spent for the purpose of locating matches of patterns, i.e. for query evaluation. The overall good performance of Fxt which can be observed in the experimental results presented in the previous section is correspondingly mainly due to the efficient implementation of queries in Fxgrep. An important contribution to the overall good performance of evaluating queries in Fxt is obtained via the optimization presented in Section 10.3, which removes the expensive select patterns by efficiently implementable binary queries.

Furthermore, Fxt is another proof that functional programming languages can keep up with imperative ones, while offering the safety and the comfort emerging from extensive static type checking, and type inference, and that they are very suitable for processing tree-structured data.

Chapter 12

Fxt vs. Other Transformation Languages

As argued in the introduction in Section 8, non-trivial XML processing requires a domain specific approach. In particular, constructs and syntax specially tailored for XML processing are needed in order to achieve readable and easily composable transformations.

DSL approaches, and in particular DSLs for XML processing, can be roughly divided into two categories: *embedded* and *stand-alone*. Embedded DSLs are extensions of general purpose programming languages (GPLs) with domain specific syntax and constructs. In contrast, stand-alone DSLs are independent of any constraints which an underlying GPL might impose. Fxt , XSLT and XQuery are for example stand-alone DSLs for XML processing.

Of course, both DSL approaches have their strengths. The stand-alone approach enjoys the complete freedom of choosing a suitable syntax and implementation. This allows for instance Fxt or XSLT to provide an implicit control flow as required by rule-based transformations. Also, using a stand-alone DSL does not require being aware of anything else other than the application domain, as opposed to an embedded language which additionally requires familiarity with the underlying GPL. Consequently, stand-alone DSL can be thus used also by non-programmers.

On the other side, embedded DSLs can directly use the computational power and the functionality provided by the underlying GPL and their use only requires the programmer familiar with the GPL to additionally learn a number of domain-specific constructs. Embedded DSLs are targeted thus exclusively at programmers.

The distinction between embedded and stand-alone DSLs is nevertheless not always very strict. Fxt for example has features from both, as it offers access to a GPL (SML) via optional embedding of SML code into transformations. Other stand-alone DSLs are developed such that they resemble GPLs with added domain specific features, hence their strict classification is not possible. As a general rule, in this chapter we consider to be standalone DSLs those which do not depend on some previously existing GPL.

The chapter addresses different approaches to XML transformations that have been proposed and relate them with Fxt, where possible. Embedded DSL approaches are reviewed in Section 12.1. Stand-alone DSL approaches are handled in Section 12.2.

12.1 Embedded DSL Approaches to XML Transformations

One of the main concerns in the embedded DSL approaches is static type checking, i.e. statically guaranteeing the validity of dynamically generated XML data. In the case of a statically typed GPL, when XML element types are integrated in the type system of the language, it is possible to provide some guarantees that if an input document conforms to a given input schema, the output produced by the transformation conforms to a given output schema. Checking the consistent use of XML content with the declared input and output types is supported in these cases by the type checker of the language. Stand-alone DSLs like XQuery also provide some form of static type checking. For aspects related to the type checking problem of XML transformation languages we refer the interested reader to the overview by Møller and Schwartzbach [MS05]. In particular, type checking XML rule-based transformations, which use MSO expressible binary queries as patterns (like Fxt), is considered by Maneth *et al.* in [MBPS05]. In contrast, in the following, we address the XML processing languages from the point of view of the transformation primitives that they offer.

12.1.1 Object-oriented Programming Languages

XML processing has received from its beginning a large support in the Java language, in which many of the available XML tools have been written. A number of projects like XOBE, JWig, XACT or XJ are concerned with extending the Java language with support for XML processing. All of them are implemented as Java pre-processors.

XOBE [KL03, KL04] extends the Java language with XML objects as first-class values, i.e. values which can be manipulated like any other values in the language. XML objects can be represented directly in XML syntax, while an escape mechanism inside them allows one to embed arbitrary Java expressions. Referring to sub-components of XML trees is directly possible using XPath patterns on XML objects.

XACT [KMS04], similarly to its predecessor languages BigWig [BMS02] and JWig [CMS03], offers as main type for representing XML values *XML templates*, which are XML tree fragments with named gaps which can be filled with values in an arbitrary order and are treated as first class values. The decomposition of XML values needed in XML transformations is achieved via XPath. XPath patterns are for instance used to select subtrees or gaps to be filled or created. Syntactical support is offered for representing constant XML fragments and XPath expressions directly in the XML syntax.

Similarly to XOBE and XACT, XJ [HRS$^+$05] integrates XML data as first class values into Java and uses XPath to navigate these values. As distinguishing features, XJ aims at integrating the XML Schema types into the Java type system and explicitly permitting destructive updates of the XML data.

A few projects are concerned with extending C# [Csh05] with support for

XML processing. Cω [BMS03] extends the type system of C# s.t. XML data and SQL tables can be uniformly represented as first class values in the language. To query and transform these values, the member access available in object oriented languages is generalized with wildcard, transitive and type-based member access, obtaining a functionality similar to the basic functionality of XPath. Another approach to enhancing C# for convenient XML processing is Xtatic [GLPS05], a successor project of XDuce (which is addressed in Section 12.1.2), which builds upon the type system and the pattern matching ideas of XDuce.

12.1.2 Functional Programming Languages

From the very beginning, the application domain of processing hierarchically structured documents has been attracted by the functional programming style of declarative specifications. So, the syntax of SGML [Int86], the markup language which preceded XML, as well as the XML syntax is similar to Lisp expressions. Also, the document transformation language for SGML, DSSSL [Int96], originally had been designed as a superset of Scheme.

Also, popular languages like XSLT and XQuery have one of the basic features of functional languages, namely they are side-effects free. The advantage is that the components can be combined arbitrarily, without the need to think about the order in which an implementation will evaluate them. As in functional languages, every language construct returns a value, in both cases a sequence of XML nodes.

However, there are more typical functional constructs which make up a fully fledged functional programming language, like higher order functions, pattern matching, static type checking, polymorphism. Even though some typical higher order functions can be implemented in XSLT (as e.g. in FXSLT [Nov01]), XSLT and XQuery were not conceived to explicitly support the functional programming style. The need for a functional style of XML processing was recognized by many authors, as for example by Parsia in [Par01].

In [WR99], Wallace and Runciman provide a library for XML processing in the general purpose functional language Haskell. The approach followed is a popular solution to extending functional languages to specific domains by enriching them with a library of combinators [SAS98]. Combinators are higher-order functions which can be used to provide more elaborate functionality out of basic processing functions. Combinators are to be uniformly defined such that they can be themselves flexibly combined with one other. A small core of functions are defined in terms of which it should be possible to express all the functionality required by the specific domain. One advantage of the combinator approach in general is that a number of algebraic laws for combinators can be derived which could be used for code optimizations. However, even though a set of operators can be defined to improve readability, the syntax remains limited to the syntax of the implementation language, which in particular is not suitable for describing XML content. Furthermore, writing transformations in terms of combinators leads itself to not especially readable code.

Designing a new functional programming language dedicated to XML processing like in XMLambda [MS99], XDuce [HP03] or CDuce [BCF03] has the advantage that a syntax convenient for XML can be chosen. Actually, these languages could be as well considered stand-alone DSLs, but we prefer to mention them here, as they can be seen as extensions of general-purpose functional programming languages.

The language proposed in [MS99], XMLambda, is similar to Haskell but dedicated to XML. XML elements are basic values in the language and can be written directly in the XML syntax. An escape mechanism allows embedding arbitrary expressions of the language in the XML content. Element types from DTDs can be represented one to one into types in the language. XMLambda uses this type information to statically check that the expressions have their required types. The type checker works similarly to a validator of XML content but takes additional care of the embedded expressions.

Similarly to classic functional languages, pattern matching with variable bindings is also provided. A pattern for an XML element can be seen as an XML tree in which variables are used as place-holders for XML content. When an XML value is matched against the pattern, the variables are bound to the corresponding XML content. To resolve possible ambiguities, type annotations have to be provided for the variables in the pattern. As opposed to Fxgrep and XPath patterns, the XML information which may be selected via variable bindings in patterns is always at a fixed depth within the node against which matching is performed. No deep matching capabilities are provided. The ideas of XMLambda were pursued by Shields and Meijer in [SM01], yet apparently they were not made available in any practical implementation.

In contrast, a publicly available implementation of XDuce [HP03] exists, which is another functional language specially designed for XML processing. Similarly as in the XMLambda proposal, in XDuce, the traditional pattern matching capabilities from functional languages are extended with regular expression constructs. To avoid using tedious type annotations, a type inference scheme is provided which, given the type of a value to be matched against a pattern, automatically infers the type of the variables in the pattern. Basically, the XDuce patterns are forest grammars. XML values can be de-constructed into their component parts by using patterns with variables. A variable in a pattern is a name for a distinguished sub-pattern and allows one to individually address sequences of nodes of arbitrary lengths. Evaluating a pattern with k variables simultaneously binds the k variables, and can be thus seen as a k-ary query. Patterns may be in general ambiguous. That is, matching against an input value may bind a variable to different values. XDuce initially solved the ambiguity by adopting a left longest match policy, and later by issuing a warning and non-deterministically choosing one possibility. This ensures yielding at most one match tuple, as required for the purpose of pattern matching in a programming language. An *all-matches* semantics is however more suitable for a query language, both as a stand-alone tool or embedded within a rule-based transformation language as in Fxgrep and Fxt, respectively. Nevertheless, as mentioned in Section 5.4, in the presence of a disambiguating policy, pattern-matching with k variable bindings can be efficiently implemented using push-down forest automata.

XDuce focuses on static type checking and does not provide any efficient algorithm for pattern-matching evaluation other than naive backtracking. CDuce [BCF03] is based on XDuce and improves its pattern matching evaluation by an implementation based on a combination of top-down and bottom-up tree automata [Fri04] similar to the pushdown forest automata and optimized to take static type information into account. Besides, CDuce extends XDuce with the use of higher-order functions, more basic types, non-linear capture variables, products, records and handling of XML attributes.

12.2 Stand-alone XML Transformation Languages

In this section we consider XSLT and XQuery as the prominent XML transformation languages available. Note that the term "query languages" occurs with different acceptances in the literature. In some of them "querying" is not distinguishable from "transforming", as one can see for example in the name of "XQuery". In contrast, we adopt the terminology in which a query language is a language which, given an input XML document, only identifies sub-documents having some specified properties.

12.2.1 Fxt and XSLT

As a stable W3C Recommendation, XSLT 1.0 [XSL99] is probably the most widely used XML transformation language. As a rule-based transformation language, XSLT has a processing model similar to Fxt. The basic constructs of XSLT, like those of Fxt, offer possibilities for selecting and copying existing nodes, creating new XML nodes, and recursively applying transformation rules. The syntax of XSLT and Fxt is also similar, as they both adopt an XML syntax. As in Fxt, by default, element names occurring in a transformation specification are to appear as such in the output. Calls to transformation primitives are given by elements with reserved names.

Example 12.1: Compare the Fxt specification in Fxt Example 1, producing a list of section titles, reproduced for convenience below in Fxt Example 30, with the equivalent transformation in XSLT Example 1. Rules, called *templates* in XSLT terminology, are defined in XSLT via `xsl:template` elements. The match pattern is given as the value of an attribute `match` and the corresponding action is given by the content of the `xsl:template` element.

Fxt Example 30

```
<fxt:spec>
  <fxt:pat>/*</fxt:pat>
    <ul>
      <fxt:apply/>
    </ul>

  <fxt:pat>//section/title/""</fxt:pat>
    <li>
       <fxt:current/>
    </li>

  <fxt:pat>default</fxt:pat>
    <fxt:apply/>
</fxt:spec>
```

XSLT Example 1

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/*">
```

```
    <ul>
      <xsl:apply-templates/>
    </ul>
  </xsl:template>

  <xsl:template match="//section/title/text()">
    <li>
      <xsl:copy-of select="."/>
    </li>
  </xsl:template>

  <xsl:template match="text()|@*"/>

</xsl:stylesheet>
```

The match patterns of the first two rules are equivalent in the two transformations, up to the syntactical differences between XPath and Fxgrep , the pattern languages used by XSLT and Fxt, respectively. The recursive application of the transformation rules on the children of the current element, performed in Fxt by the `fxt:apply` element in the first rule, is achieved in XSLT by the `xsl:apply-templates` element. Copying the current node, achieved in Fxt by the special element `fxt:current`, is specified in XSLT by selecting the current node with the XPath pattern "`.`" and copying it via the `xsl:copy-of` element. The `xsl:copy-of` is the XSLT correspondent of the `fxt:copyContent` element. Finally, the last rule specifies in both transformations that nodes not matched by any previous rule are to be ignored. Since, in XSLT, element nodes are anyway ignored if they do not fulfill any match-pattern, this remains to be explicitly required only for text content and attributes, as in the last rule in XSLT Example 1. □

Despite their similar syntax, XSLT and Fxt are quite different in terms of expressiveness, as described next.

Pattern Language

A main difference is the pattern language used. XSLT uses general XPath patterns, as select patterns, and XPath patterns in which navigation is restricted only downwards in the input tree, as match patterns. Fxt, on the other hand, uses general Fxgrep patterns both as match and select patterns. The different expressiveness of Fxt and XSLT mainly originates in the different expressiveness of Fxgrep and XPath (presented in Section 3.4).

Selection

Besides by unary patterns as in XSLT, nodes can be selected in Fxt by using binary patterns. As presented in Chapter 10, this allows the extraction of supplementary information from a match pattern which can be used in the action part of the corresponding rule. A binary pattern may deliver not only the target node of the rule but also the nodes which caused this to be a target node, as matches of an arbitrary node in the pattern. For example the pattern `//chapter[(//%section//"query")]/title` not only locates the title elements of chapters containing section sub-elements containing the word "query", but also allows the selection of exactly these sections.

Even though Fxt and XSLT are not directly comparable because of the different capabilities of the underlying pattern languages, one might argue that the use of binary patterns in Fxt is an increase in terms of declarativeness as compared to XSLT, as suggested in the following.

Example 12.2: As an example of the different flavors of selection in Fxt and XSLT re-consider the XML transformation addressed in Example 10.1, which is reproduced for convenience below. It handles documents having `department` elements which contain an `id` element followed by a sequence of `employee` elements as for example:

XML Example 38

```
<company>
  <department>
    <id>Public Relations</id>
    <employee>Jan Smith</employee>
    <employee>Meg Rush</employee>
  </department>
  <department>
    <id>Sales</id>
    <employee>David Hughes</employee>
    <employee>Angela Dimm</employee>
  </department>
</company>
```

The transformation is to produce a list of employees, each of them containing the `id` of his department, as below:

XML Example 39

```
<employee><dept>Public Relations</dept>Jan Smith</employee>
<employee><dept>Public Relations</dept>Meg Rush</employee>
<employee><dept>Sales</dept>Don Hughes</employee>
<employee><dept>Sales</dept>Angela Dimm</employee>
```

The Fxt solution, presented in Fxt Example 22, is reproduced for convenience below:

Fxt Example 31

```
<fxt:spec>
  <fxt:pat>//company</fxt:pat>
    <employees><fxt:apply/></employees>

  <fxt:pat>//department[%id]/employee</fxt:pat>
     <employee>
       <dept><fxt:copyContent name="1"/></dept>
       <fxt:copyContent/>
     </employee>

  <fxt:pat>default</fxt:pat>
     <fxt:apply/>

</fxt:spec>
```

The binary Fxgrep pattern simultaneously locates an `employee` and the `id` of her company.

To achieve the same in XSLT, the following two rules have to be specified:

XSLT Example 2

```
<xsl:template match="employee">
  <employee>
    <dept><xsl:copy-of select="../id/text()"/></dept>
    <xsl:copy-of select="text()"/>
  </employee>
</xsl:template>

