Fachbereich Elektrotechnik und Informationstechnik
Institut für Datentechnik
Fachgebiet Industrielle Prozeß- und Systemkommunikation
Prof. Dr.-Ing. Ralf Steinmetz

Technische Universität Darmstadt

Diplomarbeit

Design and Implementation of a Data Manipulation Processor
for an XML Query Language

von

Patrick Lehti
August 2001

Betreuer: Gerald Huck (GMD/IPSI)

Nr. KOM-D-149

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen

Darmstadt, den 27. August 2001 Patrick Lehti

Table of Contents

List of Figures

List of Tables

List of Examples

Akronyms

ABQL	Angle Bracket Query Language
AST	Abstract Syntax Tree
DOM	Document Object Model
DTD	Document Type Definition
OQL	Object Query Language
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language
XQL	XML Query Language
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformations

1 Introduction

XML is a versatile markup language, which has developed over the past few years from its role as markup language for online documents to now also the de facto format for interchanging data between heterogeneous systems. This has in part been stimulated by the growth of the Web and e-commerce. Today nearly every vendor of data management tools has added support for exporting, viewing or even importing XML-formatted data. There are now even native XML document repositories like Software AG's Tamino or ObjectDesign's eXcelon.

As a result, there has been great interest in languages and systems expressing queries over XML data. The World Wide Web Consortium is in the process of developing a standard for an XML query language, called *XQuery*. This query language supports no updates in the current proposal, but updates are necessary in order to evolve XML into a universal data representation and storage format. So it should be possible to modify content within XML documents and to express updates to XML views, which are percolated back to the original data.

This thesis presents an extension for updates to the XQuery language based on a syntax proposal of members of the XQuery working group. It explains the semantics and shows which updates will result in conflicts. Then the implementation of an XML update processor is presented, which is able to evaluate these XQuery updates, including algorithms of transforming, sorting and executing the updates. In the last part of the thesis, this XML update solution is compared to other XML update proposals.

2 Basics

This chapter procures the basic knowledge that is required to understand the following chapters. If you are familiar with these things, you can skip this chapter.

2.1 XML

XML, the *Extensible Markup Language*, is a Recommendation of the World Wide Web Consortium (W3C) from February 1998 [BPSM00]. It was created so that richly structured documents could be used over the Web. But stimulated by the growth of the Web and e-commerce, it is now also the de facto standard for information interchange.

```
<?xml version="1.0"?>

<bib>
     <book year="1994">
          <title>TCP/IP Illustrated</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price>65.95</price>
     </book>
     <book year="1992">
          <title>Advanced Programming in the Unix environment</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price>65.95</price>
     </book>
     <book year="2000">
          <title>Data on the Web</title>
          <author><last>Abiteboul</last><first>Serge</first></author>
          <author><last>Buneman</last><first>Peter</first></author>
          <author><last>Suciu</last><first>Dan</first></author>
          <publisher>Morgan Kaufmann Publishers</publisher>
          <price>39.95</price>
     </book>
</bib>
```

Example 1: An XML document

An XML document consists mainly of *Elements* and *Attributes*. Elements are the most common form of markup. Delimited by angle brackets, most elements identify the nature of the content they surround. Some elements may be empty, in which case they have no content. If an element is not empty, it begins with a start-tag, `<element>`, and ends with an end-tag, `</element>`.

Attributes are name-value pairs that occur inside start-tags after the element name. For example, `<book year="1994">` is a `book` element with the attribute `year` having the value `1994`. In XML, all attribute values must be quoted. Beside of Elements and Attributes you can also add *Comments*, *Processing Instructions*, *Entity Definitions*, *Entity References*, *CDATA Sections* and *Document Type Declarations* to an XML document. These objects can be seen as

the nodes of the XML tree.

If an XML document obeys all syntax rules for XML, like that every start tag must have a corresponding end tag and there must be exactly one root element, it is called *well-formed*. A document can also include or be associated with a Document Type Declaration(*DTD*) or an XML *Schema*. Such a DTD provides a grammer, a Schema provides a structure for the XML document, that means it specifies the elements and attributes that may occur in this document.

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

Example 2: A DTD for Example 1

If an XML document contains a proper document type declaration and if the document obeys the constraints of that declaration, like element sequence and nesting is valid, required attributes are provided and attribute values are of the correct type, it is called *valid*.

Beside of the XML specification there are other standards that are associated with XML. So there are e.g. XML Schema for providing structures for XML documents (see [Fal01], [TBMM01], [BM01]), *XSL* and *XSLT*, for transforming and presenting XML, *XLink* and *XPointer*, for linking XML sources together, *XPath* [CD99], to identify nodes in an XML document and *Namespaces* [BHL99], for qualifiying element and attribute names.

2.2 XQuery

With the increasing amount of stored XML data the need for an XML query language becomes more and more important. So for human-readable documents you may e.g. want to search for structures or information in structures found within the document, generate table of contents or generate new documents as the result of a query. For data-oriented documents like database data or object data you may e.g. want to extract data from these sources, transform data into new XML representations, or integrate data from multiple heterogeneous data sources.

For this purpose the W3C is currently developing an XML query language, called XQuery [CCF+01]. This has still the status of a Working Draft, but the main concepts seem to be stable. It is not fixed to one special syntax, so at the moment there exists a human-readable and an XML syntax, called *XQueryX* [MRS01]. XQuery is derived from an XML query language called Quilt and borrowed features and ideas from other languages like XPath, *XQL*, *XML-QL*, *SQL*, *OQL*, *Lorel* and *YATL*.

2.2.1 Path Expressions

One main concept of XQuery is the use of XPath expressions for selecting nodes. XPath is a language to address parts of an XML document. XPath gets its name from its use of a path notation as in *URL*s for navigating through the hierarchical structure of an XML document. A path expression consists of a series of "steps". Each step represents movement through a document along a specified "axis", and each step can apply one or more predicates to eliminate nodes that fail to satisfy a given condition. The result of each step is a sequence of nodes that serves as a starting point for the next step.

```
document("zoo.xml")//chapter[2]//figure[caption = "Tree Frogs"]
```

Example 3: An XPath query

The above example uses a path expression consisting of three steps. The first step locates the root node of a document. The second step locates the second chapter of the document. The third step finds figure elements occurring anywhere within the chapter, but retains only those figure elements that have a caption with the value "Tree Frogs."

2.2.2 Element Constructors

In addition to searching for elements in existing documents, a query often needs to generate new elements. The simplest way to generate a new element is to embed the element directly in a query using XML notation. In other words, one of the permitted forms of an XQuery expression is an XML element that represents itself.

So the following example creates a new "emp" element. The value of the attribute and the content of the element are specified by variables that are bound in other parts of the query.

```
<emp empid = {$id}>
     {$name}
     {$job}
</emp>
```

Example 4: An Element Constructor query

2.2.3 FLWR Expressions

A FLWR (pronounced "flower") expression is constructed from FOR, LET, WHERE, and RETURN clauses, which must appear in a specific order. A FLWR expression binds values to one or more variables and then uses these variables to construct a result. A FOR-clause is used whenever iteration is needed. Each variable in a FOR-clause can be thought of as iterating over the values returned by its respective expression, in document order. A LET-clause is also used to bind one or more variables to one or more expressions. Unlike a FOR-clause, however, a LET-clause simply binds each variable to the value of its respective expression without iteration, resulting in a single binding for each variable.

The binding-tuples generated by the FOR and LET clauses are subject to further filtering by an optional WHERE-clause. Only those tuples for which the condition in the WHERE-clause is true are used to invoke the RETURN clause. The RETURN-clause generates the output of the FLWR expression, which may be any sequence of nodes or primitive values. The RETURN-clause is executed once for each tuple of bindings that is generated by the FOR and LET-clauses and satisfies the condition in the WHERE-clause, preserving the order of these tuples.

```
FOR $b IN document("bib.xml")//book
WHERE $b/publisher = "Morgan Kaufmann"
AND $b/year = "1998"
RETURN $b/title
```

Example 5: A FLWR query

The above query lists the titles of books published by Morgan Kaufmann in 1998 from a document called "bib.xml".

2.2.4 Other features

Additionally to the presented expressions the XQuery language has more features, like sorting expressions, conditional expressions, quantified expressions and functions. For more details see [CCF+01].

2.2.5 XQueryX

XQueryX [MRS01] is an XML syntax for XQuery. This syntax is not particularly convenient for humans to read and write, but it is easy for programs to parse, and because XQueryX is represented in XML, standard XML tools can be used to create, interpret, or modify queries.