<xsl:template match="text()"/>
```

The select pattern "`../id/text()`" in the first rule, processing `employee` elements, indicates the navigation steps to be performed to get to the `id` sibling: first going to the father node (as specified by ".."), then descending to the `id` child, and finally descending to its text content.

□

The binary patterns used in Fxt extract more information from the input, similar to how pattern matching with variable bindings extract more information as compared to simple pattern recognition. The binary patterns can be seen in this respect as a particular form of pattern matching with variable bindings tailored to the needs of rule-based XML transformations. The approach of Fxt is more declarative, because rather than specifying *how* to navigate to get to the nodes of interest, one directly captures the nodes of interest with a pattern.

12.2.1.1 Binary Queries for XSLT Implementations

Some results presented in the case of Fxt basically take over to XSLT, when considering the (fairly large) subset of XPath (Core XPath) covered by Fxgrep.

In particular, the relation between a match node and the nodes identified via (unary) select patterns in the node's context can be captured by binary queries with similar benefits as for Fxt (as presented in Section 10.3). This implies, as in the case of Fxt, that pattern matching can be performed efficiently before the actual transformation begins. The pattern evaluation can thus tabulate the information needed for node selection beforehand, s.t. at transformation time, selection may be performed by simply looking up these tables.

XSLT keys are similar to the keyed tables used for grouping in Fxt (presented in Section 9.6), that is, they are a particular form of binary queries. Basically, XSLT keys are pairs consisting of a node and a string value (the node's key). The node is identified using a match pattern, while the value is given by a select pattern evaluated in the context of the node. The usage pattern is completely similar to that of match and select patterns in transformation rules, which can be efficiently implemented as addressed above. Thus, binary queries can also be used to efficiently implement XSLT keys.

12.2.2 Fxt and XQuery

XQuery is another language for transforming XML documents proposed by the W3C Consortium. XQuery version 1.0 [XQu05a], at the time of writing still a working draft, is likely to gain a broad acceptance in the XML processing area. Together with XSLT it belongs already to the most prominent XML transformation languages. While XSLT is rule-based and intended also for non-programmers, XQuery offers similar constructs as SQL, the standard language for extracting information from relational databases, and is more similar to a traditional programming language. Thus XQuery is targeted more at people familiar with programming languages. Even though they compete in achieving the same purposes, it is likely that XSLT and XQuery will be used in parallel, each of them having its advantages for different tasks. While XSLT is more intuitive as a stylesheet language (in so-called *document-oriented applications*), XQuery is especially expressive for typical database operations like joins and sorting (in so-called *data-oriented applications*).

Despite their different processing models, both XQuery and XSLT use XPath as a sub-language. Any XPath pattern is a valid XQuery expression. Like in XSLT, XPath patterns are used as a means of selecting nodes of interest from XML data.

In contrast to XSLT, an XQuery implementation is only required to implement a subset of XPath axes, namely: `child`, `attribute`, `parent`, `self`, `descendant` or `descendant-or-self`. The only reverse axis used is thus `parent`. Since patterns containing the `parent` axis can be rewritten using the other available axes, navigation can be restricted in XQuery strictly downwards the input.

Also unlike in XSLT (at least in the current recommendation, version 1.0 [XSL99]), XPath patterns in XQuery can select nodes not only from the input XML document, but also from XML fragments constructed as intermediate results. This implies that for this type of patterns, evaluation can be only performed along with the transformation. In the following we restrict our observations to the cases in which queries are to be evaluated on XML input data available beforehand. We consider typical usage patterns of XPath queries in XQuery and suggest how techniques already presented in Fxt and Fxgrep can be used to optimize their evaluation.

12.2.2.1 Binary Queries for XQuery implementations

XQuery's basic type is the flat heterogeneous sequence of either simple values (e.g., integers, strings) or XML nodes. In fact, every XQuery expression returns a sequence. In particular, an XPath expression returns the sequence of nodes obtained when the expression is evaluated in the current context.

The XQuery language can be reduced without loss of expressiveness to a small core of constructs [XQu05b]. The fundamental construct in the core of XQuery is the `for` construct which allows iteration over a sequence. The `for` expression has the form:

XQuery Example 1

```
for $x in expression_1
return expression_2
```

For each value t in the sequence given by the evaluation of $expression_1$, $expression_2$ is evaluated in a context in which variable `$x` is bound to t. The result of the `for` expression is given by concatenating the sequences resulting from the evaluations of $expression_2$.

Example 12.3: The expression:

XQuery Example 2

```
for $x in (1,11)
```

```
return ($x+1,$x+2)
```

evaluates to the sequence `(2,3,12,13)`. □

In particular, every step in an XPath pattern can be represented as a `for` expression, where *expression*$_1$ is the set of nodes selected in the previous step (or the root of the input document if this is the first step), and *expression*$_2$ selects the nodes according to the current step. This observation allows the core language of XQuery to represent XPath patterns as a sequence of nested `for`-expressions, each of them representing one step in the pattern.

Example 12.4: The pattern `/a//b/../text()` could be evaluated in the document `"input.xml"` by:

XQuery Example 3

```
for $root in document("input.xml") return
  for $dot1 in $root/a return
    for $dot2 in $dot1//b return
      for $dot3 in $dot2/.. return
        $dot3/text()
```

□

Evaluation of Select Patterns

Nodes from some input document are delivered in XQuery essentially by XPath patterns. An XPath select pattern is either *absolute*, if its evaluation context is the root of some input document, or *relative* if its evaluation context is a previously selected node. Consequently, any sequence of nodes selected from some input document can be constructed by an expression of the form:

XQuery Example 4

```
for $n_1 in /pattern_1 return
  for $n_2 in $n_1/pattern_2 return
    for $n_3 in $n_2/pattern_3 return
      ⋮
        for $n_k in $n_{k-1}/pattern_k return
        return $n_k/pattern_{k+1}
```

That is, a select pattern $pattern_{k+1}$ is evaluated in the context of a set of nodes which in turn have been selected by a pattern $pattern_k$, and so on, with the first set of nodes being selected by an absolute pattern $/pattern_1$.

An expression as in XQuery Example 4 can be efficiently evaluated (for the class of patterns expressible with grammar queries) by a construction using binary grammar queries, as presented below. As mentioned in Section 3.4, this class of XPath queries is quite large, the restriction being basically that no arithmetic and data value comparison can be used in patterns. The construction is similar to that used for removing select patterns in Fxgrep presented in Section 10.3. It implies that at most two traversals of the input document are enough for the evaluation of most XPath patterns in XQuery.

Let $Q_i = ((R_i, E_i), T_i)$ be the grammar query obtained by translating $pattern_i$ for all $i = 1, \ldots k+1$. Given a query Q_{i+1} to be evaluated for each match of a query Q_i, a binary query $Q_{i,i+1} = ((R_{i,i+1}, E_1), T_i \times T_{i+1})$ can be constructed which directly locates every match of Q_i together with the corresponding matches of Q_{i+1}, where $R_{i,i+1}$ is obtained as follows:

$$\begin{aligned} R_{1,2} &= \mathit{combine}((R_1, T_1), (R_2, E_2)) \\ R_{i,i+1} &= \mathit{combine}((R_{i-1,i}, T_i), (R_{i+1}, E_{i+1})) \text{ for all } i > 1 \end{aligned} \tag{12.1}$$

with *combine* defined as in Section 10.3.

Example 12.5: Consider the following instance of the expression in XQuery Example 4 where $\mathit{pattern}_1$=`//a`, $\mathit{pattern}_2$=`//b` and $\mathit{pattern}_3$=`//c`.

XQuery Example 5

```
for $x in //a return
  for $y in $x//b return
    $y//c
```

The corresponding unary queries are

$$\begin{array}{lcl} Q_1 &=& ((\{(1),(2)\},\{\_\ x_1|x_a\ \_\}),\{x_a\}) \\ Q_2 &=& ((\{(3),(4)\},\{\_\ x_2|x_b\ \_\}),\{x_b\}) \\ Q_3 &=& ((\{(5),(6)\},\{\_\ x_3|x_c\ \_\}),\{x_c\}) \end{array}$$

where the productions are as below:

$$\begin{array}{lcll} x_1 & \to & *\langle \_\ x_1|x_a\ \_\rangle & (1) \\ x_a & \to & a\langle \_\rangle & (2) \\ x_2 & \to & *\langle \_\ x_2|x_b\ \_\rangle & (3) \\ x_b & \to & b\langle \_\rangle & (4) \\ x_3 & \to & *\langle \_\ x_3|x_c\ \_\rangle & (5) \\ x_c & \to & c\langle \_\rangle & (6) \\ x_a & \to & a\langle \_\ \&\ \_\ x_2|x_b\ \_\rangle & (7) \\ x_b & \to & b\langle \_\ \&\ \_\ x_3|x_c\ \_\rangle & (8) \end{array}$$

The productions for the constructed binary queries are:

$$\begin{array}{lcl} R_{1,2} &=& \{(1),(2),(3),(4),(7)\} \\ R_{2,3} &=& \{(1),(2),(3),(4),(5),(6),(7),(8)\} \end{array}$$

The query $Q_{1,2} = ((R_{1,2},\{\_\ x_1|x_a\ \_\}),\{(x_a,x_b)\})$ selects each a descendant of the root together with all its b descendants. The query $Q_{2,3} = ((R_{2,3},\{\_\ x_1|x_a\ \_\}),\{(x_b,x_c)\})$ selects each b node which have an a ancestor, together with all its c descendants. □

Given the grammar queries constructed as above, an expression like in XQuery Example 4 can be evaluated as simply as presented in Listing 12.1:

Listing 12.1: Iteration via Binary Queries

```
forall $n_1 \in \mathcal{M}_{Q_1}$
  forall $(n_1, n_2) \in \mathcal{M}_{Q_{1,2}}$
    forall $(n_2, n_3) \in \mathcal{M}_{Q_{2,3}}$
      ⋮
        forall $(n_k, n_{k+1}) \in \mathcal{M}_{Q_{k,k+1}}$
          return $n_{k+1}$
```

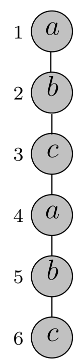


Figure 12.1: Sample input

where $\mathcal{M}_{Q_1}$ are the unary matches of Q_1 and $\mathcal{M}_{Q_{i,i+1}}$ are the binary matches of $Q_{i,i+1}$ for all $i = 1, \ldots, k$.

Thus, by locating the matches $\mathcal{M}_{Q_1}$ and $\mathcal{M}_{Q_{i,i+1}}$ for all i (which can be performed by at most two traversals of the XML input data), one is able to identify quite straightforwardly the set of nodes selected by any XPath pattern as above.

In fact, for the evaluation of an expression like that in XQuery Example 4 only the evaluation of $Q_{k,k+1}$ is needed. However, in general, rather than simply locating the node of the innermost patterns, one needs the variable bindings defined by all patterns as these may be referred to in the original expression in which the patterns occur. For a correct evaluation of the original expression, in which the order of the elements in the resulting sequence correspond to the nested structure in the `for` expression, one must also ensure the correct visibility of the variables n_1 to n_k.

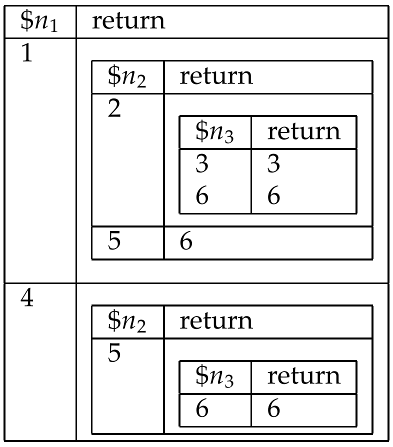
To ensure this, it is enough tabulate the binary matches of $Q_{1,2}$ to $Q_{k,k+1}$ as in the case of binary match patterns in Fxt. That is, a primary node is stored in one entry together with all the secondary nodes with which it forms a match pair. Using this construction, given a binding π_i for n_i, the sequence of bindings for n_{i+1} in the scope of n_i is obtained by looking up the entry for n_i in the tabulated result of $Q_{i,i+1}$ (stored in a table $Tab_{\mathcal{M}_{Q_{i,i+1}}}$). The nested scopes can be implemented by starting with a binding for n_1 to a match for Q_1 (stored in a table in $Tab_{\mathcal{M}_{Q_1}}$) and proceeding as above.

Example 12.6: Consider the XML input graphically represented in Figure 12.1 where nodes are identified via numbers. Figure 12.2 (i) depicts the variable bindings and their scope during the evaluation of the expression in XQuery Example 5 for this input. As depicted in Figure 12.2 (ii) and described above, the nested scopes can be implemented by looking up the tables storing the matches of the initial select pattern and the constructed binary queries.

□

12.2.2.2 XQuery Use-Cases in Fxt

The W3C Consortium has released together with the XQuery specification a set of so-called *XML Query Use Cases* [XQu05c]. XML Query Use Cases aims at providing a representative set of XML transformations. By presenting the proposed solution in XQuery for each of these transformations, these use-cases



$n_1	return
1	(nested table below)
4	(nested table below)

$n_2	return
2	(nested table below)
5	6

$n_3	return
3	3
6	6

$n_2	return
5	(nested table below)

$n_3	return
6	6

(i)

$Tab_{\mathcal{M}_{Q_1}}$:

n_1
1
4

$Tab_{\mathcal{M}_{Q_{1,2}}}$:

n_1	n_2
1	2 5
4	5

$Tab_{\mathcal{M}_{Q_{2,3}}}$:

n_2	n_3
2	3 6
5	6

(ii)

Figure 12.2: Implementing nested scopes

are suited for testing XQuery implementations for their conformance to the XQuery specification.

As the sample transformations are intended to offer a representative overview of the different type of transformations needed for XML documents, it was interesting to use them to compare the expressiveness of Fxt and XQuery. The results were obtained as part of a student project [Eis05]. The described transformations are divided by the specification into classes (*use-cases*), according to the type of task that they achieve.

All use-cases up to two were completely implemented. One of these two cases is the "NS" use case which is concerned with namespace information, currently not supported by Fxgrep and Fxt. The other is the "STRONG" use case, not applicable for Fxt as it deals with type information taken from the schema of the XML input, which is not considered in Fxt. For the more important of the other use cases a brief overview is presented in the following, together with remarks on how they were implemented in Fxt.

The use-case "TREE" covers transformations which basically maintain the structure of the input document. These queries are likely to occur when producing the layout for XML marked-up text. As expected, this class could be implemented without difficulties, since the required top-down transformation model is well suited for the rule-based approach, as suggested by the following example.

Example 12.7: Transformation "Q1" in "TREE" requires producing a hierarchically structured table of contents for an XML document representing a book by listing all the sections and their titles. The solution in Fxt is presented in Fxt Example 32.

Fxt Example 32

```
<fxt:spec>
  <fxt:pat>/book</fxt:pat>
    <toc>
      <fxt:apply/>
    </toc>
  <fxt:pat>//section[%title]</fxt:pat>
    <section>
      <fxt:copyAttributes/>
      <fxt:copyContent select="1"/>
      <fxt:apply/>
    </section>

  <fxt:pat>default</fxt:pat>
</fxt:spec>
```

□

Some transformations in "TREE" require counting the number of nodes fulfilling a certain pattern. One way to achieve this is to use Fxt variables as in Example 12.8.

Example 12.8: The transformation "Q4" in "TREE" requires reporting the number of top-level sections. The solution in Fxt is presented in Fxt Example 33.

Fxt Example 33

```
<fxt:spec>
  <fxt:global name="sec" type="int"/>
  <fxt:push name="sec" val="0"/>


  <fxt:pat>/book</fxt:pat>
    <fxt:apply/>
    <top_section_count><fxt:get name="sec"/></top_section_count>

  <fxt:pat>/book/section</fxt:pat>
    <fxt:inc name="sec"/>

   <fxt:pat>default</fxt:pat>
     <fxt:apply/>

</fxt:spec>
```

A global variable `sec` is used as a counter. The transformation traverses the input top-down and increments the counter at every top level section. The value of the counter is output after processing all children of the root element `book`. □

The use-case "R" contains SQL-like queries on XML views of relational databases in which each table is represented by an XML document. XQuery was specially designed to perform this type of queries. A frequently needed operation is joining two table representations by comparing some data values. As data value comparison is not expressible in Fxgrep, the Fxt transformations have to use SML-code in this case. To perform a data value based join, the data value from the first table is used to construct an Fxgrep pattern with which the elements of interest are selected from the second table, as in the next example.

Example 12.9: The "R" use case deals with a table storing information used by an online auction. The "items.xml" file contains an `item_tuple` element for each item offered in the auction, with an identifier `itemno` and a `description` element. The "bids.xml" file contains a `bid_tuple` element for each offer, with an `itemno` element identifying the item for which the bid was provided. The transformation “Q4” in “R” requires performing a join between the two tables stored in the files “items.xml” and “bids.xml”, by listing item numbers and descriptions of items that have no bids. The solution in Fxt is presented in Fxt Example 34.

Fxt Example 34