The following example shows the same query as in example 5, but in the XQueryX syntax.

```
<q:query xmlns:q="http://www.w3.org/2001/06/xquerx">
  <q:flwr>
    <q:forAssignment variable="$b">
      <q:step axis="DESCENDANT">
     <q:function name="document">
       <q:constant datatype="CHARSTRING">bib.xml</q:constant>
     </q:function>
     <q:identifier>book</q:identifier>
      </q:step>
    </q:forAssignment>
    <q:where>
      <q:function name="AND">
     <q:function name="EQUALS">
       <q:step axis="CHILD">
         <q:variable>$b</q:variable>
         <q:identifier>publisher</q:identifier>
       </q:step>
       <q:constant datatype="CHARSTRING">Morgan Kaufmann</q:constant>
     </q:function>
     <q:function name="EQUALS">
       <q:step axis="CHILD">
         <q:variable>$b</q:variable>
         <q:identifier>year</q:identifier>
       </q:step>
       <q:constant datatype="CHARSTRING">1998</q:constant>
     </q:function>
      </q:function>
    </q:where>
    <q:return>
      <q:step axis="CHILD">
     <q:variable>$b</q:variable>
     <q:identifier>title</q:identifier>
      </q:step>
    </q:return>
  </q:flwr>
</q:query>
```

Example 6: An XQueryX query

3 Updating XML

This chapter now presents requirements and usage scenarios for updating XML. XML updates are necessary to make XML a full-featured data storage format, i.e. if you have large documents stored in databases, where you want to modify only small parts of data. The following content was influenced by [AQM+97], [CFMR01a], [CFMR01b], [Mar00] and [TIHW01].

3.1 Desiderata

A good XML update solution must comply with some requirements. These requirements are presented in this section. To differentiate the importance of the various requirements I use the words *must*, *should* and *may* as done in the XQuery requirements paper.

- An XML update solution *must* be able to execute these updates independent of the physical representation of the data, that means it *must* process updates on the filesystem, on native XML databases and on XML views of other data sources like relational databases.
- Updates *must* be possible on well-formed and valid documents. They *need not* be possible on not well-formed documents. The effects of an update query *must* result in an well-formed document. Updates that violate the Schema of a valid document *may* result in an error or warning, but this is out of scope for this thesis.
- The update query *must* be able to select any kind of nodes within an XML document for the target of the update. These nodes are elements, attributes, comments, processing-instructions, text nodes or document nodes.
- It *must* be possible to create new nodes within a query, i.e. elements, attributes, comments, processing-instructions and text nodes. It *should* be possible to create new document nodes.
- The query language *must* be able to insert new nodes at any position in the XML tree. That means for ordered node lists, like the children of an element, insertions before or after a child node or at the end of the list. For unordered lists, such as attribute lists, it means always appending to the list. Attribute insertions of attributes with a name that already exists in this attribute list *must* fail.
- The query language *must* be able to delete any node including its children out of the XML tree. This is often called a deep delete. It *should* also be able to delete documents.
- The query language *should* be able to rename elements and attributes. Rename operations preserve the identity of a node, and are so different from a delete and insert of the renamed node. Identity could e.g. mean the same memory address for transient XML objects or object-ids for database objects.
- The query language *may* offer more functionality for often used operations. These could be seen as shortcuts for sequences of the elementary insert, delete and rename operations. These operations could be e.g. replace or move operations, which are shortcuts for inserts followed by delete operations.
- It *should* be possible to modify Namespaces, like namespace URIs and namespace prefixes.

- It *should* be possible to specify more than one update in a query.
- The update processor *should* report errors in the update queries. These errors can be e.g. type errors, like attribute insertion before elements, or conflicts between updates, like a replace and delete of the same node.

3.2 Usage Scenarios

This sections presents use cases for an XML update language in an abstract manner. Concrete use cases with solutions in the extended XQuery language can be found in Appendix B.

- Deletion of elements, attributes, text nodes, comments or processing-instructions.
- Insertion of new elements, text nodes, comments or processing-instructions at a specific position in the XML Tree, i.e. in, before or after other nodes.
- Insertion of new attributes into the attribute list of an element.
- Renaming of elements, attributes and namespaces.
- Copying and inserting of existing elements, attributes, text nodes, comments or processing-instructions at a different position.
- Moving of elements, text nodes, comments or processing-instructions within a list of children.
- Moving of elements, attributes, text nodes, comments or processing-instructions into other elements.
- Replacing of elements, text nodes, comments or processing-instructions with other nodes of any kind, except attributes.
- Replacing of attributes with other attributes.
- Modifying the structure of the XML tree by e.g. flatten or deepen the structure, which means inserting or removing additional elements, or promote or demote nodes, which means increasing or decreasing of the hierarchy level of a node.
- Updating the value of an element or attribute, where the value of an element means a single text node.
- Updates based on other data in the document, e.g. increasing number values based on the old value.
- Creation of new documents out of other documents, i.e. merging the data of several documents and store the results in a new document.
- Updates based on conditions, that means an update is only processed if a conditional expression evaluates to true.
- Creation, deletion, or renaming of documents.

4 Update extensions for XQuery

This chapter shows an extension for the XQuery language to support updates. It is based on the requirements and use cases presented in the previous chapter.

The presented proposal is based on an update extension proposal from members of the XQuery working group and on the XQuery specification from June 7th, 2001. When adding these update expressions to the original XQuery grammar, some modifications to the syntax proposal were made in order to remove ambiguities. The grammar of the extended XQuery and XQueryX syntax are found in appendix A.

4.1 Syntax

Every update expression starts with the keyword "UPDATE". This is one of the modifications to the original proposal. The following sections are taken out of the syntax proposal with only some modifications respecting my syntax changes.

The *Insert*, *Delete*, *Replace* and *Rename* expressions do the modifications to documents. Each of these expressions identifies a set of nodes to be operated on, and applies an update to each node in the set. Changes made by the update statement itself are not taken into account when identifying these nodes. Mulitple updates can be composed in one query.

4.1.1 Insert

The insert expression inserts content at positions identified by the "PRECEDING" and "FOLLOWING"(formerly "BEFORE" and "AFTER") clauses. Consider the following example:

```
UPDATE
INSERT <warning>High Blood Pressure!</warning>
PRECEDING //blood_pressure[systolic>180]
```

Example 7: Inserting an element preceding other nodes

The "PRECEDING" or "FOLLOWING" clause identifies the set of nodes. For each node in the set, a new sibling node is inserted. The following expression is the same as the previous one, except that the <warning/> element is inserted after the corresponding <blood_pressure/> element:

```
UPDATE
INSERT <warning>High Blood Pressure!</warning>
FOLLOWING //blood_pressure[systolic>180]
```

Example 8: Inserting an element following other nodes

"PRECEDING" and "FOLLOWING" are undefined for unordered lists, such as attribute lists. The "INTO" clause allows content to be inserted into a position without requiring order. The following expression inserts a warning attribute into <blood_pressure/> elements:

```
UPDATE
INSERT ATTRIBUTE warning { "High Blood Pressure!" }
INTO //blood_pressure[systolic>180]
```

Example 9: Inserting an attribute into elements

If "INSERT INTO" is used with an ordered list, it appends to the end of the list. For instance, the following expression inserts a <warning/> element as the last child of the corresponding <blood_pressure/> element:

```
UPDATE
INSERT <warning>High Blood Pressure!</warning>
INTO //blood_pressure[systolic>180]
```

Example 10: Inserting an element into other elements

4.1.2 Delete

The "DELETE" expression deletes nodes. It contains a subexpression, which identifies the nodes that are to be deleted. For instance, the following expression deletes blood pressure nodes with high systolic readings:

```
UPDATE
DELETE //blood_pressure[systolic>180]
```

Example 11: Deleting nodes

4.1.3 Replace

The "REPLACE" expression replaces nodes. The entire node is replaced, not just the content of the node. Consider the following:

```
UPDATE
REPLACE //job[.="bit banger"]
WITH <profession>Computer Scientist</profession>
```

Example 12: Replacing nodes

4.1.4 Rename

The "RENAME" expression changes the name of an element or attribute, or the target of a processing-instruction. The following expression changes the name of <job/> elements, but does not change their content:

```
UPDATE
RENAME //job[.="bit banger"] AS "profession"
```

Example 13: Renaming nodes

4.1.5 Composition

A query can also be a composition of several updates, where the order of the execution of the single updates is undefined and the effect of one update is not taken into account when evaluting the other updates.

```
UPDATE
RENAME //blood_pressure[systolic>180] AS "high_blood_pressure"
INSERT <warning>High Blood Pressure!</warning>
INTO //blood_pressure[systolic>180]
```