```
<fxt:spec>
  <fxt:global name="bidsForItem" type="Forest"/>
  <fxt:push name="bidsForItem" val="emptyForest"/>

  <fxt:pat>/*</fxt:pat>
    <result>
      <fxt:apply/>
    </result>

  <fxt:pat>//item_tuple[%description]/itemno</fxt:pat>
    <fxt:setForest name="bidsForItem">
      <fxt:copyContent file="bids.xml"
       selectExp='String2Vector(
       "//itemno/\""^(Vector2String (getTextContent current))^"\"")'/>
    </fxt:setForest>

    <fxt:if test='(Globals.get G.bidsForItem) = emptyForest'>
      <no_bid_item>
        <fxt:current/>
        <fxt:copyContent select="1"/>
      </no_bid_item>
    </fxt:if>

  <fxt:pat>default</fxt:pat>
    <fxt:apply/>

</fxt:spec>
```

The transformation is to be applied on the “items.xml” file. A global variable `bidsForItem` is used to store the bids for a given item. The first rule produces an enclosing `result` element containing the required items. The second rule handles the items in “items.xml” identified by their `itemno`. When processing an `itemno`, an Fxgrep pattern is constructed via an SML expression in the attribute `selectExp` which selects the references to the current item in the “bids.xml” table. The result of this selection is stored in the variable `bidsForItem` via the `fxt:setForest` element. If the value of the variable equals the empty forest (as tested by the `fxt:if` element), then no bids were found and the item is output as required. □

The use case “XMP” is meant to cover typical transformations for XML input representing both marked up text documents and database XML views and could be implemented quite straightforwardly. A few transformations performing data value based joins have been implemented again using SML code. Also, to compute aggregation functions like the minimum, sum or mean of a sequence of selected values, SML code had to be used in the Fxt implementation.

The use case “SEQ” exploits the relative order of elements of elements in a document and could be implemented in Fxt without difficulties since Fxgrep is especially good at specifying contextual conditions. To account for the ordinal number of nodes in a selection, Fxt variables were used.

The use case “STRING”, uses string searching capabilities in XML documents. These capabilities were for the most part covered by the Fxgrep match patterns used.

The implementation of the XQuery use-cases shows that most of the usual XML transformations are expressible in Fxt. As a rule-based language, Fxt can easily specify stylesheet transformations, similarly to XSLT but possibly in a more declarative and efficiently implementable manner. Performing data value comparisons as required by joins in database-like applications is achievable, though one has to resort to embedding SML code into transformations. The same holds true in the cases where different aggregation functions are needed. This justifies the adequacy of the design decision of Fxt to allow one to embed SML code in transformations, to account for eventual limitations of the pattern language or unanticipated but necessary functionality.

12.2.3 XPath 2.0 and XSLT 2.0

At the time of writing, the W3C consortium is working at a successor specification for XPath, XPath 2.0 [XPa05], currently having the status of a W3C Working Draft. XPath 2.0 is a superset of XPath 1.0 and loosely speaking a subset of XQuery. In fact, most of the text in the specifications of XQuery and XPath 2.0 is the same.

As opposed to version 1.0, XPath 2.0 is strictly speaking not a pattern language but rather a transformation language, as it not only identifies parts of input XML documents, but also is able to restructure the input data. One extension as compared to XPath 1.0 is that, like XQuery, XPath 2.0 provides `for` expressions for iterations over sequences. The observations regarding XPath 1.0 patterns and `for` expressions in XQuery thus apply to XPath 2.0 as well. As opposed to XQuery, XPath 2.0 does not provide constructors for XML elements, meaning that nodes which are selected are always from the original input, as assumed in our remarks.

Like XQuery, XPath 2.0 may use type information that becomes available when the input documents are validated using an associated instance of an XML Schema [XML01]. The type of a node influences the way it is handled within the evaluation of an expression.

XSLT version 2.0 [XSL03] is a rule-based transformation language using XPath 2.0 as a pattern language. Its match patterns are still the restriction of XPath 2.0 expressions to XPath 1.0 match patterns while its select patterns can be XPath 2.0 expressions evaluating to sequences of nodes. Implementation techniques for evaluating XPath 2.0 queries as mentioned above are thus useful also in XSLT 2.0.

12.3 Summary

One possibility of providing convenient tools for XML processing is to extend existing GPLs with XML-specific constructs. Since, traditionally, object-oriented GPLs have been mainly used for XML processing, there are a few approaches in which object-oriented GPLs are used as the basis of embedded DSLs for XML, which we briefly reviewed. Furthermore, we considered some approaches which attempt to provide highly desirable features of functional programming languages, like absence of side-effects, pattern-matching or extensive static type checking in an XML processing setting. We have briefly remarked how techniques for query evaluation which proved their robustness in Fxgrep and Fxt could also be used as a basis for efficient implementations of pattern matching primitives in functional programming languages specialized for XML.

By using an XML syntax for XML constructs and pattern matching capabilities (either as simple as XPath for concisely addressing parts of XML trees, or more elaborated patterns which extract more information from the data), embedded DSLs highly improve the convenience of writing and the readability of programs, yet, as such, they are exclusively targeted at programmers of the underlying GPLS.

As an alternative accessible also to non-programmers, XML processing is already primarily performed via stand-alone DSLs which can be used by solely manipulating XML concepts. We compared Fxt as a stand-alone DSL to the most prominent languages in the same category: XSLT and XQuery. A distinguishing feature of Fxt among XML transformation tools is its use of efficiently implemented binary queries. In contrast, popular languages like XSLT and XQuery only use unary queries. Nevertheless, typical usage scenarios combine unary queries in such a way that they can be internally implemented via binary queries, and benefit from them similarly to Fxt. This was sustained by the successful implementation of an important part of the XML sample transformations, representative for the purposes of XQuery.

Conclusion

We have presented querying techniques for unranked trees based on (extensions of) tree grammars and tree automata and showed how these can be used to support declarative and efficient pattern and transformation languages for XML data. The contributions of this work are summarized in the following.

An expressive method for specifying k-ary queries We have introduced a simple yet powerful method based on forest grammars allowing the formulation of queries which identify tuples of k related nodes in the input document tree.

Efficient evaluation of binary queries Unary grammar queries have previously been shown to be efficiently implementable. We have shown here that binary queries can also be efficiently evaluated by providing an algorithm based on pushdown forest automata. Binary queries are a fundamental feature of the XML querying tool Fxgrep and of Fxt, an XML transformation tool built upon Fxgrep. To ensure the reliability of the query evaluation in these XML tools, we have proven the correctness of the algorithm for evaluating binary queries.

Evaluation of k-ary queries We have mentioned how the algorithm for answering binary queries can be generalized for the evaluation of queries for arbitrary arity. The complexity of query evaluation grows exponentially with k. Nevertheless, we have suggested restrictions for k-ary grammar queries under which their evaluation is efficiently implementable.

Event-driven evaluation of grammar queries on XML streams We have presented an efficient algorithm which allows the evaluation of unary grammar queries in an event-based manner. This processing method is suitable for very large documents which cannot be built completely in memory. The algorithm is also useful in settings in which documents are received linearly on some communication channel and would ideally be processed while being received, rather than waiting until the whole document is locally available.

Implementation in Fxgrep The querying techniques introduced here have been implemented in the XML querying tool Fxgrep. Fxgrep makes it possible to express powerful contextual conditions, as permitted by grammar queries, via an intuitive pattern language. The practical implementation confirms the efficiency of the introduced algorithms.

A rule-based transformation language based on grammar queries Fxgrep has been used as the basis of Fxt, a rule-based transformation language for XML documents. Fxt allows the specification of common, simple transformations in an intuitive and concise way. More complex transformations are possible via a variable mechanism, as well as via a functional programming interface which allows the embedding of arbitrary SML code into transformations.

Binary queries in XML transformations We have identified binary queries as an especially useful case of queries in XML transformations. Given the efficient evaluation of binary grammar queries, we use them in Fxt both at the specification and at the implementation level. As a means of specification, Fxgrep binary patterns allow one to identify together the nodes to which a rule is applicable and the nodes to be selected in the body of the rule. In the implementation, user-specified, unary selection patterns are automatically replaced by more efficiently implementable binary queries. This ensures that the number of traversals of the input performed for purposes of pattern matching is always limited to two. Besides efficiency, binary queries also improve the expressiveness and declarativeness of XML transformation languages as presented in the case of Fxt and as suggested for other popular XML processing languages.

Bibliography

[ACGG$^+$02] Iliana Avila-Campillo, Todd J. Green, Ashish Gupta, Dan Suciu, and Makoto Onizuka. XMLTK: An XML Toolkit for Scalable XML Stream Processing. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2002. PLAN-X 2002.

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Procedings of the 28th International Conference on Very Large Data Bases (VLDB 2000)*, September 2000.

[AM94] Andrew W. Appel and David B. MacQueen. Separate Compilation for Standard ML. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 1994.

[BBS03] Sacha Berger, François Bry, and Sebastian Schaffert. A Visual Language for Web Querying and Reasoning. In *Proceedings of Workshop on Principles and Practice of Semantic Web Reasoning, Mumbai, India (9th–13th December 2003)*, volume 2901 of *LNCS*, 2003.

[BC05] Mikolaj Bojanczyk and Thomas Colcombet. Tree-Walking Automata Do Not Recognize All Regular Languages. In *Proceedings Of The ACM Symposium on Theory of Computing, STOC 2005*, 2005.

[BCD$^+$05] Steven Bird, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng. Extending XPath to Support Linguistic Queries. In *Workshop on Programming Language Technologies for XML (PLAN-X) 2005*, 2005.

[BCF03] Veronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric General-purpose Language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 51–63. ACM Press, 2003.

[Bec03] Oliver Becker. Transforming XML on the Fly. In *XML Europe 2003*, 2003.

[BKMW01] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular Tree and Regular Hedge Languages over Non-Ranked Alphabets. Research report, HKUST Theoretical Computer Science Center, May 2001.

[BKR05] Nick Benton, Andrew Kennedy, and Claudio Russo. SML.NET. Software and Documentation, University of Cambridge, 2005.

[BKW00] Anne Brüggemann-Klein and Derick Wood. Caterpillars: A Context Specification Technique. Research Report TCSC-2000-8, HKUST Theoretical Computer Science Center, January 2000.

[Blu97] Matthias Blume. CM – A Compilation Manager for SML/NJ. User Manual, Princeton University, 1997.

[BMS02] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` Project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[BMS03] G.M. Bierman, E. Meijer, and W. Schulte. Programming with Circles, Triangles and Rectangles. In *Proceedings of XML 2003*, 2003.

[Bos99] The Complete Plays of Shakespeare Marked up in XML, 1999.

[BRS$^+$05] Didier Le Botlan, Andreas Rossberg, Christian Schulte, Gert Smolka, and Guido Tack. Alice Manual. Online Documentation, Saarland University, 2005.

[BS86] Gérard Berry and Ravi Sethi. From Regular Expressions to Deterministic Automata. *Theoretical Computer Science Journal*, 48:117–126, 1986.

[BS02] Alexandru Berlea and Helmut Seidl. Binary Queries. In *Extreme Markup Languages 2002*, August 2002.

[BS04] Alexandru Berlea and Helmut Seidl. Binary Queries for Document Trees. *Nordic Journal of Computing*, 11(1):41–71, March 2004.

[BYFJ04] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the Memory Requirements of XPath Evaluation over XML Streams. In *Proceedings of the 20th Symposium on Principles of Database Systems (PODS 2004)*, 2004.

[CCD$^+$99] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A graphical language for querying and restructuring XML documents. In *Sistemi Evoluti per Basi di Dati*, pages 151–165, 1999.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[CDG$^+$05] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2005. Release April, 6th 2005.

[CDTW00] Jianjun Chean, David J. DeWitt, Feng Tian, and Yang Wang. NiagaraCQ: a Scalable Continuous Query System for Internet Databases. In *Proceedings of the International Conference on Management of Data (SIGMOD 2000)*, 2000.

[CEKL05] Woongsik Choi, Hyunjun Eo, Jungtaek Kim, and Oukseh Lee. nML Programming Language System. Online Documentation, National Creative Research Initiatives Center, Korea, 2005.

[CFGR02] Chee-Yong Chan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the International Conference on Data Engineering (ICDE 2002)*, 2002.

[CML03] Chemical Markup Language. http://wwmm.ch.cam.ac.uk/moin/, 2003.

[CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for High-level Web Service Construction. *ACM Trans. Program. Lang. Syst.*, 25(6):814–875, 2003.

[Con96] The Unicode Consortium. *The Unicode Standard, Version 2.0*. Addison Wesley Developers Press, Reading, Massachusetts, 1996.

[Con03] The Unicode Consortium. *The Unicode Standard, Version 4.0*. Addison Wesley Developers Press, Boston, Massachusetts, 2003.

[CR04] Cristiana Chitic and Daniela Rosu. On Validation of XML Streams using Finite State Machines. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 85–90, New York, NY, USA, 2004. ACM Press.

[Csh05] C # Language Specification. http://msdn.microsoft.com/library/, 2005.

[DBL05] Computer Science Bibliography. http://dblp.uni-trier.de/, 2005.

[Des01] Arpan Desai. Introduction to Sequential XPath. In *XML Conference 2001*, December 2001.

[DF03] Yanlei Diao and Michael Franklin. YFilter: Query Processing for High-Volume XML Message Brokering. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*, 2003.

[DFFT02] Yanlei Diao, Peter Fischer, Michael J. Franklin, and Raymond To. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of the International Conference on Data Engineering (ICDE 2002)*, February 2002.

[DKV00] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[DOM98] Document Object Model (DOM) Level 1 Specification, Version 1.0, October 1998.

[Eis05] Volker Eiseler. XML Query Use Cases in Fxt, February 2005. Student Project, Technische Universität München.

[Erw03] Martin Erwig. Xing: a Visual XML Query Language. *Journal of Visual Languages and Computing*, 14(1):5–45, 2003.

[Feg04] Leonidas Fegaras. The Joy of SAX. In *Proceedings of the First International Workshop on XQuery Implementation, Experience and Perspectives*, June 2004.

[FGK03] Markus Frick, Martin Grohe, and Cristoph Koch. Query Evaluation on Compressed Trees. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*, pages 188–197, 2003.

[Fou05] Free Software Foundation. Gnu grep. Software and Documentation, 2005.

[Fri04] Alain Frisch. Regular Tree Language Recognition with Static Information. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

[FS98] Mary Fernández and Dan Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of the International Conference on Data Engineering (ICDE 1998)*, 1998.

[FSW99] Mary Fernandez, Jerome Siméon, and Philip Wadler. XML Query Languages: Experiences and Exemplars. Draft manuscript: http://www.w3.org/1999/09/ql/docs/xquery.html, September 1999.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB 2002)*, pages 95–106, Hong Kong, China, 2002. Morgan Kaufmann.

[GKP03] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The Complexity of XPath Query Evaluation. In *Proceedings of the Eighteenth Symposium on Principles of Database Systems (PODS 2003)*, June 2003.

[GKS04] G. Gottlob, C. Koch, and K. Schulz. Conjunctive Queries over Trees. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2004)*, 2004.

[GLPS05] Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. The Xtatic Experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, January 2005.

[GMOS03] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata. In *Proceedings of International Conference on Database Theory (ICDT 2003)*, pages 173–189, 2003.

[GS97] Ferenc Gécseg and Magnus Steinby. Tree Languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1, pages 1–68. Springer, Heidelberg, 1997.

[GS03] Ashish Gupta and Dan Suciu. Stream Processing of XPath Queries with Predicates. In *Proceedings of the International Conference on Management of Data(SIGMOD 2003)*, 2003.

[Har99] 1998 Baseball Statistics – XML Sample Files. http://metalab.unc.edu/xml/examples/1998validstats.xml, 1999.

[Har01] Elliotte Rusty Harold. *XML Bible*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[HLPR94] R. Harper, P. Lee, F. Pfenning, and E. Rollins. A Compilation Manager for Standard ML of New Jersey. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, July 1994.

[HP00] Haruo Hosoya and Benjamin C. Pierce. XDuce: A Typed XML Processing Language. In *Proceedings Of The Third International Workshop on the Web and Databases (WebDB2000), Dallas, Texas*, pages 111–116, May 2000.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.

[HRS$^+$05] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML Processing in Java. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, pages 278–287, New York, NY, USA, 2005. ACM Press.

[Int86] International Organization for Standardization. Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML). Ref. No. ISO 8879:1986 (E). Geneva/New York, 1986.

[Int96] International Organization for Standardization. Information technology – Processing Languages – Document Style Semantics and Specification Language (DSSSL). Ref. No. ISO/IEC 10179:1996(E), 1996.

[JFB05] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML Streams. *The VLDB Journal*, 14(2):197–210, 2005.

[Kay05] Michael Kay. Saxon. Software Documentation, 2005.

[KL03] Martin Kempa and Volker Linnemann. Type Checking in XOBE. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *Proceedings of Datenbanksysteme für Business, Technologie und Web (BTW), 10. GI-Fachtagung,*, volume P-26 of *Lecture Notes in Informatics*, pages 265–274. Gesellschaft für Informatik, 26.-28. Februar 2003.

[KL04] Martin Kempa and Volker Linnemann. Type Safe Programming of XML-based Applications. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *Proceedings der 34. GI-Jahrestagung, 3. Arbeitstagung Programmiersprachen und Rechenkonzepte (ATPS 2004)*, volume P-51 of *Lecture Notes in Informatics*, pages 397–407, Ulm, 20.-24. September 2004. Gesellschaft für Informatik.

[KMS00] N. Klarlund, A. Møller, and M. Schwartzbach. DSD: A Schema Language for XML. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*, August 2000.

[KMS04] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static Analysis of XML Transformations in Java. *IEEE Trans. Softw. Eng.*, 30(3):181–192, 2004.

[KS03] Christoph Koch and Stefanie Scherzinger. Attribute Grammars for Scalable Query Processing on XML Streams. In *Database Programming Languages (DBPL)*, pages 233–256, 2003.

[KSSS04] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based Scheduling of Event-Processors and Buffer Minimization for Queries on Structured Data Streams. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*, 2004.

[LDG$^+$05] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml System,Documentation and User's Guide. Online Documentation, I.N.R.I.A., France, 2005.

[Leg02] LegalXML. http://www.legalxml.org/, 2002.

[LMP02] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, 2002.

[Mat03] Mathematical Markup Language (MathML) Version 2.0 (Second Edition). http://www.w3.org/TR/2003/REC-MathML2-20031021/, October 2003.

[MBPS05] Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. XML Type Checking with Macro Tree Transducers. In *Proceedings of the 20th Symposium on Principles of Database Systems (PODS 2005)*, 2005.

[Mic05] Sun Microsystems. Java API for XML Processing (JAXP). Software Documentation, 2005.

[MLM01] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML Schema Languages Using Formal Language Theory. In *Extreme Markup Languages 2001, Montreal, Canada*, August 2001.

[MP00] Kevin D. Munroe and Yannis Papakonstantinou. BBQ: A Visual Interface for Integrated Browsing and Querying of XML. In *VDB 5: Proceedings of the Fifth Working Conference on Visual Database Systems*, pages 277–296, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[MS99] Erik Meijer and Mark Shields. XMλ: A Functional Language for Constructing and Manipulating XML Documents. (Draft), 1999.

[MS01] Holger Meuss and Klaus U. Schulz. Complete Answer Aggregates for Tree-like Databases: a Novel Approach to Combine Querying and Navigation. *ACM Transactions on Information Systems*, 19(2):161–215, 2001.

[MS05] Anders Møller and Michael I. Schwartzbach. The Design Space of Type Checkers for XML Transformation Languages. In *Proc. Tenth International Conference on Database Theory, ICDT '05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.

[MSW$^+$05] Holger Meuss, Klaus U. Schulz, Felix Weigel, Simone Leonardi, and François Bry. Visual Exploration and Retrieval of XML Document Collections with the Generic System X^2. *Journal on Digital Libraries, Special Issue on "Information Visualization Interfaces for Retrieval and Analysis"*, 5, 2005.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Mur97] Makoto Murata. Transformation of Documents and Schemas by Patterns and Contextual Conditions. In *Principles of Document Processing '96*, volume 1293, pages 153–169. Springer-Verlag, 1997.

[Mur01] Makoto Murata. Extended Path Expressions for XML. In *Proceedings of the 17th Symposium on Principles of Database Systems (PODS 2001)*, 2001.

[NB02] Frank Neven and Jan Van Den Bussche. Expressiveness of Structured Document Query Languages Based on Attribute Grammars. *Journal of the ACM*, 49(1):56–100, 2002.

[NB05] Andreas Neumann and Alexandru Berlea. fxgrep 4.6.1. http://www2.informatik.tu-muenchen.de/~berlea/Fxgrep/, 2005.

[Neu00] Andreas Neumann. *Parsing and Querying XML Documents in SML*. PhD thesis, University of Trier, Trier, 2000.

[Neu04] Andreas Neumann. fxp 1.4.6. http://www2.informatik.tu-muenchen.de/~berlea/Fxp/, 2004.

[Nev05] Frank Neven. Extensions of Attribute Grammars for Structured Document Queries. *Journal of Computer and System Sciences*, 70(2):221–257, 2005.

[NN01] Keisuke Nakano and Susumu Nishimura. Deriving Event-Based Document Transformers from Tree-Based Specifications. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[Nov01] Dimitre Novatchev. The Functional Programming Language XSLT - A Proof Through Examples. Online article, TopXML, 2001.

[NPTT05] Joachim Niehren, Laurent Planque, Jean-Marc Talbot, and Sophie Tison. N-ary Queries by Tree Automata. (Manuscript), 2005.

[NS98a] Andreas Neumann and Helmut Seidl. Locating Matches of Tree Patterns in Forests. In V. Arvind and R. Ramamujan, editors, *Foundations of Software Technology and Theoretical Computer Science, (18th FST&TCS)*, volume 1530 of *Lecture Notes in Computer Science*, pages 134–145, Heidelberg, 1998. Springer.

[NS98b] Andreas Neumann and Helmut Seidl. Locating Matches of Tree Patterns in Forests. Technical Report 98-08, Mathematik/Informatik, Universität Trier, 1998.

[NS99] Frank Neven and Thomas Schwentick. Query Automata. In *Proceedings of the Eighteenth Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 205–214. ACM Press, 1999.

[NS00] Frank Neven and Thomas Schwentick. Expressive and Efficient Pattern Languages for Tree-structured Data. In *Proceedings of the Nineteenth Symposium on Principles of Database Systems 2000*, pages 145–156. ACM Press, 2000.

[NS02] Frank Neven and Thomas Schwentick. Query Automata over Finite Trees. *Theoretical Computer Science Journal*, 275(1-2):633–674, 2002.

[NS03] Frank Neven and Thomas Schwentick. Automata- and logic-based pattern languages for tree-structured data. In K.-D. Schewe L. Bertossi, G. Katona and B. Thalheim, editors, *Semantics in Databases*, volume 2582 of *Lecture Notes in Computer Science*, pages 160–178, Heidelberg, 2003. Springer.

[NSV01] Frank Neven, Thomas Schwentick, and Victor Vianu. Towards Regular Languages over Infinite Alphabets. In *MFCS '01: Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science*, pages 560–572, London, UK, 2001. Springer-Verlag.

[OAS01] RelaxNG Specification. http://www.relaxng.org/, 2001.

[OFB04] Dan Olteanu, Tim Furche, and François Bry. Evaluating Complex Queries against XML Streams with Polynomial Combined Complexity. In *Proc. of 21st Annual British National Conference on Databases (BNCOD21)*, July 2004.

[OMFB02] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *Proc. of Workshop on XML-Based Data Management (XMLDM) at EDBT 2002*, March 2002.

[Org02] OpenOffice.org Organization. OpenOffice.org XML File Format, Technical Reference Manual. http://xml.openoffice.org, 2002.

[Par01] Bijan Parsia. Functional Programming and XML. Online Article, XML.com, 2001.

[PC03] Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data. In *Proceedings of the International Conference on Management of Data(SIGMOD 2003)*, 2003.

[Pit96] Richard Pito. Tgrep Documentation. http://www.ldc.upenn.edu/ldc/online/treebank/, 1996.

[Pro05] Apache XML Project. Xalan-java 2.6.0. Software Documentation, 2005.

[PSD] Protein Information Ressource: The Protein Sequence Database. http://pir.georgetown.edu/home.shtml.

[Roh04] Douglas L. T. Rohde. Tgrep2 User Manual. tedlab.mit.edu/ dr/Tgrep2/tgrep2.pdf, 2004.

[RSS03] RSS 2.0 Specification. http://blogs.law.harvard.edu/tech/rss, July 2003.

[SAS98] D. S. Swierstra, P. Azero, and J. Saraiva. Designing and Implementing Combinator Languages. In D. S. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming, Braga, Portugal, 1998*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206, Heidelberg, 1998. Springer.

[SAX98] SAX 1.0: The Simple API for XML. http://www.megginson.com/SAX/index.html, May 1998.

[Sch00] Thomas Schwentick. On Diving into Trees. In *Proceedings of the 25-th Symposium on Mathematical Foundations of Computer Science 2000*, pages 660–669. ACM Press, 2000.

[Ses05] Peter Sestoft. Moscow ML. Software and Documentation, Royal Veterinary and Agricultural University, Copenhagen, Denmark, 2005.

[SK05] Stefanie Scherzinger and Alfons Kemper. Syntax-directed Transformations of XML Streams. In *Workshop on Programming Language Technologies for XML (PLAN-X) 2005*, 2005.

[SKK04] Bernhard Stegmaier, Richard Kuntschke, and Alfons Kemper. StreamGlobe: Adaptive Query Processing and Optimization in Streaming P2P Environments. In *Proceedings of the International Workshop on Data Management for Sensor Networks*, pages 88–97, 2004.