Example 14: A composition update

4.1.6 Conditional Updates

Conditional updates are done using "IF/THEN/ELSE", which is an different expression than the "IF/THEN/ELSE" expression in standard XQuery. So for this conditional update expression the "ELSE" clause is optional in contrast to the "IF/THEN/ELSE" expression in standard XQuery. The following expression implements a gender-based downsizing policy:

```
UPDATE
IF (count(//employee)>1000) THEN
     IF (count(//employee[sex=male]>100)) THEN
          DELETE //employee[sex=male]
```

Example 15: A conditional update

4.1.7 FLW-Update expressions

FLW-Update expressions are similar to "FLWR expressions" and allow more sophisticated updates. This expression has the same form as a "FLWR expression", except that the "RETURN" clause is replaced by a list of update expressions. For instances, the following query replaces screwdrivers with hammers on a set of invoices:

```
UPDATE
FOR $i IN //invoice
LET $s := $i//product[name="screwdriver"]
WHERE not (empty($s))
INSERT
     <product>
          <name>Hammer</name>
          <price>15.40</price>
     </product>
PRECEDING $s
DELETE $s
```

Example 16: A FLW-Update

FLW-Update expressions may be nested, which allows quite sophisticated updates. Consider the following, which sets the price of a hammer to twice the price of the screwdriver that it replaces:

```
UPDATE
FOR $i IN //invoice
     LET $s := $i//product[name="screwdriver"]
     FOR $a IN $s
          DELETE $a
          INSERT
               <product>
                    <name>Hammer</name>
                    <price>{ 2*$a/price }</price>
               </product>
          PRECEDING $a
```

Example 17: A nested FLW-update

4.2 Semantics

This semantics chapter explains the meaning of the different update expressions. This includes allowed types and expected results.

4.2.1 Insert

The "INSERT PRECEDING" and "INSERT FOLLOWING" expressions can be explained together, because the semantics is the same except for the target of the expressions.

```
INSERT expr1 PRECEDING expr2
INSERT expr1 FOLLOWING expr2
```

"expr1" can be an element, comment, processing-instruction, text node or a set of these nodes. The same applies for "expr2":

```
expr1,expr2 :: (CElem | CMisc | CString)+
```

The expected result is that *all* nodes of "expr1" are copied and inserted before or after *every* node of "expr2" as a sibling. An exception is when "expr2" is the root element of a document, then only comments and processing-instructions are allowed in "expr1".

The "INSERT INTO" expression is defined for a different set of expression types.

```
INSERT expr1 INTO expr2
```

Here "expr1" can be an element, attribute, comment, processing-instruction, text node or a set of these nodes. "expr2" can only be an element, document node or a set of these nodes:

```
expr1 :: (CElem | CAttr | CMisc | CString)+
expr2 :: (CElem | Document)+
```

The meaning is that *all* elements, comments, processing-instructions and text nodes of "expr1" are appended to the list of children of *every* node in "expr2" and *all* attributes in "expr1" are appended to the list of attributes of *every* node in "expr2". If "expr2" is a document node then you have to check if this update does not result in more than one root node in this document. If you insert attributes, you have to check that the nodes in "expr2" are *all* elements and that there are *no* other attributes with the same name in these elements.

4.2.2 Delete

The "DELETE" update has only one expression.

```
DELETE expr
```

The "expr" can be of any type inclusive set of nodes:

```
expr :: (Content | Document)+
```

It results in removing *every* node of the "expr" out of the XML tree. One exception is the deletion of the root node of a document, which should not be allowed, because it would result in a not well-formed document.

4.2.3 Replace

The "REPLACE" update has one expression for the target and one for the replacement.

```
REPLACE expr1 WITH expr2
```

The "expr1" can be an element, attribute, comment, processing-instruction, text node or a set of these nodes. The same applies for "expr2":

```
expr1,expr2 :: (CElem, CAttr, CMisc, CString)+
```

It means that *every* node of "expr1" is replaced with *all* nodes of "expr2", where the entire node is replaced, not just the content of the node. Note that attributes can only be replaced by other attributes and check if there are not attributes with the same name in one element after the replacement. Check if the root node is replaced, it can only be replaced by *one* other element.

The "REPLACE" expression can be seen as a short cut for a "INSERT INTO" expression followed by a "DELETE" expression for replacing attributes and as a "INSERT PRECEDING" expression followed by a "DELETE" expression for other nodes. The only exception for this is the replace of the root element, where a "INSERT PRECEDING" would violate the one-root-element constraint.

4.2.4 Rename

The meaning of the "RENAME" update is as follows:

```
RENAME expr1 AS expr2
```

"expr1" of the "RENAME" update can be an element, attribute, document node or a set of these nodes. "expr2" *must* evaluate to a String:

```
expr1 :: (CElem, CAttr, Document)+
expr2 :: CString
```

The result is that *every* element name, attribute name or document name of "expr1" is changed to the value of "expr2" by preserving the identity of the node.

4.3 Conflicts

The update extension for XQuery allows several update operations in one query. This results in possible conflicts between the updates. A conflict is either an undefined result, e.g. when a "replace" and a "delete" update operate on the same node or updates that affect nested nodes of other updates, e.g. an update in a nested node and a delete of the entire subtree.

The following table shows the compatibility of different updates on the same node:

	Insert	**Delete**	**Replace**	**Rename**	**Update of nested node**
Insert	ok	ok	ok	ok	ok
Delete		conflict	conflict	conflict	conflict
Replace			conflict	conflict	conflict
Rename				conflict	ok

Tabelle 1: Conflict table

"insert" updates *never* causes conflicts when used together with other updates of the same node, because the insertion nodes are always copies of the old node and the target of an "insert into" is just an update of a nested node. A "delete" update *always* results in a conflict when used together with other updates on the same node. The reason for this is obvious: you cannot perform an update on a node that does not exist anymore. The same applies to the "replace" expression, because it is only a shortcut for an "insert" followed by a "delete". Two "rename" updates on the same node cause an undefined result and that means also a conflict.

5 Implementation

After you learned about updating XML and the update extensions for XQuery, this chapter now presents an implementation and algorithm for an XML update processor.

5.1 Architecture

This implementation is an extension of an existing XQuery processor prototype, called QuiP, which was developed by Sven Eric Panitz and Patrick Lehti as a research project at Software AG. It was required to use this prototype for update processing, because the update expressions include standard XQuery expressions which must be evaluated. So I can use the existing code for this evaluation and I also get a GUI.

The architecture is split in two parts. The first part is the query compiler, which parses the query and generates XQueryX code out of it. (XQueryX was formerly known as *ABQL* and is still called so in the implementation.) The XQueryX syntax seems to be more stable than XQuery and because it is an XML syntax it is easier to parse, this makes it easier for the executor to handle new query syntax. This first part is written in Java, because the XQuery working group provides a *JavaCC* grammar for XQuery.

The second part is the actual executor, which parses the ABQL query, reads the XML documents, executes the queries on the data and writes the results back to the documents. This second part is written in CLEAN [KPES00], a functional programming language, which is similar to Haskell. Functional programming languages help focus on the algorithms, instead of worrying too much about implementation details. A functional program can be regarded as an executable specification.

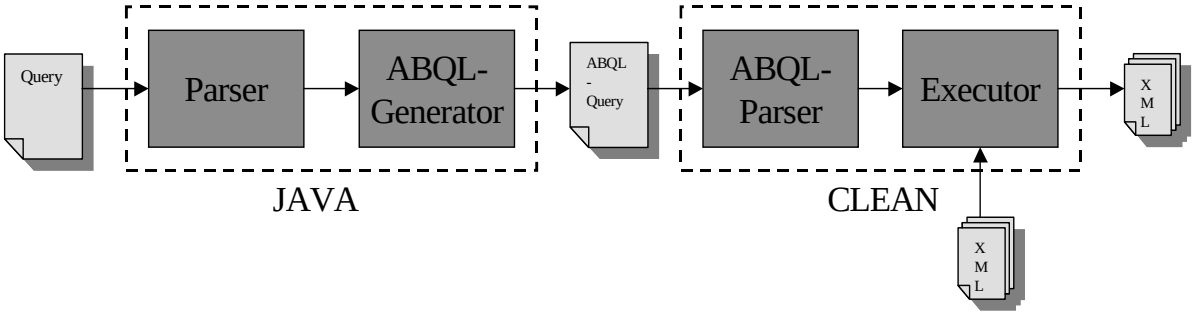

Figure 1: Architecture

The above figure shows the architecture. A query can be either a file, when using the command line interface or a Java-String from the GUI. The result of the query compiler is a file, which contains the ABQL query. Then the CLEAN ABQL parser reads this file. The executor reads the required XML documents and performs the updates on the data. The result of this are modified XML documents. The temporary file for the ABQL query is needed, because there is no existing interface between Java code and CLEAN code. This performance critical task is

acceptable, because it is only a prototype, where the overall performance is not regarded.