[SM01] Mark Shields and Erik Meijer. Type-indexed Rows. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 261–275. ACM Press, 2001.

[SMI98] Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. http://www.w3.org/TR/REC-smil/, June 1998.

[SML05] Standard ML of New Jersey. http://www.smlnj.org/, 2005.

[SSM03] Helmut Seidl, Thomas Schwentick, and Anca Muscholl. Numerical Document Queries. In *Proceedings of the Eighteenth Symposium on Principles of Database Systems (PODS 2003)*, pages 155–166, June 2003.

[SV02] Luc Segoufin and Victor Vianu. Validating Streaming XML Documents. In *Symposium on Principles of Database Systems*, pages 53–64, 2002.

[SVG03] Scalable Vector Graphics (SVG) 1.1 Specification. http://www.w3.org/TR/SVG/, January 2003.

[TW68] J.W. Thatcher and J.B. Wright. Generalized Finite Automata with an Application to a Decision Problem of Second Order Logic. *Mathematical Systems Theory*, 2:57–82, 1968.

[vB98] Jan-Pascal van Best. Tree-Walking Automata and Monadic Second Order Logic, 1998. Master's Thesis, TU Delft.

[Voi04] Voice Extensible Markup Language (VoiceXML) Version 2.0. http://www.w3.org/TR/voicexml20/, March 2004.

[Wee05] Stephen Weeks. MLton. Software and Documentation, mlton.org, 2005.

[WML01] Wireless Markup Language Specification Version 2.0. http://www.openmobilealliance.org/tech/affiliates/wap/wap-238-wml-20010911-a.pdf, September 2001.

[WR99] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic Combinators or Type-based Translation? In Peter Lee, editor, *Proceedings Of The International Conference on Functional Programming 1999, Paris, France*, pages 148–259. ACM Press, New York, Sept 1999.

[XHT02] XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). www.w3.org/TR/xhtml1/, August 2002.

[XMI03] XML Metadata Interchange (XMI) Specification. http://www.omg.org/cgi-bin/doc?formal/2003-05-02, 2003.

[XML98a] Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml/, February 1998.

[XML98b] Common XML Pull API. http://www.xmlpull.org/, May 1998.

[XML01] XML Schema Language. http://www.w3.org/TR/xmlschema-0/, May 2001.

[XML04] Extensible Markup Language (XML) 1.1. http://www.w3.org/TR/2004/REC-xml11-20040204/, February 2004.

[XPa99] XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, November 1999.

[XPa05] XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20, February 2005.

[XQu05a] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, February 2005.

[XQu05b] XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/query-semantics/, February 2005.

[XQu05c] XML Query Use Cases. http://www.w3.org/TR/query-semantics/, February 2005.

[XSL99] XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, November 1999.

[XSL01] Extensible Stylesheet Language (XSL). http://www.w3.org/TR/xsl, October 2001.

[XSL03] XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20, November 2003.

Appendix A

Proofs

A.1 Proof of Theorem 5.3

We start by showing that if a derivation f' of a forest f labels a node π with x, then the trees $f[\pi]$ and $f'[\pi]$ are in the derivation relation $\mathcal{D}eriv_x$.

Lemma A.1: If $(f, f') \in \mathcal{D}eriv_r$ and $lab(f'[\pi]) = x$ then $(f[\pi], f'[\pi]) \in \mathcal{D}eriv_x$.

The proof is by induction on the length of π.

Let $\pi = i$ and $last_f(\lambda) = n$. Thus $f = f[1] \dots f[n]$ and $f' = f'[1] \dots f'[n]$. From the definition of $\mathcal{D}eriv_r$ it follows that there is some $x_1 \dots x_n \in [\![r]\!]$ with $(f[k], f'[k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. In particular $(f[i], f'[i]) \in \mathcal{D}eriv_{x_i}$.

Now let $\pi = \pi_1 i$, $last_f(\pi_1) = n$ and let $lab(f[\pi_1]) = a$, $lab(f'[\pi_1]) = x'$. By the induction hypothesis, $(f[\pi_1], f'[\pi_1]) \in \mathcal{D}eriv_{x'}$. By the definition of $\mathcal{D}eriv_{x'}$ there is some $x' \rightarrow a\langle r_1 \rangle \in R$ with $(f[\pi_1 1] \dots f[\pi_1 n], f'[\pi_1 1] \dots f'[\pi_1 n]) \in \mathcal{D}eriv_{r_1}$. By the definition of $\mathcal{D}eriv_{r_1}$ there is some $x_1 \dots x_n \in [\![r_1]\!]$ with $(f[\pi_1 k], f'[\pi_1 k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. In particular $(f[\pi_1 i], f'[\pi_1 i]) \in \mathcal{D}eriv_{x_i}$.

In either case, from $(f[\pi], f'[\pi]) \in \mathcal{D}eriv_{x_i}$ it follows by the definition of $\mathcal{D}eriv_{x_i}$ that $x_i = lab(f'[\pi]) = x$. □

In the following we show that if a derivation f' of a forest f labels a node π with x, and there is a derivation t' of the tree $f[\pi]$ from the same x, then we obtain another derivation of f' by grafting t' into f' at π.

Lemma A.2: Assume $(f, f') \in \mathcal{D}eriv_r$, $lab(f'[\pi]) = x$ and $(f[\pi], t') \in \mathcal{D}eriv_x$. Then $(f, f'/^\pi t') \in \mathcal{D}eriv_r$.

The proof is by induction on the length of π.

If $\pi = i$ then let $f = t_1 \dots t_n$ and let $(t_1 \dots t_i \dots t_n, t'_1 \dots t'_i \dots t'_n) \in \mathcal{D}eriv_r$. By the definition of $\mathcal{D}eriv_r$ there is some $x_1 \dots x_n \in [\![r]\!]$ with $(t_k, t'_k) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. Since $t'_i = f'[i] = x\langle \_ \rangle$ it follows that $x_i = x$. From $(f[i], f'[i]) \in \mathcal{D}eriv_{x_i}$ we have that $(t_1 \dots t_i \dots t_n, t'_1 \dots t' \dots t'_n) \in \mathcal{D}eriv_r$ which is $(f, f'/^i t') \in \mathcal{D}eriv_r$.

If $\pi = i j \pi_1$ we have that $(f[1] \dots f[i] \dots f[n], f'[1] \dots f'[i] \dots f'[n]) \in \mathcal{D}eriv_r$. By the definition of $\mathcal{D}eriv_r$ there is some $x_1 \dots x_n \in [\![r]\!]$ with $(f[k], f'[k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \dots, n$. From $(f[i], f'[i]) \in \mathcal{D}eriv_{x_i}$ it follows that $f[i] = a\langle f_1 \rangle$, $f'[i] = x_i\langle f'_1 \rangle$ and there is $x_i \rightarrow a\langle r_1 \rangle \in R$

and $(f_1, f'_1) \in \mathcal{D}eriv_{r_1}$. As $f_1[j\pi_1] = f[ij\pi_1]$ and $f'_1[j\pi_1] = f'[ij\pi_1]$ we have that $(f_1[j\pi_1], t') \in \mathcal{D}eriv_x$ and $f'_1[j\pi_1] = x\langle\_\rangle$. It follows by the induction hypothesis that $(f_1, f'_1/^{j\pi_1} t') \in \mathcal{D}eriv_{r_1}$. By the definition of $\mathcal{D}eriv_{x_i}$, $(a\langle f_1\rangle, x_i\langle f'_1/^{j\pi_1} t'\rangle) \in \mathcal{D}eriv_{x_i}$ which is $(f[i], x_i\langle f'_1/^{j\pi_1} t'\rangle) \in \mathcal{D}eriv_{x_i}$. Therefore, $(f[1]\ldots f[i]\ldots f[n], f'[1]\ldots x_i\langle f'_1/^{j\pi_1} t'\rangle \ldots f'[n]) \in \mathcal{D}eriv_r$ which is $(f, f'/^{ij\pi_1} t') \in \mathcal{D}eriv_r$. □

Now we show that the forest obtained by grafting t into f at π has the nodes below π labeled as in t and all other nodes as in f.

Lemma A.3:

$$lab((f/^{\pi} t)[\pi_1]) = \begin{cases} lab(t[1\pi_2]), & \text{if } \pi_1 = \pi\pi_2 \\ lab(f[\pi_1])\ , & \text{otherwise} \end{cases}$$

First, observe the definition of the subtree located in a grafted forest:

$$(f/^{i\pi_1} t)[j\pi_2] = \begin{cases} f[j\pi_2] & , \text{ if } i \neq j \\ t[1\pi_2] & , \text{ if } i = j, \pi_1 = \lambda \\ a\langle f_1/^{\pi_1} t\rangle & , \text{ if } i = j, \pi_1 \neq \lambda, \pi_2 = \lambda, f[i] = a\langle f_1\rangle \\ (f_1/^{\pi_1} t)[\pi_2], & \text{if } i = j, \pi_1 \neq \lambda, \pi_2 \neq \lambda, f[i] = a\langle f_1\rangle \end{cases}$$

The proof is by induction on the length of π.

If $\pi = i$ then if $\pi_1 = i\pi_2$, $(f/^{i} t)[\pi_1] = t[1\pi_2]$ thus $lab((f/^{i} t)[\pi_1]) = lab(t[1\pi_2])$. If $\pi_1 = j\pi_2$, $j \neq i$ then $(f/^{i} t)[\pi_1] = f[\pi_1]$ thus $lab((f/^{i} t)[\pi_1]) = lab(f[\pi_1])$.

We consider now the case where $\pi = i\pi'$, $\pi' \neq \lambda$.

If $\pi_1 = i\pi_2$ then $(f/^{\pi} t)[\pi_1] = (f_1/^{\pi'} t)[\pi_2]$, where f[i]=$a\langle f_1\rangle$. If $\pi_1 = \pi\pi_3$, i.e. if $i\pi_2 = i\pi'\pi_3$, $\pi_2 = \pi'\pi_3$ then by the induction hypothesis $lab((f_1/^{\pi'} t)[\pi_2]) = lab(t[1\pi_3])$. Thus $lab(f/^{i\pi'} t)[i\pi_2]) = lab(t[1\pi_3])$ and therefore we obtain that $lab(f/^{\pi} t[\pi\pi_3]) = lab(t[1\pi_3])$ as required.

Otherwise, also by the induction hypothesis $lab((f_1/^{\pi'} t)[\pi_2]) = lab(f_1[\pi_2])$. Since $f_1[\pi_2] = f[i\pi_2] = f[\pi_1]$ it follows that $lab((f/^{\pi} t)[\pi_1]) = lab(f[\pi_1])$.

If $\pi_1 = j\pi_2$ and $j \neq i$ then $(f/^{\pi} t)[\pi_1] = f[\pi_1]$ thus $lab((f/^{\pi} t)[\pi_1]) = lab(f[\pi_1])$. □

Using the lemmas above we prove now Theorem 5.3.

Let $lab(f_1[\pi]) = lab(f_2[\pi]) = x$. By Lemma A.1 we have that $(f[\pi], f_2[\pi]) \in \mathcal{D}eriv_x$. From Lemma A.2 it follows that $(f, f_1/^{\pi} f_2[\pi]) \in \mathcal{D}eriv_r$. By Lemma A.3:

$$lab((f_1/^{\pi} f_2[\pi])[\pi_1]) = \begin{cases} lab(f_2[\pi][1\pi_2]), & \text{if } \pi_1 = \pi\pi_2 \\ lab(f[\pi_1]) & , \text{ otherwise} \end{cases}$$

With $f_2[\pi][1\pi_2] = f_2[\pi\pi_2]$ we obtain now the result of our theorem.

A.2 Proof of Theorem 5.2

We start by showing that the nodes collected in the attributes of a tree state at π are from the subtree located at π.

Lemma A.4: If $x \in p_\pi$, $\pi_1 \in x.l_1$ then $\pi_1 = \pi\pi'$.

The proof is by induction on the height of $f[\pi]$.

If $f[\pi] = a\langle\varepsilon\rangle$ then $p_\pi = Up^{\leftharpoonup}(Down^{\leftharpoonup}(q_\pi, a), a)$. By the definition of $Down^{\leftharpoonup}$, $Up^{\leftharpoonup}$ and attributes it follows that $\pi_1 = \pi$.

Otherwise, by the definition of attributes we have that $\pi_1 = \pi$ or there is $y \in q_{\pi 0}$, $y = y_{0,j}$, $x \to a\langle r_j\rangle$ and $\pi_1 \in y.l_1$. From $\pi_1 \in y.l_1$ it follows by straightforward induction on $n = last_f(\pi)$ that there is $x_1 \in p_{\pi i}$ and $\pi_1 \in x_1.l_1$. By the induction hypothesis it follows that $\pi_1 = \pi i \pi'$. □

A.2.1 Proof of (i_1)

Let $\pi' = \pi i$ and $n = last_f(\pi')$.

Left-to-right: From $\pi_1 \in x.l_1$ it follows by Lemma A.4 that $\pi_1 = \pi'\pi_1'$. In the following we do the proof by induction on the length of π_1'.

If $\pi_1' = \lambda$ then $\pi_1 = \pi'$ and by the definition of attributes it follows that $x = x^1$. Our conclusion follows now by Theorem 5.1.

If $\pi_1' = l\pi_1''$ then $l \leq n$. By Theorem 5.1 there is f_a s.t. $(f, f_a) \in \mathcal{D}eriv_{r_0}$ and $lab(f_a[\pi']) = x$. From $\pi_1 \in x.l_1$ and $\pi' \neq \pi_1$ it follows by the definition of attributes that there is $x \to a\langle r_h\rangle$, $y_{0,h} \in q_{\pi' 0}$ and $\pi_1 \in y_0.l_1$. By the definition of attributes it follows by straightforward induction on n that there is m, $0 < m \leq n$ and $x_1, \ldots, x_m, y_1, \ldots, y_m$ s.t. $(y_{k-1}, x_k, y_k) \in \delta$, $y_k \in q_{\pi' k} \cap \vec{q}_{\pi' k}$, $x_k \in p_{\pi' k}$ for $k = 1, \ldots, m$ and $\pi_1 \in x_m.l_1$. By Lemma A.4 $m = l$. By the induction hypothesis it follows that there is f_c s.t. $(f, f_c) \in \mathcal{D}eriv_{r_0}$, $lab(f_c[\pi' l]) = x_l$ and $lab(f_c[\pi_1]) = x^1$.

From $y_l \in q_{\pi' l} \cap \vec{q}_{\pi' l}$ it follows from the definition of $Side^{\leftharpoonup}$ by straightforward induction on n that there are $x_l, \ldots, x_n, y_l, \ldots, y_n$ s.t. $(y_{k-1}, x_k, y_k) \in \delta$, $y_k \in q_{\pi' k} \cap \vec{q}_{\pi' k}$, $x_k \in p_{\pi' k}$ for $k = m+1, \ldots, n$. Also by the definitions of $Down^{\leftharpoonup}$ and $Up^{\leftharpoonup}$ $y_0 = y_{0,h}$ and $y_n \in F_p$. As NFA transitions are done only inside one NFA we have that $p = h$ and it follows that $x_1, \ldots, x_n \in [\![r_h]\!]$.

By Theorem 5.1 there is f_k s.t. $(f, f_k) \in \mathcal{D}eriv_{r_0}$, $lab(f_k[\pi' k]) = x_k$ for all k, and by Lemma A.1, $(f[\pi' k], f_k[\pi' k]) \in \mathcal{D}eriv_{x_k}$. Thus $(f[\pi' 1] \ldots f[\pi' n], f_1[\pi' 1] \ldots f_n[\pi' n]) \in \mathcal{D}eriv_{r_h}$ and with $x_1 \ldots x_n \in [\![r_h]\!]$, $(f[\pi'], x\langle f_1[\pi' 1] \ldots f_n[\pi' n]\rangle \in \mathcal{D}eriv_x$. Let $t = x\langle f_1[\pi' 1] \ldots f_n[\pi' n]\rangle$ and let $f_b = f_a/^{\pi'} t$. By Lemma A.2, $(f, f_b) \in \mathcal{D}eriv_{r_0}$, $lab(f_b[\pi']) = x$, $lab(f_b[\pi' l]) = x_l$.

Let $f_d = f_b/^{\pi' l} f_c[\pi' l]$. By Theorem 5.3 we now have that $(f, f_d) \in \mathcal{D}eriv_{r_0}$, $lab(f_d[\pi']) = x$ and $lab(f_d[\pi_1]) = x^1$.

Right-to-left: The proof is by induction on the length of π_1'.

If $\pi_1' = \lambda$ it follows that $x = x^1$ and by the definition of attributes $\pi_1 \in x.l_1$.

If $\pi_1' = l\pi_1''$ then $l \leq n$ and let $x_k = lab(f_1[\pi' k])$ for $k = 1, \ldots, n$. By Corollary 5.2, $x_k \in p_{\pi' k}$. By Lemma A.1 $(f[\pi'], f_1[\pi']) \in \mathcal{D}eriv_x$ and by the definition of $\mathcal{D}eriv_x$ we have that there is $x \to lab(f[\pi'])\langle r_h\rangle$ and $x_1 \ldots x_n \in [\![r_h]\!]$. Thus there are $y_0, \ldots, y_n$ s.t. $(y_{k-1}, x_k, y_k) \in \delta_h$ for $k = 1, \ldots, n$, $y_0 = y_{0,h}$ and $y_n \in F_h$. Also, by hypothesis there are $y \in q_{\pi'} \cap \vec{q}_{\pi'}$ and y' s.t. $(y', x, y) \in \delta$. Using this, one can show by using the definition of $Down$, $Side$, and $Down^{\leftharpoonup}$, $Side^{\leftharpoonup}$ that for $k = 0, \ldots, n$, $y_k \in \vec{q}_{\pi' k}$ and $y_k \in q_{\pi' k}$, respectively.

By the induction hypothesis $\pi_1 \in x_l.l_1$. By straightforward induction on l, using the definition of $Side^{\leftharpoonup}$ and of the attributes, it follows that $\pi_1 \in y_0.l_1$. Now by the definition of $Up^{\leftharpoonup}$ and of the attributes it follows that $\pi_1 \in x.l_1$.

A.2.2 Proof of (i_2)

Let $n = last_f(\pi)$.

Left-to-right: Let $y_i = y$.

From $\pi_2 \in y.l_2$ it follows from the definition of *Side*$^{\leftharpoonup}$ and of attributes by straightforward induction on n that there are j, $i < j \leq n$, $y_{i+1}, \ldots, y_j$, $x_{i+1}, \ldots, x_j$, s.t. $(y_{k-1}, x_k, y_k) \in \delta_p$ for $k = i+1, \ldots, j$ with $y_k \in q_{\pi k} \cap \vec{q}_{\pi k}$ for all k and $\pi_2 \in x_j.l_2$. By (i_1) it follows that $\pi_2 = \pi j \pi_2'$ and there is f_a s.t. $(f, f_a) \in \mathcal{D}eriv_{r_0}$, $lab(f_a[\pi j]) = x_j$ and $lab(f_a[\pi_2]) = x^2$.

From $y_i \in q_{\pi i} \cap \vec{q}_{\pi i}$ it follows from the definitions of *Side* and *Side*$^{\leftharpoonup}$ that there are $y_0, \ldots, y_{i-1}, x_1, \ldots, x_i$ s.t. $y_k \in q_{\pi k} \cap \vec{q}_{\pi k}$ for $k = 0, \ldots, i-1$, $(y_{k-1}, x_k, y_k) \in \delta_h$ for $k = 1, \ldots, i$ and $y_0 = y_{0,h}$ for some h. By the Berry-Sethi construction, since $(y', x, y_i) \in \delta$ and $(y_{i-1}, x_i, y_i) \in \delta$, it follows that $x = x_i$. Similarly, from $y_j \in q_{\pi j} \cap \vec{q}_{\pi j}$ it follows that there are $y_j, \ldots, y_n$ s.t. $y_k \in q_{\pi k} \cap \vec{q}_{\pi k}$ for $k = j, \ldots, n$, $(y_{k-1}, x_k, y_k) \in \delta_g$ for $k = j+1, \ldots n$ and $y_n \in F_g$ for some g. Because transitions in δ can be made only inside the same NFA we have that $p = g = h$. We further get that $x_1 \ldots x_n \in [\![r_h]\!]$.

By Theorem 5.1 it follows that there is f_k s.t. $(f, f_k) \in \mathcal{D}eriv_{r_0}$, $lab(f_k[\pi k]) = x_k$ and by Lemma A.1 $(f[\pi k], f_k[\pi k]) \in \mathcal{D}eriv_{x_k}$ for $k = 1, \ldots, n$. Let the forest $f_b = f_1[\pi 1] \ldots f_n[\pi n]$. It follows that $(f[\pi 1] \ldots f[\pi n], f_b) \in \mathcal{D}eriv_{r_h}$. Let $f_c = f_b /^j f_a[\pi j]$. By Lemma A.1 $(f[\pi j], f_a[\pi j]) \in \mathcal{D}eriv_{x_j}$ and by Lemma A.3 we have that $(f[\pi 1] \ldots f[\pi n], f_c) \in \mathcal{D}eriv_{r_h}$, $lab(f_c[i]) = lab(f_b[i]) = x_i = x$ and $lab(f_c[j\pi_2']) = lab(f_a[\pi_2]) = x^2$.

Now, if $\pi = \lambda$ then $h = 0$ and $f = f[\pi 1] \ldots f[\pi n]$. As above $(f, f_c) \in \mathcal{D}eriv_{r_0}$ with the required properties.

If $\pi \neq \lambda$ then by the definition of *Down*$^{\leftharpoonup}$ there are $y'' \in q_\pi \cap \vec{q}_\pi$, $(y''', x', y'') \in \delta$, $x' \rightarrow a\langle r_h \rangle$. By Theorem 5.1 there is f_d s.t. $(f, f_d) \in \mathcal{D}eriv_{r_0}$ and $lab(f_d[\pi]) = x'$. Let $t = x'\langle f_c \rangle$. We have that $(f[\pi], t) \in \mathcal{D}eriv_{x'}$. Let $f_e = f_d /^\pi t$. By Lemma A.2 we have that $(f, f_e) \in \mathcal{D}eriv_{r_0}$ with the required properties.

Right-to-left: Let $x_k = lab(f_2[\pi k])$ for $k = 1, \ldots, n$. By (i_1) $\pi_2 \in x_j.l_2$.

We first show that $x_1 \ldots x_n \in [\![r_h]\!]$ for some h. If $\pi = \lambda$ then by the definition of $\mathcal{D}eriv_{r_0}$ it follows that $x_1 \ldots x_n \in [\![r_0]\!]$. If $\pi \neq \lambda$ let $lab(f_2[\pi]) = x'$. It follows by Theorem 5.1 that there is $(y''', x', y'') \in \delta$ and $y'' \in q_\pi \cap \vec{q}_\pi$. By Lemma A.1 $(f[\pi], f_2[\pi]) \in \mathcal{D}eriv_{x'}$. By the definitions $\mathcal{D}eriv_{x'}$ there is $x' \rightarrow a\langle r_h \rangle$ and $x_1 \ldots x_n \in [\![r_h]\!]$.

There are thus $y_0, \ldots, y_n$ s.t. $y_0 = y_{0,h}$, $y_n \in F_h$ and $(y_{k-1}, x_k, y_k) \in \delta_h$ for all k. From the definitions of transitions it follows that $y_k \in q_{\pi k} \cap \vec{q}_{\pi k}$.

By Corollary 5.2, $x_k \in p_{\pi k}$. From $\pi_2 \in x_j.l_2$ it follows by the definitions of attributes by straightforward induction on j that $\pi_2 \in y_i.l_2$. With $y = y_i$ we get the desired result.

A.3 Proof of Theorem 6.2

In the following we use the notations of Section 6.1.6, where Theorem 6.2 is stated.

Directly from Theorem 6.1 (on page 75) it follows by the definition of $\mathcal{M}_{Q,f}$ (on page 72) the following corollary:

Corollary A.1: $(f, f') \in \mathcal{D}eriv_e$ for some $e \in E_0$, $\pi' \in \Pi^{f'}_{\pi}$ with $lab(f'[\pi']) = x$ iff $y \in q_\pi \cap \vec{q}_\pi$ and $(y_1, x, y) \in \delta$ for some $y, y_1 \in Y$.

The construction presented in Section 6.1.6 ensures the following invariant:

Lemma A.5: $y_{0,i} \in q_{\pi 0}$ and $\pi_1 \in y_{0,i}.l_1$ (or $\pi_1 \in y_{0,i}.l_2$, respectively) iff $\pi_1 = \pi l \pi'_1$, $\exists (f, f') \in \mathcal{D}eriv_e$ for some $e \in E_0$ and either

1. $\pi = \lambda$, $pos(e) = \{r'_1, \ldots, r'_p\}$, $r_i = r'_k$ for some $1 \leq k \leq p$, and $f' = \&\langle f_1 \rangle \ldots \&\langle f_p \rangle$ or
2. $\pi \neq \lambda$, $x \to a\langle e_1 \rangle \in R$, $pos(e_1) = \{r'_1, \ldots, r'_p\}$, $r_i = r'_k$ for some $1 \leq k \leq p$, $\exists \pi' \in \Pi^{f'}_{\pi}$, $f'[\pi'] = x\langle \&\langle f_1 \rangle \ldots \&\langle f_p \rangle \rangle$

and $(f[\pi 1] \ldots f[\pi n], f_j) \in \mathcal{D}eriv^1_{r'_j}$ for all $j = 1 \ldots p$, $\exists k \pi_2 \in \Pi^{\&\langle f_1 \rangle \ldots \&\langle f_p \rangle}_{l \pi'_1}$ and $lab(f_k[\pi_2]) = x^1$ (or $lab(f_k[\pi_2]) = x^2$, respectively), where $n = last_f(\pi)$.

We proof the two directions of Lemma A.5 separately.

Left-to-right Let $y_{0,i} \in q_{\pi 0}$ and $\pi_1 \in y_{0,i}.l_1$. Since the primaries are propagated bottom-up by construction, i.e. the path to an already found primary match is longer than the path to the current node $\pi 0$, we have that $\pi_1 = \pi l \pi'_1$ for some l and π'_1.

The proof will be by induction on the length of π'_1.

From $y_{0,i} \in q_{\pi 0}$ it follows by the definition of the $Side^{\leftharpoonup}$ transition that the content model r_i is fulfilled by the nodes on the current level. That is, $\exists x_1 \ldots x_n \in [\![r_i]\!]$, $y_j \in q_{\pi j} \cap \vec{q}_{\pi j}$ and $(y_{j-1}, x_j, y_j) \in \delta_i$ for $j = 1, \ldots, n$. Furthermore, by the definition of the attribute l_1 at $Side^{\leftharpoonup}$ transitions, π_1 must have been propagated from some x_j, with $1 \leq j \leq n$. Since π_1 is found in the l-th subtree (because $\pi_1 = \pi l \pi'_1$), it must be that $\pi_1 \in x_l.l_1$. From $y_l \in q_{\pi l} \cap \vec{q}_{\pi l}$ and $(y_{l-1}, x_l, y_l) \in \delta_i$ it follows by Corollary A.1 that $\exists (f, f') \in \mathcal{D}eriv_e$ for some $e \in E_0$ and $\exists \pi_3 \in \Pi^{f'}_{\pi l}$ with $lab(f'[\pi_3]) = x_l$. That is, by distinguishing on whether the current level is a top level or there is a father node in the input, either:

1. $\pi = \lambda$, $pos(e) = \{r'_1, \ldots, r'_p\}$, $r_i = r'_k$ for some $1 \leq k \leq p$, and $f' = \&\langle f_1 \rangle \ldots \&\langle f_p \rangle$ or
2. $\pi \neq \lambda$, $x \to a\langle e_1 \rangle \in R$, $pos(e_1) = \{r'_1, \ldots, r'_p\}$, $r_i = r'_k$ for some $1 \leq k \leq p$, $\exists \pi' \in \Pi^{f'}_{\pi}$, $f'[\pi'] = x\langle \&\langle f_1 \rangle \ldots \&\langle f_p \rangle \rangle$

and $(f[\pi 1] \ldots f[\pi n], f_j) \in \mathcal{D}eriv^1_{r'_j}$ for all $j = 1, \ldots, p$ and $lab(f_k[l]) = x_l$.

Base case: If $\pi'_1 = \lambda$ it directly follows from $\pi_1 \in x_l.l_1$ that $x_l = x^1$ (otherwise $\pi'_1 \neq \lambda$) and f' is the derivation as required by our lemma.

Induction step: If $\pi'_1 \neq \lambda$ let $\pi'_1 = m \pi''_1$. It follows from $\pi_1 \in x_l.l_1$ by the definition of the attribute l_1 at the $Up^{\leftharpoonup}$ transition to πl that $\exists y_{0,r} \in q_{\pi l 0}$ and $\pi_1 \in y_{0,r}.l_1$ with $x_l \to a\langle e'_1 \rangle \in R$, $pos(e'_1) = \{r''_1, \ldots, r''_{p'}\}$, $r_r = r''_{k'}$ for some $1 \leq k' \leq p'$. From $y_{0,r} \in q_{\pi l 0}$ and $\pi_1 \in y_{0,r}.l_1$, it follows by the induction hypothesis that $\exists (f, f'') \in \mathcal{D}eriv_{e'}$ for some $e' \in E_0$, $\exists \pi'' \in \Pi^{f''}_{\pi l}$, $f''[\pi''] = x_l \langle \&\langle f'_1 \rangle \ldots \&\langle f'_{p'} \rangle \rangle$

with $(f[\pi l1] \dots f[\pi l\ last_{\pi l}(f)], f'_j) \in \mathcal{D}eriv^1_{r''_j}$ for all $j = 1 \dots p'$, and $\exists k'\pi'_2 \in \Pi^{\&\langle f'_1\rangle \dots \&\langle f'_{p'}\rangle}_{m\pi''_1}$ and $lab(f_{k'}[\pi'_2]) = x^1$.

The derivation required by our lemma is obtained by replacing the content of x_l from f' with the content of x_l from f'' (similarly to the grafting used in the proof for simple grammar queries).

Right-to-left The proof is by induction on the length of π'_1.

Base case: Let $\pi'_1 = \lambda$. Then, since $k\pi_2 \in \Pi^{\&\langle f_1\rangle \dots \&\langle f_p\rangle}_{l}$, it follows that $\pi_2 = l$. Now, either $\pi = \lambda$ and $f'[kl] = x^1$, or $\pi \neq \lambda$ and $f'[\pi' kl] = x^1$. It follows by Corollary A.1 that $\exists y \in q_{\pi l} \cap \vec{q}_{\pi l}$ and $(y_1, x^1, y) \in \delta_i$. It follows by the definition of the $Side^{\leftharpoonup}$ transitions that $y_{0,i} \in q_{\pi 0}$ and $\pi_1 \in y_{0,i}.l_1$.

Induction step: Consider now the case in which $\pi'_1 = l'\pi''_1$. From $k\pi_2 \in \Pi^{\&\langle f_1\rangle \dots \&\langle f_p\rangle}_{l\pi'_1}$ it follows that $\pi_2 = lk'l'\pi'_2$. Let $f_k[l] = x'\langle \&\langle f'_1\rangle \dots \&\langle f'_{p'}\rangle\rangle$. From the definition of $\mathcal{D}eriv$ we have that $x' \to a\langle e'_1\rangle \in R$, $pos(e'_1) = \{r''_1, \dots, r''_{p'}\}$, $(f[\pi l1] \dots f[\pi l\ last_f(\pi l)], f'_j) \in \mathcal{D}eriv^1_{r_j}$ for all $j = 1 \dots p'$, $k'l'\pi'_2 \in \Pi^{\&\langle f'_1\rangle \dots \&\langle f'_{p'}\rangle}_{l'\pi''_1}$ and $lab(f_{k'}[l'\pi'_2]) = x^1$. It follows by the induction hypothesis that $y_{0,m} \in q_{\pi l0}$ and $\pi_1 \in y_{0,m}.l_1$ with $r_m = r''_k$ for some $1 \leq k \leq p'$. By the computation rules of the attributes l_1 at $Up^{\leftharpoonup}$ transitions it follows that $x' \in p_{\pi l}$. Since $lab(f'[\pi kl]) = x'$ it follows by Corollary A.1 and the computation of l_1 at $Side^{\leftharpoonup}$ transitions that $\pi_1 \in y_{0,i}.l_1$ for some i.

□

A.3.1 Right-to-left Direction of Theorem 6.2

For the right-to-left direction of the proof, the cases (a) to (e) result similarly as in the proof of Theorem 5.4. Case (f) follows from Lemma A.5 as explained below.

Suppose that $y_{0,i}, y_{0,s} \in q_{\pi 0}$, $i \neq s$, $R_k \equiv x \to a\langle e'\rangle$, $r_i, r_s \in pos(e')$, and furthermore $\pi_1 \in y_{0,i}.l_1$, $\pi_2 \in y_{0,s}.l_2$.

From $y_{0,i} \in q_{\pi 0}$ and $\pi_1 \in y_{0,i}.l_1$, it follows by Lemma A.5 that $\pi_1 = \pi l\pi'_1$, $\exists (f, f') \in \mathcal{D}eriv_e$ for some $e \in E_0$ and either

1. $\pi = \lambda$, $pos(e) = \{r'_1, \dots, r'_p\}$, $r_i = r'_k$ for some $1 \leq k \leq p$, and $f' = \&\langle f_1\rangle \dots \&\langle f_p\rangle$, and since $r_i \in pos(e')$, $r_i \in pos(e)$, $e = e'$, or

2. $\pi \neq \lambda$, $x \to a\langle e_1\rangle \in R$, $pos(e_1) = \{r'_1, \dots, r'_p\}$, $r_i = r'_k$ for some $1 \leq k \leq p$, $\exists \pi' \in \Pi^{f'}_{\pi}$, $f'[\pi'] = x\langle \&\langle f_1\rangle \dots \&\langle f_p\rangle\rangle$ and since $r_i \in pos(e')$, $r_i \in pos(e_1)$, $e_1 = e'$

and $(f[\pi 1] \dots f[\pi n], f_j) \in \mathcal{D}eriv^1_{r'_j}$ for all $j = 1 \dots p$, $\exists k\pi_2 \in \Pi^{\&\langle f_1\rangle \dots \&\langle f_p\rangle}_{l\pi'_1}$ and $lab(f_k[\pi_2]) = x^1$, where $n = last_f(\pi)$.

Similarly, from $y_{0,s} \in q_{\pi 0}$ and $\pi_2 \in y_{0,s}.l_2$ it follows by Lemma A.5 that $\pi_2 = \pi m\pi'_2$, $\exists (f, f'') \in \mathcal{D}eriv_e$ for some $e \in E_0$ and either

1. $\pi = \lambda$, $pos(e) = \{r'_1, \dots, r'_p\}$, $r_s = r'_k$ for some $1 \leq k \leq p$, and $f'' = \&\langle f'_1\rangle \dots \&\langle f'_p\rangle$ or

2. $\pi \neq \lambda$, $x \rightarrow a\langle e_1 \rangle \in R$, $pos(e_1) = \{r'_1, \ldots, r'_p\}$, $r_s = r'_k$ for some $1 \leq k \leq p$, $\exists \pi' \in \Pi^{f'}_{\pi}$, $f'[\pi'] = x\langle \&\langle f'_1 \rangle \ldots \&\langle f'_p \rangle \rangle$

and $(f[\pi 1] \ldots f[\pi n], f'_j) \in \mathcal{D}eriv^1_{r_j}$ for all $j = 1 \ldots p$, $\exists k \pi''_2 \in \Pi^{\&\langle f_1 \rangle \ldots \&\langle f_p \rangle}_{m\pi'_2}$ and $lab(f_k[\pi''_2]) = x^2$.

Note that e and e_1 are the same in the two application of Lemma A.5 as r_i and r_s are positive content models of the same boolean content model. Let f''' be the forest obtained by replacing f_k in f' with f'_k (i.e. by grafting f'_k into f'). It is straightforward now that f''' is as required by the definition of (π_1, π_2) as binary match.

A.3.2 Left-to-right Direction of Theorem 6.2

Let (π_1, π_2) be a binary match. There is by the definition $(f, f') \in \mathcal{D}eriv_e$ for some $e \in E_0$ and $\exists \pi'_1 \in \Pi^{f'}_{\pi_1}, \pi'_2 \in \Pi^{f'}_{\pi_2}$ with $lab(f'[\pi'_1]) = x^1$ and $lab(f'[\pi'_2]) = x^2$.

The proof considers the different possible relative positions of π'_1 and π'_2 in f'. The nodes π'_1 and π'_2 have as nearest common ancestor in f' either a node labeled with a non-terminal x or with the symbol "&". If the nearest common ancestor is a node $\&\langle x_1\langle f_1 \rangle \ldots x_n\langle f_n \rangle \rangle$, the cases (a) to (e) result similarly as in the proof of Theorem 5.4.

Otherwise, let the π' be the nearest common ancestor of π'_1 and π'_2 in f'. It follows that $\pi'_1 = \pi' i \pi''_1$, $\pi'_2 = \pi' j \pi''_2$ with $i \neq j$, and $\pi' \in \Pi^{f}_{\pi}$ where π is the nearest common ancestor of π_1 and π_2 in f, i.e. $\pi_1 = \pi\pi_3$, $\pi_2 = \pi\pi_4$ for some π_3, π_4.

Furthermore, let $f'[\pi'] = x\langle \&\langle f_1 \rangle \ldots \&\langle f_p \rangle \rangle$. Thus, $lab(f_i[\pi''_1]) = x^1$ and $lab(f_j[\pi''_2]) = x^2$ and also $i\pi''_1 \in \Pi^{\&\langle f_1 \rangle \ldots \&\langle f_p \rangle}_{\pi_3}$, $j\pi''_2 \in \Pi^{\&\langle f_1 \rangle \ldots \&\langle f_p \rangle}_{\pi_4}$.

Since $(f, f') \in \mathcal{D}eriv_e$, $x \rightarrow a\langle e_1 \rangle \in R$, $pos(e_1) = \{r'_1, \ldots, r'_p\}$ and $(f[\pi 1] \ldots f[\pi n], f_j) \in \mathcal{D}eriv^1_{r'_j}$ for all $j = 1, \ldots, p$.

By Lemma A.5 it follows that $y_{0,i} \in q_{\pi 0}$ and $\pi_1 \in y_{0,i}.l_1$ with $r_i = r'_i$. Also by Lemma A.5, $y_{0,s} \in q_{\pi 0}$ and $\pi_2 \in y_{0,s}.l_2$ with $r_s = r'_j$. We thus meet the requirements of our case (f).

A.4 Proof of Theorem 7.3

In the following we use the notation from Appendix 7 where Theorem 7.3 was stated.

Alternative Definition of Matches

In order to proof Theorem 7.3 a more refined definition of matches is needed in which the NFA states reached while checking the content models of elements are given explicitely. Let R be a set of forest grammar productions, r_0 be a regular expression over non-terminals and f an input forest. A *(non-deterministic, accepting) run* f_R over f for R and r_0, denoted $f_R \in \mathcal{R}uns_{r_0,f}$ is defined as follows:

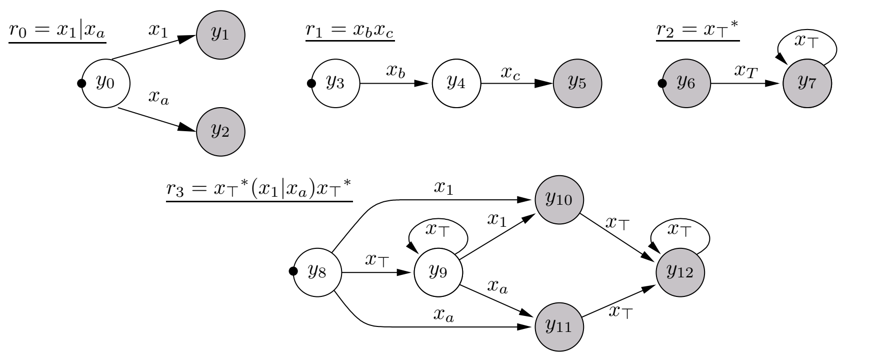

$r_0 = x_1|x_a$
$r_1 = x_b x_c$
$r_2 = x_\top{}^*$
$r_3 = x_\top{}^*(x_1|x_a)x_\top{}^*$