5.2 Query Compiler

The query compiler is the Java part of QuiP. It parses a query and generates ABQL code for it.

5.2.1 Parser

The parser is a JavaCC [MS00] generated parser. The Java Compiler Compiler (JavaCC) is one of the most popular parser generators for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. JavaCC is a so called *LL(k)* parser generator in contrast to *LALR* parser generator like e.g. *Yacc*. LL(k) parsers have a better error handling and inside a rule you can get information about the actual context.

The XQuery working group provides a LL(1)-JavaCC grammar for XQuery. This grammar is extended with some Java code for building an abstract syntax tree. This AST allows it to perform further actions on the data. For programming languages it is often used to do type checking and code generations. For QuiP it is used to generate the ABQL code.

So the tasks to enable updates for the parser, is adding rules for the update expressions into the JavaCC grammar file (see appendix A.2.). Then create one new class for every update expression for the AST and add Java code to the JavaCC rules, which creates instances of these update classes and builds the tree.

5.2.2 ABQL Generator

The ABQL Generator is implemented as a visitor on the AST. A visitor is a design pattern to perform actions on trees. It is presented in [GHJV94]. The XQuery2ABQLVisitor walks the complete AST and generates an XML *DOM* tree by creating a DOM object for every node in the tree. This DOM tree is then stored in an XML file.

The task for me was simply to add new methods to the visitor for every new class of the AST, which generates the corresponding XML nodes for the ABQL representation.

5.3 Executor

The executor is the CLEAN part of QuiP. It consists of a few modules for parsing and representing XML, parsing and representing ABQL, reading and writing documents, executing the queries and some more. This chapter presents and explains only the chapters that are important regarding updates.

5.3.1 XML Data Model

The XML data model in the QuiP executor is specified in the "XmlTypes" module. These types are used in many other modules. The following picture shows the main classes of this data model:

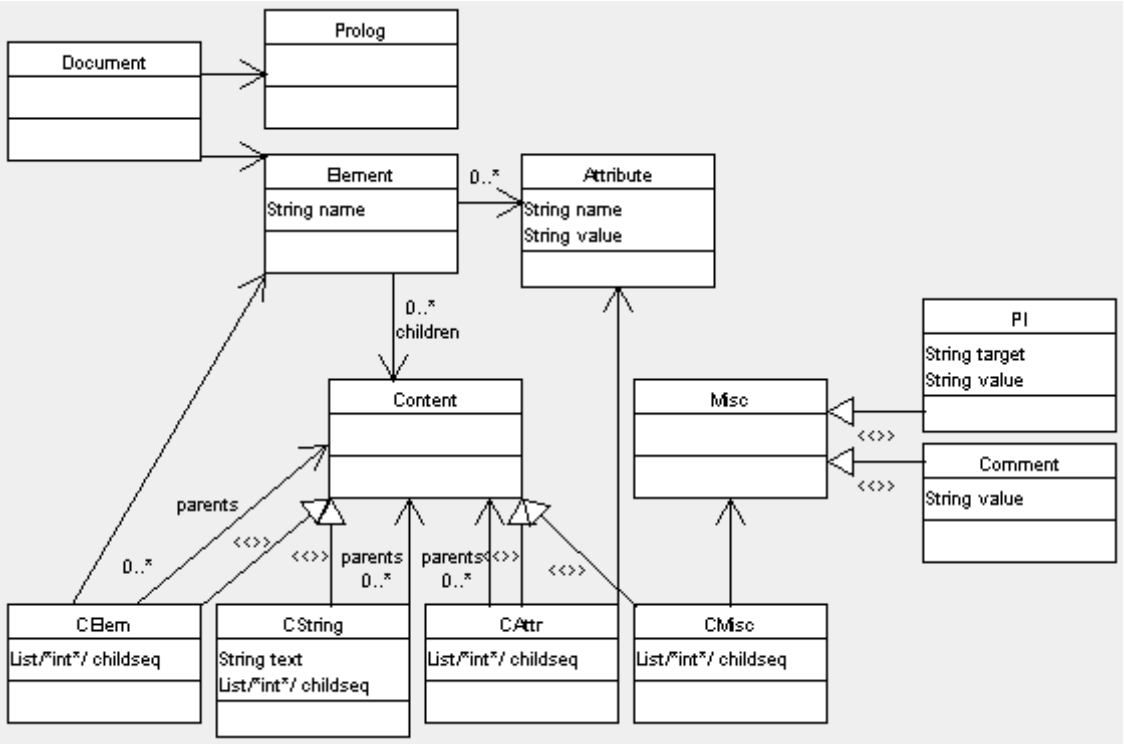

Figure 2: XML data model

The "Content" type is the super class of all nodes. The nodes itself are wrapped into classes starting with the prefix "C". These wrappers provide information about the parent node and the position of this node in the entire XML tree. This position is given in the form of a child sequence, e.g. the third child of the second element of the root element would have the child sequence: [1,2,3].

5.3.2 ABQL Types

The "AbqlTypes" module defines a representation for an ABQL query. The Abql type represents the whole query, it has a list of document declarations, a list of function definitions and one query or update expression. The following UML [OMG01] class diagram shows the structure of an Abql query and the seven update expressions with their relations to normal expressions. These expressions can be any XQuery expression, e.g. path expressions, flwr expressions or element constructors.

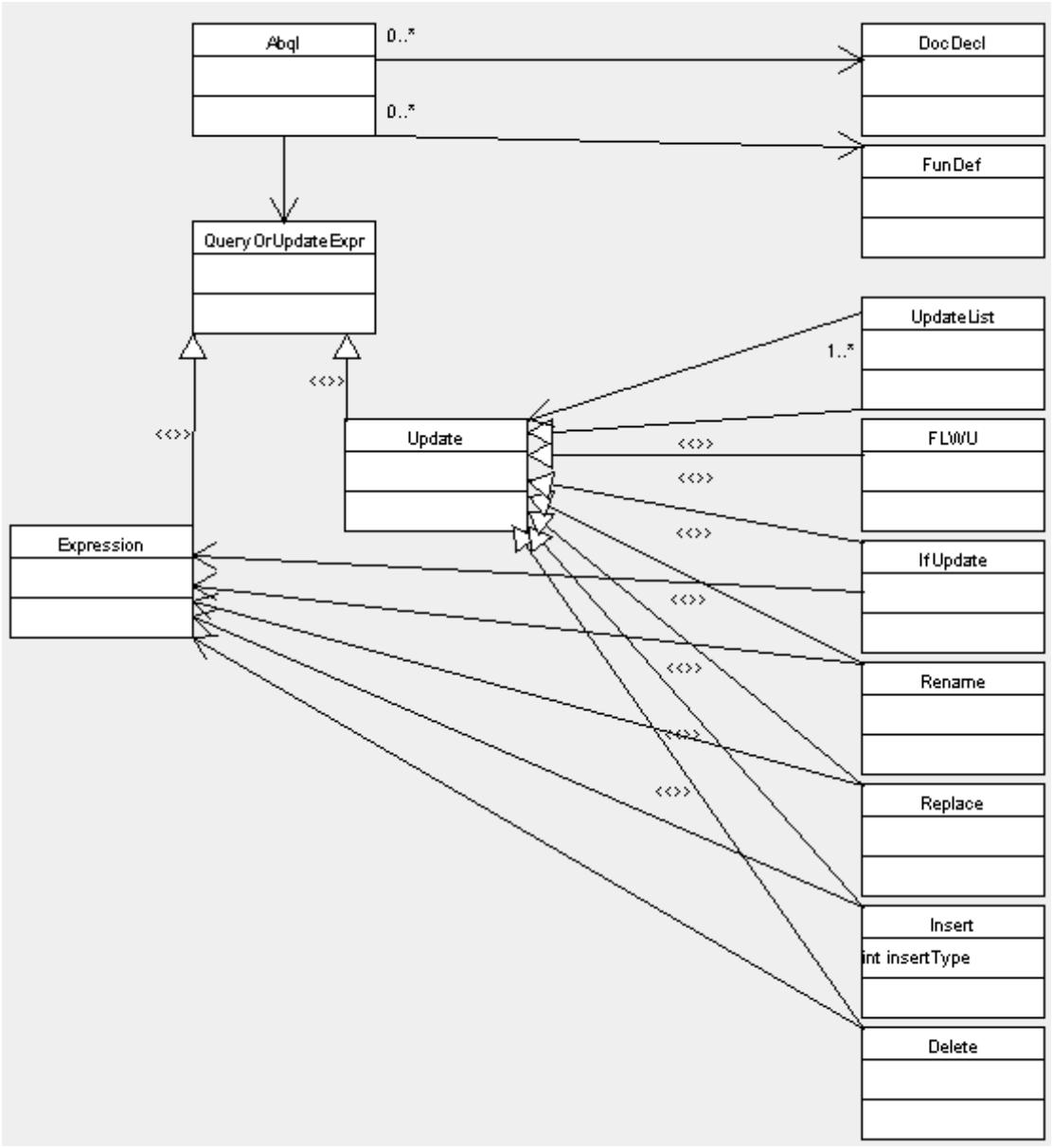

Figure 3: Class diagram ABQL types

5.3.3 ABQL Parser

The "Abql" module is the Abql parser. In a first step the XML document is parsed by an XML parser and then this XML representation is parsed to create an Abql type instance. This parser is a monadic parser. *Monads* are a pattern used in functional programming. This pattern can be used for modelling a number of different tasks, like imperative sequences of I/O statements, error handling and parsing. The monadic parser library used for reading ABQL provides higher order combinators to express a parser directly in terms of the provided DTD. You find more details about monads and monadic parsers in [HM93] and [Pan96].

5.3.4 Evaluating Updates

After the query is parsed and an Abql instance is created, the "execute" module comes into operation. If the processed query is identified as an update query then the evaluation method in the "EvalUpdate" module is called. This module is the first step in processing an update. It evaluates the XQuery expressions in the query, checks the type of these expressions and converts the update into an internal simplified update representation, called *CoreUpdates*.

The evaluation is simply done by calling the execute method of the original XQuery "execute" module. This results in an instance of a so called *QuiltResult*. This QuiltResult can be of different types, as you see in the following picture:

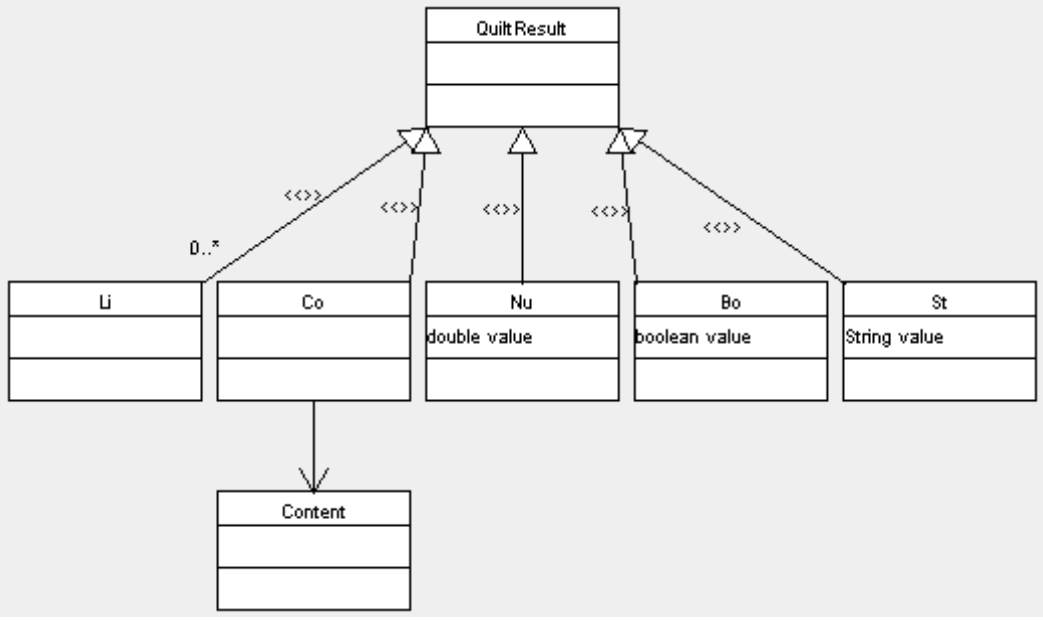

Figure 4: Class diagram QuiltResult

A QuiltResult is either a list of QuiltResults(Li), a numeric value(Nu), a boolean value(Bo), a String value(St) or a Content(Co). Content is the super class for all XML nodes as you can see in figure 2. The type of this QuiltResult is checked regarding the semantic rules found in chapter 4.2.

If the results passed the type check, then the update is transformed to a CoreUpdate:

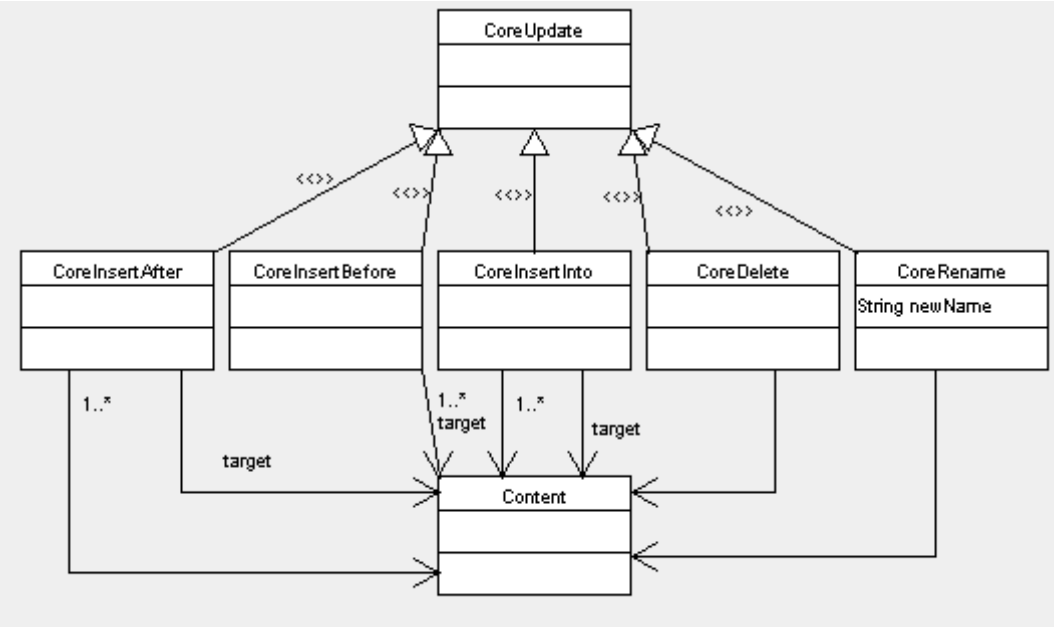

Figure 5: Class diagram core updates

A CoreUpdate is either one of the insert updates, a delete or a rename. An insert update has a list of Content to be inserted and *one* target Content. The delete has *one* Content to be deleted and the rename has *one* Content to be renamed and a String that specifies the new name. The original update queries are mapped to these core updates by using some simple rules:

- Replace the query expressions by their value. If the value is not a Content or list of Content then wrap it into a "CString" Content.
- Create one CoreUpdate for every target in the set of targets in the original query.
- Transform the "REPLACE" expression to the equivalent insert and delete expressions as described in the chapter 4.2.

The transformation of the query into this simplified representation is needed for efficiently sorting and executing the operations and to detect conflicts.

5.3.5 Sorting CoreUpdates

The sorting of the CoreUpdates is a main part of the update algorithm. It meets several tasks. So for one reason the sorting is absolutely necessary, when "insert before" or "insert after" and a "delete" expression affect the same node. Then all inserts have to be done before the delete comes into operation. The next required feature of the sorting is the detection of conflicts. All "delete" and "rename" conflicts can be detected during sorting the CoreUpdates. The detection of the "delete" - "nested node update" conflict is done during execution.

Additionally the sorting allows an optimization by executing all updates regarding one node together. The sorting is also necessary because of a characteristic of functional programming

languages, like CLEAN. In functional programming language you cannot make assignments to variables, that means you cannot make updates. So you have to create new objects with the new values instead and this for the whole path in the tree. Regarding this, it is necessary to update from the leafs to the root to resolve all dependencies before updating a node.