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12}
x_1 x_a x_b x_c $x_\top$

Figure A.1: NFAs obtained by Berry-Sethi construction for regular expressions in Example A.1

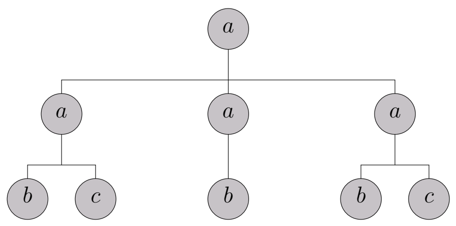


Figure A.2: Input tree t

$$y_0\langle f_1'\rangle \ \ldots \ y_{n-1}\langle f_n'\rangle \ y_n\langle\rangle \in \mathcal{R}uns_{r_0, a_1\langle f_1\rangle \ \ldots \ a_n\langle f_n\rangle} \text{ iff}$$
$$y_0 = y_{0,0}, y_n \in F_0, \text{ and}$$
$$(y_{i-1}, x_i, y_i) \in \delta_0, x_i \to a_i\langle r_i\rangle, f_i' \in \mathcal{R}uns_{r_i, f_i} \text{ for all } i = 1, \ldots, n$$

$$y \in \mathcal{R}uns_{r_0,\varepsilon} \text{ iff } y = y_{0,0}, y \in F_0$$

An example run is given immediately below.

It is straightforward to see that a derivation f' with $(f, f') \in \mathcal{D}eriv_{r_0}$ (defined on page 30) exists iff a run $f_R \in \mathcal{R}uns_{r_0,f}$ exists.

Example A.1: Let $G = (R, r_0)$ with R being the set of rules from Example 4.3 reproduced for convenience below:

(1) $x_\top \to a\langle x_\top^*\rangle$	(4) $x_1 \to a\langle x_\top^*(x_1|x_a)x_\top^*\rangle$	(6) $x_b \to b\langle x_\top^*\rangle$
(2) $x_\top \to b\langle x_\top^*\rangle$	(5) $x_a \to a\langle x_b x_c\rangle$	(7) $x_c \to c\langle x_\top^*\rangle$
(3) $x_\top \to c\langle x_\top^*\rangle$		

The NFAs for the regular expressions occurring in grammar G with the set are reproduced in Figure A.1.

Consider the input tree depicted t reproduced for convenience in Figure A.2 and one derivation of t' w.r.t. r_0 depicted in Figure A.3. A run corresponding to t' is depicted in Figure A.4 via dotted lines.

□

The derivation corresponding to a run can be obtained by taking the incoming transitions of the NFA states of the nodes which are not the first in their siblings sequence as one can see in Figure A.4. Formally, the following expresses the relation between *derivations* and *runs*:

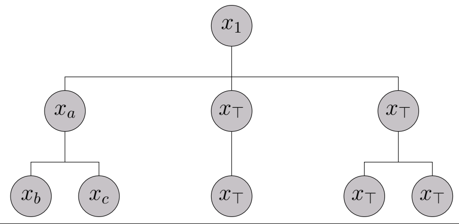


Figure A.3: Derivation t' of t w.r.t r_0

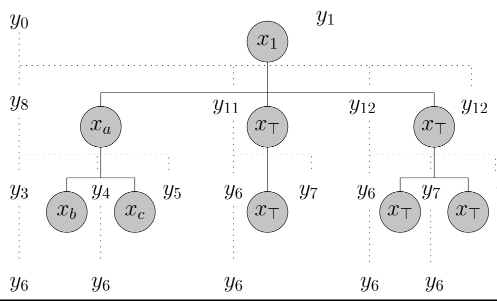


Figure A.4: Run corresponding to t'

$$\begin{aligned}&(f, f') \in \mathcal{D}eriv_r \\ &\quad \text{iff} \;\; \exists f_R \in \mathcal{R}uns_{r,f} \text{ with } L(f') = N(f_R) \\ &\qquad \text{and } lab(f'[\pi p]) = in(lab(f_R[\pi(p+1)])) \text{ for all } \pi p \in N(f').\end{aligned}$$

Let f be an input forest and $Q = ((R, r_0), T)$ a grammar query. Matches of Q, which were originally defined in terms of *derivations*, can be equivalently defined in terms of runs as it follows:

$$\pi p \in \mathcal{M}_{Q,f} \text{ iff } \exists f_R \in \mathcal{R}uns_{r_0,f} \text{ s.t. } in(lab(f_R[\pi(p+1)])) \in T.$$

Notations

Before proceeding with the proof we further introduce a couple of useful notations. The set of *matches defined by runs with label y at location l* is defined as:

$$\begin{aligned}\pi p \in \mathcal{M}^{l,y}_{Q,f} \text{ iff } \;\; &\exists f_R \in \mathcal{R}uns_{r_0,f} \text{ s.t. } in(lab(f_R[\pi(p+1)])) \in T \\ &\text{and } lab(f_R[l]) = y\end{aligned}$$

The set of *l-right-ignoring matches defined by a run with label y at l* is defined as:

$$\pi \in ri\text{-}\mathcal{M}^{l,y}_{Q,f} \text{ iff } \pi \in \mathcal{M}^{l,y}_{Q,f_2} \forall f_2 \in \mathcal{R}ightCompl_{f,l}$$

A node π' is a *πi-upper-right ignoring match defined by a run with label y at πi* iff for any right-completion f_2 at the parent of πi there is a run defining π' as a match of Q in f_2 which labels πi with y, formally:

$$\pi' \in uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f} \text{ iff } \pi' \in \mathcal{M}^{\pi i,y}_{Q,f_2} \forall f_2 \in \mathcal{R}ightCompl_{f,\pi}$$

Given a location πi and an NFA state y, a sequence of states is a *suffix run from y at πi* iff the last state in the sequence is a final state and the sequence of siblings to the right of πi allows to visit the sequence of states, formally:

$y_i, \ldots, y_n \in \mathcal{S}uf_{\pi i,y}$
iff $(y_{k-1}, x_k, y_k) \in \delta_j, f_1[\pi k] \in [\![R]\!]\ x_k$ with $x_k = in(y_k), \forall k \in i, \ldots, n$ and $y_{i-1} = y, y_n \in F_j$ where $n = last_{f_1}(\pi)$

To denote the information on top of the stack at the some moment πi we write $\pi i.q$, $\pi i.m$ and $\pi i.ri$ in analogy to attributes of attribute grammars. Similarly to attribute grammars, these are computed by local rules as presented in Section 7.2.1.

Proof

Theorem 7.3 is a straightforward corollary of the following theorem:

Theorem A.1: The construction presented in Section 7.2.1 keeps the following invariant:

$$\begin{array}{l} \pi' p \in \pi i.m(y), y \in \pi i.q \cap Y_j, \exists c \in \mathcal{S}uf_{\pi i,y} \text{ and } r_j \in \pi i.ri \\ \text{iff } \pi' p < \pi i \text{ and } \pi' p \in uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f} \end{array} \tag{A.1}$$

We proof the two directions of Theorem A.1 separately.

Left-to-right

We show that (A.1) holds at all locations in the input by induction using the lexicographic order on locations.

Base case Initially, at location 1, $1.m(y) = \emptyset, \forall y \in 1.q$, thus $\pi' p \in 1.m(y)$ is false, and the left-to-right direction trivially holds.

Induction step Supposing that (A.1) holds at all locations up to some location l we show that it also holds at the immediately next location.

Start-tag transition We first show that if (A.1) holds at $\pi i \in N(f)$, so does it at $\pi i1$.

Let $\pi' p \in \pi i1.m(y_0)$, $y_0 \in Y_j$ and suppose $\exists c \in \mathcal{S}uf_{\pi i1,y_0}$ and $r_j \in \pi i1.ri$. Since $\pi' p \in \pi i1.m(y_0)$, it follows by our construction (conform to (7.1) on page 93) that $\exists y \in \pi i.q$ with $(y, x, y') \in \delta_k$, $x \to a\langle r_j \rangle$, $rightIgn(y')$, $r_k \in \pi i.ri$ and either (i) $\pi' p \in \pi i.m(y)$ or (ii) $\pi' p = \pi i$ and $x \in T$.

In case (i) it follows from (A.1) at πi that $\pi' p < \pi i$ and $\pi' p \in uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f}$. Thus, obviously $\pi' p < \pi i < \pi i1$ and it remains to show that $\pi' p \in uri\text{-}\mathcal{M}^{\pi i1,y_0}_{Q,f}$. This follows directly from $\pi' p \in uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f}$, $c \in \mathcal{S}uf_{\pi i1,y_0}$ and $rightIgn(y')$ by grafting the run over the children of πi corresponding to c into the run corresponding to $uri\text{-}\mathcal{M}^{\pi i,y}_{Q,f}$.

In case (ii), $\pi' p = \pi i < \pi i1$. The proof will use in this case the following lemma (also used later on):

Lemma A.6: If there is a suffix run within a right ignoring content model, then, independently of what follows in the input after the enclosing element, there is a run over the input forest containing that suffix. Formally, if $y \in \pi i.q \cap Y_k$,

$r_k \in \pi i.ri$ and $\exists c \in \mathcal{S}uf_{\pi i,y}$ then $\forall f_2 \in \mathcal{R}ightCompl_{f,\pi}\ \exists f_R \in \mathcal{R}uns_{r_0,f_2}$ with $c = lab(f_R[\pi i]), \ldots, lab(f_R[\pi\ last_{f_R}(\pi)])$.

The proof is by straightforward induction on the locations in the input forest. The assertion trivially holds at location 1. For the induction step, let $\pi i \in N(f)$. We show that if the assertion holds at the location πi, it also holds at (i) $\pi i1$ and (ii) at $\pi(i+1)$. In case (i) $\exists y \in \pi i.q$ with $(y, x, y') \in \delta_k$, $x \to a\langle r_j\rangle$, $rightIgn(y')$, $r_k \in \pi i.ri$. The required run is obtained by grafting the run over the children of πi corresponding to c into the run $y, y', \ldots$ corresponding to the induction hypothesis. In case (ii) the existence of the suffix run at $\pi(i+1)$ implies the existence of a run at πi and our conclusion follows by the induction hypothesis. □

We continue now with the proof of Theorem A.1.

Since $c \in \mathcal{S}uf_{\pi i1,y_0}$ it follows (straightforwardly by definition) that $f[\pi i] \in [\![R]\!]\ x$. Given that $(y, x, y') \in \delta_k$ and $rightIgn(y')$ it follows that there is thus a suffix run $c \in \mathcal{S}uf_{\pi i,y}$ with $c = y, y', \ldots$. With $r_k \in \pi i.ri$ it follows by Lemma A.6 that $\forall f_2 \in \mathcal{R}ightCompl_{f,\pi}\ \exists f_R \in \mathcal{R}uns_{r_0,f_2}$ with $lab(f_R[\pi i]) = y'$. Since $rightIgn(y')$ it follows that $\exists f_R \in \mathcal{R}uns_{r_0,f_2}$ for any $f_2 \in \mathcal{R}ightCompl_{f,\pi i}$ and $lab(f_R[\pi i]) = y'$. With $c \in \mathcal{S}uf_{\pi i1,y_0}$ it follows that $\exists f'_R \in \mathcal{R}uns_{r_0,f_2}$ (obtained by grafting the run over the children of πi corresponding to c into f_R) with $lab(f'_R[\pi i]) = y'$ and $lab(f'_R[\pi i1]) = y_0$. Thus $\pi' p \in uri\text{-}\mathcal{M}^{\pi i1,y_0}_{Q,f}$.

End-tag transition We next show that if (A.1) holds at $l\ \forall l < \pi(i+1)$, so does it at $\pi(i+1)$.

Let $\pi' p \in \pi(i+1).m(y'')$, $y'' \in Y_k$ and suppose $\exists c \in \mathcal{S}uf_{\pi(i+1),y''}$ and $r_k \in \pi(i+1).ri$. Since $\pi' p \in \pi(i+1).m(y'')$, it follows by our construction (conform to (7.2) on page 94) that $\exists y \in \pi i.q$, $y' \in \pi i(n+1).q$ with $y' \in F_j$, $x \to a\langle r_j\rangle$, $(y, x, y'') \in \delta_k$ and either (i) $\pi' p \in \pi i.m(y)$, or (ii) $\pi' p \in \pi i(n+1).m(y')$, or (iii) $\pi' p = \pi i$ and $x \in T$.

In case (i) our conclusion follows directly from (A.1) at πi.

We continue with the cases (ii) and (iii). Given c and $r_k \in \pi(i+1).ri$ it follows by Lemma A.6 that $\forall f_2 \in \mathcal{R}ightCompl_{f,\pi}\ \exists f_R \in \mathcal{R}uns_{r_0,f_2}$ s.t. $lab(f_R[\pi(i+1)]) = y''$. Further we use the following lemma (also employed later on):

Lemma A.7: If $y \in \pi n.q \cap \mathcal{F}_j$ then $\exists f_R \in \mathcal{R}uns_{r_j,f[\pi 1]\ldots f[\pi n]}$ with $lab(f_R[\pi(n+1)]) = y$.

The proof is straightforward by induction on the depth of $f[\pi]$. □

In case (ii), $\pi' p$ was found either before or while visiting the content of πi, that is either $\pi' p \leq \pi i$ or $\pi i < \pi' p < \pi(i+1)$, respectively. In the first case our conclusion follows directly from (A.1) at πi. In the second case $\pi' p < \pi(i+1)$ we further need the following lemma:

Lemma A.8: If $y \in \pi n.q \cap \mathcal{F}_j$, $\pi' p \in \pi n.m(y)$ and $\pi 1 \leq \pi' p \leq \pi n$ then $\exists f_R \in \mathcal{R}uns_{r_j,f[\pi 1]\ldots f[\pi n]}$ with $lab(f_R[\pi(n+1)]) = y$ and $in(lab(f_R[\pi'(p+1)])) \in T$ where $n = last_f(\pi)$.

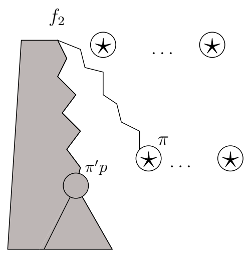


Figure A.5: Right completion of f at π

The proof is by induction on the depth of $f[\pi]$. By Lemma A.7 $\exists f'_R \in \mathcal{R}uns_{r_j, f[\pi 1]\ldots f[\pi n]}$ with $lab(f'_R[\pi(n+1)]) = y$.

For depth 1 it directly follows that $\pi' p = \pi i$ for some $1 \leq i \leq n$ and $in(lab(f'_R[\pi'(p+1)])) \in T$. Therefore $f_R = f'_R$ is the sought after run. If the depth is more than 1, then either (A) $\pi' p = \pi i$ for some $1 \leq i \leq n$ and $in(lab(f_R[\pi'(p+1)])) \in T$ as above or (B) $\exists y' \in \pi i n'.q \cap \mathcal{F}_k$, $\pi' p \in \pi i n'.m(y)$ and $\pi i 1 \leq \pi' p \leq \pi i n'$ for some $1 \leq i \leq n$ and $n' = last_f(\pi i)$. In case (B) f_R in our conclusion can be constructed by grafting the run over the children of πi existent by the induction hypothesis into f'_R. □

Our conclusion results now for the case (ii) $\pi i < \pi' p < \pi(i+1)$ by grafting the run corresponding to the children which defines the match (as implied by Lemma A.8) into f_R.

In case (iii) $\pi' p = \pi i < \pi(i+1)$ and it remains to show that $\pi' p \in uri\text{-}\mathcal{M}_{Q,f}^{\pi(i+1),y''}$. We have by Lemma A.7 that $\exists f'_R \in \mathcal{R}uns_{r_j, f[\pi i 1]\ldots f[\pi i n]}$ and $in(lab(f'_R[\pi'(p+1)])) \in T$. From f_R and f'_R it results (by grafting f'_R into f_R at π) that $\exists f''_R \in \mathcal{R}uns_{r_0, f_2}$ s.t. $in(lab(f''_R[\pi'(p+1)])) \in T$ and $lab(f''_R[\pi(i+1)]) = y''$, thus $\pi' p \in uri\text{-}\mathcal{M}_{Q,f}^{\pi(i+1),y''}$.

Right-to-left

Let $\pi' p < \pi i$ and $\pi' p \in uri\text{-}\mathcal{M}_{Q,f}^{\pi i,y}$. Let f_2 be a right-completion of f at π obtained by adding on every level from the root to π inclusively an arbitrary number of right siblings $\star\langle\rangle$, as depicted in Figure A.5, where $\star$ is a symbol not occurring in any of the rules in the grammar. Since $\pi' p \in uri\text{-}\mathcal{M}_{Q,f}^{\pi i,y}$ it follows that $\exists f_R \in \mathcal{R}uns_{G,f_2}$ s.t. $in(lab(f_R[\pi'(p+1)])) \in T$ and $lab(f_R[\pi i]) = y$.

Also, since $\star$ does not occur in any rule f_R must label all the ancestors of the πi node with right-ignoring states, i.e. $rightIgn(lab(f_R[\pi_1(k+1)]))\forall \pi_1 k \in ancestors_f(\pi i)$, where $ancestors_f : N(f) \mapsto N(f)$ is defined as follows:

$$\begin{array}{lcl} ancestors_f(i) & = & \emptyset \\ ancestors_f(\pi i) & = & \{\pi\} \cup ancestors_f(\pi) \end{array}$$

It follows that $\exists f'_R \in \mathcal{R}uns_{G,f}$ s.t. $in(lab(f'_R[\pi'(p+1)])) \in T$ and $lab(f'_R[\pi i]) = y$. Suppose that $y \in Y_j$. Since y is part of a run (f'_R), it obviously holds that $\exists c \in \mathcal{S}uf_{\pi i,y}$.

Also, since $rightIgn(lab(f_R[\pi_1(k+1)]))\forall \pi_1 k \in ancestors_f(\pi i)$, we obtain by using the NFA transitions in f'_R at the corresponding steps in our construction that all content models of the elements enclosing πi are right ignoring, thus $r_j \in \pi i.ri$.

Given that $\pi' p < \pi i$ it follows that there is an ancestor of $\pi' p$ which is either (i) a sibling of an ancestor a of πi or (ii) an ancestor a of πi. In any case it follows by using the NFA transitions in f'_R at the corresponding steps in our construction that $\pi' p$ is propagated down at location a until πi, thus $\pi' p \in \pi i.m(y)$.

We have proven thus that $\pi' p \in \pi i.m(y)$, $y \in \pi i.q \cap Y_j$, $\exists c \in \mathcal{S}uf_{\pi i,y}$ and $r_j \in \pi i.ri$.

This completes the proof of Theorem A.1.

$\square$

Theorem 7.3 follows now directly from Theorem A.1.

Appendix B

The Funtional Document Model

The signature of the structure including the FDM code gives an overview of the provided features. The names are mostly self-explaining, however a description of the FDM features is given bellow the signature.