The CoreUpdates are stored in this ordered structure:

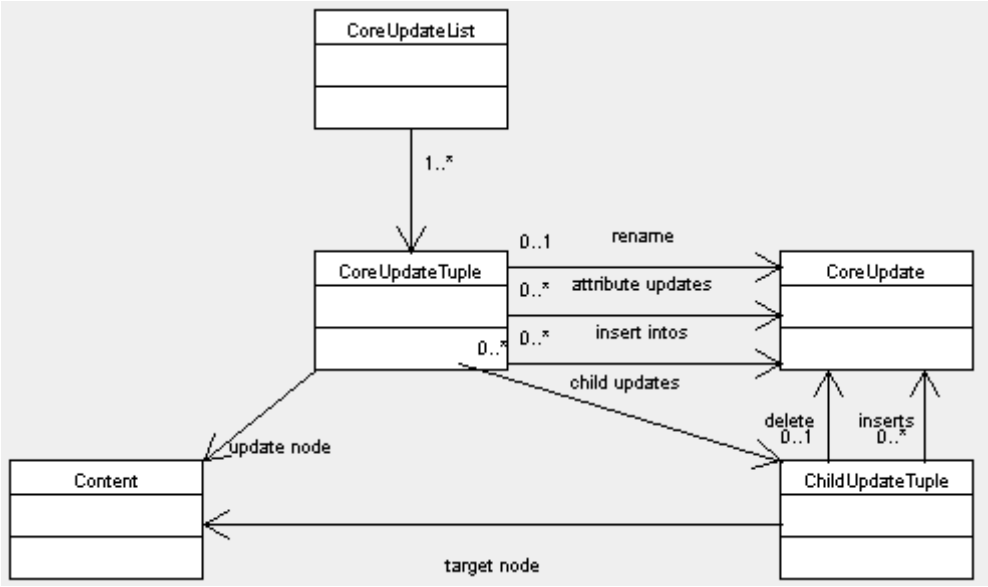

Figure 6: Class diagram update list

The CoreUpdateList is ordered list of CoreUpdateTuple. A CoreUpdateTuple combines all CoreUpdates that affect the same node. It is ordered for the functional programming language problem. Such a CoreUpdateTuple has maybe a rename update, a list of attribute updates and list of ChildUpdateTuples. ChildUpdateTuple combines all "insert after", "insert before" and "delete" updates that affect one child.

The sorting of the CoreUpdateList is done during adding the CoreUpdates, so possible conflicts are detected immediately. When a new CoreUpdate is added to the list, the following steps are performed to find the correct place for it:

1. Iterate through the update list and compare always the document node of the update node in the CoreUpdateTuple with the document node of the target node in the new CoreUpdate. If they are equal go on with 2., if the update list as no more items, then go to 3.
2. Compare the child sequence of the update node in the CoreUpdateTuple with the child sequence of the target node in the new CoreUpdate. The algorithm for this comparison is that nodes nearer to the root are greater and siblings were ordered in natural order. Consider this code fragment, which shows this comparison algorithm:

```
compareContentPos :: [Int] [Int] -> Int
compareContentPos [a:as] [b:bs]
      |a > b = 1
      |a < b = -1
            = compareContentPos as bs
compareContentPos [] [_:_] = 1
compareContentPos [_:_] [] = -1
```

```
compareContentPos [] [] = 0
```

If the child sequence of the new CoreUpdate is equal(0) then go to 4., else if the child sequence is less(-1) then go to 3., if the child sequence is greater go on with 1.

3. Create a new tuple and insert it at the current position to the list.
4.
   4.1. If the new CoreUpdate is a rename and this CoreUpdateTuple has already a rename, then a rename conflict is detected, otherwise insert the CoreUpdate to the tuple.
   4.2. If the new CoreUpdate is an attribute update, then append it to the list of attribute updates.
   4.3. If the new CoreUpdate is an "insert into" update, then append it to the list of insert into updates.
   4.4. If the new CoreUpdate is an "insert after", "insert before" or "delete" update then iterate through the list of ChildUpdateTuple and compare the target position of the new CoreUpdate with the position of the ChildUpdateTuple target. If they are equal then add it to this tuple. If the new update is a delete and there is already a delete for this target then a delete conflict is detected. If there are no more tuple in the list, create a new tuple and add it to the list.

The sorting at item 2 is only needed because of the functional programming language problem with updates as mentioned earlier.

5.3.6 Executing CoreUpdates

After all updates are evaluated and the resulting "CoreUpdates" are stored in the "CoreUpdateList", the updates were executed tuple per tuple. One *CoreUpdateTuple* is executed by performing all children updates on the children, performing all attribute updates on the attributes and performing the possible rename update on the name of the element. Then a new element is created out of the new name, new attributes and new children and two new CoreUpdates are added to the "CoreUpdateList": a "delete" of the old element and a "insert before" of the new node(this is only needed because of the functional programming language characteristic!). If a conflict is detected during adding the "delete" of the old node, this signalizes a "delete" - "nested node update" conflict. See also the following code fragment:

```
executeCoreUpdateTuple :: CoreUpdateTuple *CoreUpdateList -> (Either String Content,
*CoreUpdateList)
executeCoreUpdateTuple (theNode=:(CElem (Elem name attrs childs) _ _), maybeRename,
attrUpdates, (childUpdates, insertIntos)) restUpdates
 //execute child updates
 # newChilds = executeChildUpdateTuples childs childUpdates
 # newChilds = executeChildInsertIntos newChilds insertIntos
 //execute attribute updates
 # newAttrs = executeUpdatesOnAttrs attrs attrUpdates
 //execute rename
 # newName = executeRename name maybeRename
 //create new element
 # newElem = cElem$Elem newName newAttrs newChilds
```

```
 //delete the old node
 # restUpdates = addCoreUpdate (CoreDelete theNode) restUpdates
 |isRight restUpdates
   //insert newNode before oldNode
   # restUpdates = addCoreUpdate (CoreInsertBefore [newElem] theNode) (getRight
restUpdates)
   |isRight restUpdates
     = (Right newElem, getRight restUpdates)
   //else
     = (Left$getLeft restUpdates, [])
 //else if delete of old node fails then conflict:
   = (Left "Conflict between two updates on nested nodes", [])
```

The *rename update* simply changes the name of the element, the *attribute updates* are a bit more sophisticated. The list of attribute updates is not ordered because the set of attributes is not ordered. So for every attribute update you have to check the complete set of attributes. A "rename" update simply changes the name of the attribute, the "delete" update removes this attribute from the attribute set. "insert into" updates simply append the new attribute to the set of attributes. Consider the following code fragment:

```
executeUpdateOnAttrs :: [Attribute] CoreUpdate -> [Attribute]
executeUpdateOnAttrs [a:as] b
 //if update is insert into then append new attributes to the list
 |isInsertInto b
   = [a:as]++(map getAttr (getInsertContent b))
 //if the attribute is found then
 |isTheAttr (getAttrName b) a
   //execute the update
   # newA = executeUpdateOnAttr a b
   //and merge the results
   = newA++as
 //else search the rest of the attribute list for the right attribute
   = [a:(executeUpdateOnAttrs as b)]
 where
   getAttrName (CoreDelete (CAttr (name, _) _ _)) = name
   getAttrName (CoreRename (CAttr (name, _) _ _) _) = name
   getAttrName _ = abort "AttrName not defined for this CoreUpdate"
   //the updateing of the found attribute:
   executeUpdateOnAttr :: Attribute CoreUpdate -> [Attribute]
   executeUpdateOnAttr a (CoreDelete _) = []
   executeUpdateOnAttr a (CoreRename (CAttr (name, value) _ _) str) = [(str, value)]
   executeUpdateOnAttr _ _ = abort "Unexpected argument for executeUpdateOnAttr"
```

The *children updates* are split into two parts. One part is the list of "insert into" updates, which are simple to perform by appending the new nodes to the end of the children list:

```
executeChildInsertIntos :: [Content] [CoreUpdate] -> [Content]
executeChildInsertIntos as [(CoreInsertInto newCont _):bs]
 //append the new content
 # newAS = as++newCont
 //go on with the next update
 = executeChildInsertIntos newAS bs
//if update list is empty then return the new children
executeChildInsertIntos as [] = as
```

The other part is the executing of the list of "ChildUpdateTuples". This is again done by executing tuple per tuple, but here you have to search for the corresponding child node in the list of children before. When this node is found, then the "delete" and "insert after", "insert before" updates were executed on this node, before continuing on the next tuple. The "delete" update simply removes the specified node from the children list, the "insert" updates add the new nodes before or after the specified node. The following code fragment shows the performing of the children updates:

```
executeChildUpdateTuples :: [Content] [ChildUpdateTuple] -> [Content]
executeChildUpdateTuples [a:as] [b=:(node, maybeDelete, inserts):bs]
 //if the first update effects the first child in the list then
 |identical a node
   //execute the delete
   # newA = executeDelete a maybeDelete
   //execute the inserts
   # newA = executeInserts newA inserts
   //execute the rest updates
   # newAS = executeChildUpdateTuples as bs
   //merge the results
   = newA++newAS
 //else
   = [a:executeChildUpdateTuples as [b:bs]]
 where
   //execute a delete
   executeDelete a (Just _) = []
   executeDelete a Nothing = [a]
   //execute a insert before
   executeInserts a [(CoreInsertBefore newCont _):bs]
     # newA = newCont++a
     = executeInserts newA bs
   //execute a insert after
   executeInserts a [(CoreInsertAfter newCont _):bs]
     # newA = a++newCont
     = executeInserts newA bs
   //if empty update list then return the new node
   executeInserts a [] = a
//If there are no more updates simply return the children.
executeChildUpdateTuples a [] = a
```

After all updates are executed, the original source of this document is updated. In my implementation this can be a simple file or an XML node in Tamino.