```
signature FDM =
  sig
    type Attribute
    datatype Tree =
      ELEM of int * Attribute vector * Forest
    | TEXT of UniChar.Vector
    | PI of UniChar.Vector * Forest
    withtype Forest = Tree vector

    val emptyTree : Tree
    val emptyForest : Forest

    val hasElementType : Unicode.Vector -> Tree -> bool
    val text : Unicode.Vector -> Tree
    val element : Unicode.Vector -> Attribute vector -> Tree vector
                -> Tree
    val processingInstruction : Unicode.Vector -> Unicode.Vector
                -> Tree
    val isElement : Tree -> bool
    val isText : Tree -> bool
    val isProcessingInstruction : Tree -> bool
    val getElementType : Tree -> Unicode.Vector
    val getElement : Tree -> (Unicode.Vector * Attribute vector
                            * Tree vector)
    val getText : Tree -> Unicode.Vector
    val hasTextContent : Tree -> bool
    val getTextContent : Tree -> Unicode.Vector
    val getProcessingInstruction : Tree -> (Unicode.Vector
                                         * Unicode.Vector)
    val hasAttribute : Unicode.Vector -> Tree -> bool
val getAttribute : Unicode.Vector -> Tree -> Unicode.Vector
val Tree2UnicodeVector : Tree -> Unicode.Vector
val sons : Tree -> Forest

val getAttributes : Tree -> Attribute vector
val getAttributeValue : Attribute -> Unicode.Vector
val getAttributeName : Attribute -> Unicode.Vector
val makeAttribute : Unicode.Vector -> Unicode.Vector
                      -> Attribute
val replaceAttribute : Attribute vector -> Unicode.Vector
                        -> Unicode.Vector -> Attribute vector
val addAttribute : Attribute vector -> Unicode.Vector
                     -> Unicode.Vector -> Attribute vector
val addAttributes : Attribute vector -> Attribute vector
                      -> Attribute vector
val deleteAttribute : Attribute vector -> Unicode.Vector
                        -> Attribute vector
val changeAttribute : Tree -> Unicode.Vector -> Unicode.Vector
                        -> Tree
val insertAttribute : Tree -> Unicode.Vector -> Unicode.Vector
                        -> Tree
val removeAttribute : Tree -> Unicode.Vector -> Tree

val fromList : Tree list -> Forest
val toList : Forest -> Tree list
val length : Forest -> int
val lengthPred : (Tree -> bool) -> Forest -> int
val sub : Forest * int -> Tree
val extract : Forest * int * int option -> Forest
val concat : Forest list -> Forest
val app : (Tree -> unit) -> Forest -> unit
val map : (Tree -> 'a) -> Forest -> 'a vector
val foldl : (Tree * 'a -> 'a) -> 'a -> Forest -> 'a
val foldr : (Tree * 'a -> 'a) -> 'a -> Forest -> 'a
val appi : (int * Tree -> unit) -> Forest * int * int option
            -> unit
val mapi : (int * Tree -> 'a) -> Forest * int * int option
            -> 'a vector
val foldli : (int * Tree * 'a -> 'a) -> 'a
              -> Forest * int * int option -> 'a
val foldri : (int * Tree * 'a -> 'a) -> 'a
              -> Forest * int * int option -> 'a
val deleteAll : (Tree -> bool) -> Forest -> Forest
val deleteFirst : (Tree -> bool) -> Forest -> bool * Forest
val deleteFirstN : int -> (Tree -> bool) -> Forest
                     -> int * Forest
val deleteLast : (Tree -> bool) -> Forest -> bool * Forest
val deleteLastN : int -> (Tree -> bool) -> Forest
                     -> int * Forest
val deleteIth : int -> Forest -> Forest
```

```
    val filter : (Tree -> bool) -> Forest -> Forest
    val filterFirst : (Tree -> bool) -> Forest -> Tree

    val countTreeNodes : Tree -> int
    val countForestNodes : Forest -> int


    val sort : (Tree*Tree -> bool) -> Forest -> Forest
    (*  use (>=,&lt;=) functions to preserve the order
        of elements with equal keys  *)

    val concatForests : Forest vector -> Forest

    val putTree : Tree -> string -> string option -> unit
    val putForest : Forest -> string -> string option -> unit
    val printTree : Tree -> unit
    val printForest : Forest -> unit
  end
```

where:

`Tree` the abstract representation for an XML document or for some part of an XML document

`Forest` a sequence of XML trees (as for example the sons of an element node)

`Attribute` the type of the attributes of XML elements

`emptyTree, emptyForest` values provided as representation of the empty XML tree and forest respectively.

`hasElementType` returns true if the element supplied as the first argument is a node element and has the name supplied as the second argument. If applied on a non node element or if the node element on which it is applied has not this type (name) false is returned.

`text` creates a text node with the specified content.

`element` creates an element node with the name, attributes and the sons vector given as arguments.

`processingInstruction` creates a processing instruction node for a processor given as the first argument and the content given as the second.

`isElement, isText, isProcessingInstruction` returns true if the node given as argument is an element, text or processing instruction respectively.

`getElementType` returns the type of the element node given as argument. If the node is not an element an exception is thrown.

`getElement` returns for an element node a triplet formed of its type, the attributes and the vector of sons. If the node is not an element an exception is thrown.

`getText` returns for a text node the content of the node. If the node is not a text an exception is thrown

`hasTextContent` returns true for a text node or for an element node which only contains a text node

`getTextContent` if applied on a text node returns its content. If applied on an element node which only contains a text node returns the content of this text node. Otherwise an exception is thrown.

`getProcessingInstruction` returns for a processing instruction node a pair containing the corresponding processor and instruction. If the node is not a processing instruction an exception is thrown.

`hasAttribute` applied on an element node given as its second argument returns true if the element has an attribute with the name given as the first argument. If the node argument is not an element an exception is thrown.

`getAttribute` returns for an element node given as the second argument having the attribute given as the first argument the value of this argument. If the node is not an element or if it has not a such attribute an exception is thrown.

`Tree2UniCodeVector` gives the Unicode string representation for a tree

`sons` given a tree, returns the forest of itssons

`getAttributes` returns for an element node a vector containing its attributes. If the node is not an element an exception is thrown.

`getAttributeValue` returns the value of the attribute given as argument.

`getAttributeName` returns the name of the attribute given as argument.

`makeAttribute` creates an attribute with the name given as the first argument and the value given by the second.

`replaceAttribute` returns a vector obtained by replacing in the vector of attributes given as the first argument the attribute with the name and the value given by the second and the third attribute. If there is no attribute with the given name the vector is returned unchanged as the result.

`addAttribute` returns a list obtained by adding to the vector of attributes given as the first argument the attribute with the name and the value given by the second and the third attribute. If the vector has the attribute with the given name already the value of this attribute is set to the given value in the returned vector.

`addAttributes` returns a list obtained by adding to the list of attributes given as the first argument the list of the attributes given as the second argument. If in both lists, attributes of the second list will overwrite those in the first in the returned list.

`deleteAttribute` returns a vector obtained from the vector of attributes given as the first arguments by deleting the attribute with the name given as the second argument. If no attribute with the given name is present in the input vector the vector is returned unchanged as the result.

`changeAttribute` returns for an element node the element node obtained by replacing in its attributes list the value of the attribute given as the first argument with the value given as the third argument. If no attribute with the given name is present in the attribute list of the input element node, it is added to the attributes list of the output node. If the input node is not an element an exception is thrown.

`insertAttribute` returns for an element node the element node obtained by inserting in its attributes list an attribute with the name given as the second argument and the value given as the third. If the attribute with the given name is already present in the attribute list of the input element node, it is changed to the new value in the output node. If the input node is not an element an exception is thrown.

`removeAttribute` returns for an element node the element node obtained by removing from its attribute list the element with the name given as the second argument. If no attribute with the given name is present in the attribute list of the input node, the input node is returned unchanged as the result.

`fromList,length, sub, extract, concat, app, map, foldl,` **foldr,appi, mapi, foldli, foldri** have the SML meaning, given that the type Forest is represented as a vector of Trees.

`toList` transforms a forest in a list of trees

`lengthPred` given a predicate working on trees and a forest, returns the number of trees in the forest that satisfy the predicate.

`deleteAll` given a predicate working on trees and a forest, returns a forest obtained from the input forest by eliminating all the trees that satisfy the predicate.

`deleteFirst` given a predicate working on trees and a forest, returns a pair formed of a boolean and a forest. The boolean indicates if the predicate was fulfilled by any of the trees in the forest. The output forest is obtained from the input forest by eliminating the first tree that satisfy the predicate.

`deleteFirstN` given an integer number, a predicate working on trees and a forest returns a pair formed of an integer and a forest. The output forest is obtained from the input forest by eliminating the first trees that satisfy the predicate, in a number that is not greater than the indicated number. The integer value returned indicates how many trees in the input forest were actually deleted (may be less than the requested number).

`deleteLast` given a predicate working on trees and a forest, returns a pair formed of a boolean and a forest. The boolean indicates if the predicate was fulfilled by any of the trees in the forest. The output forest is obtained from the input forest by eliminating the last tree in the forest that satisfy the predicate.

`deleteLastN` given an integer number, a predicate working on trees and a forest, returns a pair formed of an integer and a forest. The output forest is obtained from the input forest by eliminating the last trees that satisfy the predicate, in a number which is not greater than the indicated number. The integer value returned indicates how many trees in the input forest were actually deleted (may be less than the requested number).

`deleteIth` returns a forest obtained by deleting the i-th tree (i given as the first argument) in the forest given as the second argument.

`filter` given a predicate over trees and a forest returns the trees in the forest that satisfy the predicate.

`filterFirst` given a predicate over trees and a forest returns the first tree in the forest that satisfies the predicate.

`countTreeNodes` counts the nodes in a tree.

`countForestNodes` counts the nodes in a forest.

`sort` given an order relationship as a predicate working on a pair of trees and a forest, returns a forest where trees are sorted such that the predicate order is fulfilled.

`concatForest` concatenates a vector of Forests

`putTree` given a tree, a file name and an encoding option, writes the tree to the specified file in the specified encoding. If the encoding specified is NONE, UTF8 is used.

`putForest` given a forest, a file name and an encoding option, writes the forest to the specified file in the specified encoding. If the encoding specified is NONE, UTF8 is used.

`printTree` given a tree prints it at the standard output.

`printForest` given a forest prints it at the standard output.

Index