5.4 Experiences

The first part of the implementation was the parser written in Java and based on a JavaCC grammar. This task was straight forward, because I have some good experiences with programming in Java and with writing JavaCC grammars. An interesting experience in the course of the implementation was the programming of the executor in the CLEAN language. The functional programming with CLEAN was something totally new, because it follows a completely different paradigm. Not only in its functional side effect free paradigm, but it also has a rather mathematical notation.

At first glance it seemed to be very hard to read and especially to write CLEAN code. But after a short time you can write simple programs and even complex algorithms with a very few lines of code. A very good experience was also CLEAN's type system. This type system results in a sometimes very nit picking parser, but it turns out that programs that once passed the type checker are also in most cases semantically correct. No long and tedious debugging is needed.

The amount of potential programming errors is also reduced by the side effect free nature of the programming language. A function call will only produce a result for this call and will never have any influences to other parts or data of the program. Side effects are a frequent source for programming errors in imperative languages.

However, reading existing code of the Quip implementation was sometimes not very easy as a result of the CLEAN syntax. This syntax allows e.g. to write the same expression in a lot of different ways and allows to specify own higher order operators. This makes it very hard to read code written by other programmers, especially for beginners.

Overall it was an interesting experience to program in CLEAN and the language was well suited for rapid prototyping and to turn the given specification into a program.

6 Related Work

Although the topic of updating XML data has received little attention thus far, as the XML community has primarily focused on development of query language such as XQuery and their semantics, there are already some proposals or solutions for that. In this chapter I present some of them and try to compare them to my solution.

6.1 XML Tree Diff

XML Tree Diff from IBM Alphaworks [Alp99] is a tool to differentiate and update DOM trees. This tool exists since 1998 and is working on all Java Platforms. The differentiation tool compares two DOM trees and stores the result in an XML syntax. Actually there are two different syntaxes, one flat XML syntax, called FUL and one structered XML syntax for improved readability, called XUL.

These differentation results can be used to manipulate or update a DOM tree. So FUL and XUL can be called update languages. But these language were originally not designed to be readable or writable by humans. So you would have a hard job of writing update queries with this approach. But this solution is quite stable and has proved its usability for years. Consider the following FUL and XUL examples:

The following FUL example moves a subtree:

```
<move match="/*[1]/*[2]" tmatch="xfl16" parent="/*[1]" psib="/*[1]/*[5]"/>
```

Example 18: A FUL query

This XUL example also moves a subtree:

```
<node id="/">
     <node id="/*[1]">
          <node id="/*[1]/*[2]" op="move" to="xul_1"/>
          <node id="/*[1]/*[3]" />
      <node op="move" from="xul_1"/>
     </node>
</node>
```

Example 19: An XUL query

6.2 Lorel

Lorel [AQM+97] is a language designed for querying semistructured data. It was presented in April 1997 and it can be viewed as an extension of *OQL*. Lorel is implemented as the query language of the *Lore* prototype database managment system at Stanford University.

Lorel supports simple creation and deletion operations and modifications to the value of existing objects on the Lore data graph. Recent extensions to Lorel have migrated the query language to the XML data model, but the update features were not ported in the process.

For example, the following query adds the restaurant's city as a direct subobject of the restaurant object whenever the city is Palo Alto or Menlo Park:

```
update X.city += Z
from Guide.restaurant{x}.address.city Z
where Z = "Palo Alto" or Z = "Menlo Alto"
```

Example 20: Lorel update query

6.3 XUpdate/Lexus

XUpdate is an XML update language proposal from the XML:DB initiative. Lexus is a Java implementation of XUpdate. The actual working draft is from September 2000 and seems to be work in progress. XUpdate provides functionality for creation, insertion, deletion, value updating and renaming of XML nodes. It uses an XML syntax for the query and XPath expressions for the selection of the nodes to be updated.

The XML syntax makes it hard to write queries by humans. At the current version it supports no conditional updates. You find more about XUpdate in [LM00] [Mar00].

The following XUpdate update inserts a new address element after the first address:

```
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/xupdate">
     <xupdate:insert-after select="/addresses/address[1]" >
          <xupdate:element name="address">
               <xupdate:attribute name="id">2</xupdate:attribute>
               <fullname>Lars Martin</fullname>
               <born day='2' month='12' year='1974'/>
               <town>Leizig</town>
               <country>Germany</country>
          </xupdate:element>
     </xupdate:insert-after>
</xupdate:modifications>
```

Example 21: An XUpdate query

6.4 XPathLog/LoPiX

The XPathLog/LoPiX project is the continuation and migration of the F-Logic/Florid language to XML [May01]. XPathLog is a language that extends XPath with variable bindings to an XML querying and data manipulation language. LoPiX is an implementation of the XPathLog language. This project seems to be quite new because all documents are produced in 2001. It is developed by the Institute for Computer Science at the University of Freiburg.

Consider the following example, where "Bavaria" gets a (PCDATA) subelement name:

```
C/name[text()->"Bavaria" :- //country->C[car_code="BAV"]
```

Example 22: An XPathLog query

7 Conclusions and Future Work

This thesis presented a solution for updating XML documents. For this I specified requirements and usage scenarios in chapter 3, and there I decided to focus on well-formed documents and leave valid documents out of scope for this project. In chapter 4 I showed the actual XML update language, which is an extension to XQuery. This extension is based on a proposal of XQuery working group members, but modified to fit into the XQuery grammar. I also added a semiformal semantic specification for the update expressions.

Chapter 5 presented the design and implementation of the update processor, which is an extension to an existing XQuery processor developed by Sven Eric Panitz and me at Software AG. The architecture is split into two parts, the first one is written in Java and parses and transforms the query to an XML syntax (the XQueryX syntax). The second part is the executor, which is written in CLEAN, a pure functional programming language. The actual executing of the updates is done by evaluating the XQuery expression parts of the update and creating a sorted list of an simplified presentation of the updates. This list is then processed in order to perform the update on the document nodes.

The resulting processor is a complete implementation of the update proposal and all use cases can be handled. The processor can do updates on file system as well as on Tamino, which means they are processed via HTTP.

In Chapter 6 I presented then some other XML update proposals or solutions and showed the problems of these solutions in comparison to my proposal.

Caused by the limited time for this thesis some tempting topics were not treated here and have to be postponed to another project. The first thing here are valid documents. To handle these you also have to read the schema and check the update against it, i.e if the update will not violate this schema. Another thing are comments, processing-instructions and namespaces, which are not covered by the existing XQuery processor and so were also not covered by my update processor. But this seems to be a trivial task and the same algorithms can be used for that. It also would be interesting to make performance tests of this algorithm, but therefore a suitable test environment has to be set up.

I believe that my presented algorithms work not only in my special CLEAN environment, but can be easily translated into an other programming language, like C, C++ or Java. Certainly you have to make some modifications on the algorithms or you can remove some parts that are only needed for a functional programming language, but these parts are explained in this thesis and should therefore be trivial to perform.

Appendix A - Grammar

A.1 XQuery grammar

This is the extended XQuery grammar. The new update rules are the rules 62 to 69, modified is the rule 2.

```
[1]    QueryModuleList    ::=    QueryModule ( ";" QueryModule)*
[2]    QueryModule    ::=    ContextDecl* FunctionDefn* (ExprSequence | "update"
UpdateSequence)?
[3]    ContextDecl    ::=    ("namespace" NCName "=" StringLiteral)
   | ("default" "namespace" "=" StringLiteral)
   | ("schema" StringLiteral StringLiteral)
   | ("document" NCName "=" StringLiteral)
[4]    FunctionDefn    ::=    "define" "function" QName "(" ParamList? ")"
             ("returns" Datatype)? EnclosedExpr
[5]    ParamList    ::=    Param ("," Param)*
[6]    Param    ::=    Datatype? Variable
[7]    Expr    ::=    SortExpr
   | OrExpr
   | AndExpr
   | BeforeAfterExpr
   | FLWRExpr
   | IfExpr
   | SomeExpr
   | EveryExpr
   | TypeSwitchExpr
   | EqualityExpr
   | RelationalExpr
   | InstanceofExpr
   | RangeExpr
   | AdditiveExpr
   | MultiplicativeExpr
   | UnaryExpr
   | UnionExpr
   | IntersectExceptExpr
   | PathExpr
[8]    SortExpr    ::=    Expr "sortby" "(" SortSpecList ")"
[9]    SortSpecList    ::=    Expr ("ascending" | "descending")? ("," SortSpecList)?
[10]    OrExpr    ::=    Expr "or" Expr
[11]    AndExpr    ::=    Expr "and" Expr
[12]    BeforeAfterExpr    ::=    Expr ("before" | "after") Expr
[13]    FLWRExpr    ::=    (ForClause | LetClause)+ WhereClause? "return" Expr
[14]    ForClause    ::=    "for" Variable "in" Expr ("," Variable "in" Expr)*
[15]    LetClause    ::=    "let" Variable ":=" Expr ("," Variable ":=" Expr)*
[16]    WhereClause    ::=    "where" Expr
[17]    IfExpr    ::=    "if" "(" Expr ")" "then" Expr "else" Expr
[18]    SomeExpr    ::=    "some" Variable "in" Expr "satisfies" Expr
[19]    EveryExpr    ::=    "every" Variable "in" Expr "satisfies" Expr
[20]    TypeSwitchExpr    ::=    "typeswitch" "(" Expr ")" ("as" Variable)?
             CaseClause+ "default" "return" Expr
[21]    CaseClause    ::=    "case" Datatype "return" Expr
[22]    EqualityExpr    ::=    Expr ("=" | "!=" | "==" | "!==") Expr
[23]    RelationalExpr    ::=    Expr ("<" | "<=" | ">" | ">=") Expr
[24]    InstanceofExpr    ::=    Expr "instanceof" "only"? Datatype
[25]    RangeExpr    ::=    Expr "to" Expr
[26]    AdditiveExpr    ::=    Expr ("+" | "-") Expr
```

```
[27]    MultiplicativeExpr    ::=    Expr ("*" | "div" | "mod") Expr
[28]    UnaryExpr    ::=    ("-" | "+") Expr
[29]    UnionExpr    ::=    Expr ("union" | "|") Expr
[30]    IntersectExceptExpr    ::=    Expr ("intersect" | "except") Expr
[31]    PathExpr    ::=    RelativePathExpr
   | ("/" RelativePathExpr?)
   | ("//" RelativePathExpr?)
[32]    RelativePathExpr    ::=    StepExpr ( ("/" | "//") StepExpr)*
[33]    StepExpr    ::=    AxisStepExpr | OtherStepExpr
[34]    AxisStepExpr    ::=    Axis NodeTest StepQualifiers
[35]    OtherStepExpr    ::=    PrimaryExpr StepQualifiers
[36]    StepQualifiers    ::=    ( ("[" Expr "]") | ("=>" NameTest) )*
[37]    Axis    ::=    (NCName "::") | "@"
[38]    PrimaryExpr    ::=    "."
   | ".."
   | NodeTest
   | Variable
   | Literal
   | FunctionCall
   | ParenthesizedExpr
   | CastExpr
   | ElementConstructor
   | ElementConstructor2
   | AttributeConstructor2
[39]    Literal    ::=    NumericLiteral | StringLiteral
[40]    NodeTest    ::=    NameTest | KindTest
[41]    NameTest    ::=    QName | Wildcard
[42]    KindTest    ::=    PITest | CommentTest | TextTest | AnyKindTest
[43]    PITest    ::=    "processing-instruction" "(" StringLiteral? ")"
[44]    CommentTest    ::=    "comment" "(" ")"
[45]    TextTest    ::=    "text" "(" ")"
[46]    AnyKindTest    ::=    "node" "(" ")"
[47]    ParenthesizedExpr    ::=    "(" ExprSequence? ")"
[48]    ExprSequence    ::=    Expr ("," Expr)*
[49]    FunctionCall    ::=    QName "(" (Expr ("," Expr)*)? ")"
[50]    CastExpr    ::=    (("cast" "as") | ("treat" "as")) Datatype "(" Expr ")"
[51]    Datatype    ::=    QName
[52]    ElementConstructor    ::=    "<" NameSpec AttributeList ("/>" |
              (">" ElementContent* "</" (QName S?)? ">") )
[53]    NameSpec    ::=    QName | ( "{" Expr "}" )
[54]    AttributeList    ::=    (S (NameSpec S? "=" S? (AttributeValue
              | EnclosedExpr) AttributeList)? )?
[55]    AttributeValue    ::=    ( ["] AttributeValueContent* ["] )
   | ( ['] AttributeValueContent* ['] )
[56]    ElementContent    ::=    Char
   | ElementConstructor
   | EnclosedExpr
   | CdataSection
   | CharRef
   | PredefinedEntityRef
[57]    AttributeValueContent    ::=    Char
   | CharRef
   | EnclosedExpr
   | PredefinedEntityRef
[58]    CdataSection    ::=    "<![CDATA[" Char* "]]>"
[59]    EnclosedExpr    ::=    "{" ExprSequence "}"
[60] ComputedElementConstructor ::= "element" (QName | "{" Expr "}") "{" Expr "}"
[61] ComputedAttributeConstructor ::= "attribute" (QName | "{" Expr "}") "{" Expr "}"

[62] UpdateSequence    ::=    UpdateExpr (UpdateExpr)*
[63] UpdateExpr ::= Insert
     | Delete
     | Rename
```

```
      | Replace
      | FLWUExpr
      | IfUpdateExpr
      | "(" UpdateExpr ")"
[64] Insert ::= "insert" Expr ("following" | "preceding" | "into") Expr
[65] Delete ::= "delete" Expr
[66] Rename ::= "rename" Expr "as" Expr
[67] Replace ::= "replace" Expr "with" Expr
[68] FLWUExpr ::= (ForClause | LetClause)+ WhereClause? UpdateSequence
[69] IfUpdateExpr ::= "if" "(" Expr ")" "then" UpdateSequence ("else" UpdateSe-
quence)?
```

A.2 JavaCC grammar

The following code are the JavaCC rules for updates, which were modified from or added to the original JavaCC XQuery grammar provided by the XQuery working group.

```
void QueryModule():{}
{
  (ContextDecl())* (FunctionDefn())* [ExprSequence() | <Update> UpdateSequence()]
}

void UpdateSequence():{}
{
  Update() (Update())*
}

void Update():{}
{
      Insert() |
      Delete() |
      Rename() |
      Replace() |
      FlwuExpr() |
      IfUpdateExpr() |
      <Lpar> Update() <Rpar>
}

void Insert():{}
{
      <Insert> Expr() (<Following> | <Preceding> | <Into>) Expr()
}

void Delete():{}
{
      <Delete> Expr()
}

void Rename():{}
{
      <Rename> Expr() <As> Expr()
}

void Replace():{}
{
      <Replace> Expr() <With> Expr()
}

voidr FlwuExpr():{}
```

```
{
 ( ForClause()| LetClause() )+
   [WhereClause()]
   UpdateSequence()
}

void IfUpdateExpr():{}
{
     <IfLpar> Expr() <Rpar>  <Then> UpdateSequence() [ <Else> UpdateSequence() ]
}
```

A.3 XQueryX DTD

This is the extended XQueryX DTD. You find the added tags at the end of the DTD, the element query was modified.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % ORDER_LITERALS "(ASCENDING | DESCENDING)">
<!ENTITY % QUANTIFIER_TYPE "(SOME | EVERY)">
<!ENTITY % expression "(q:variable | q:constant | q:function | q:flwr
 | q:elementConstructor | q:predicatedExpr | q:sortBy | q:ifThenElseExpr
 | q:quantifier | q:exprList | q:step | q:identifier | q:nodeKindTest)">
<!ENTITY % AXIS_TYPE "(DEREFERENCE | ANCESTOR | ANCESTORORSELF
| ATTRIBUTE | CHILD | DESCENDANT | DESCENDANTORSELF | FOLLOWING
| FOLLOWINGSIBILING | NAMESPACE | PARENT | PRECEDING
| PRECEDINGSIBLING | SLASHSLASH | SELF)">
<!ENTITY % NODE_KIND "(NODE | TEXT | COMMENT | DATA | PROCESSING_INSTRUCTION)">
<!ENTITY % update_expression "(q:delete | q:insert | q:rename | q:replace |
     q:flwu | q:ifUpdateExpr | q:updateList)">
<!ENTITY % INSERT_KIND "(AFTER | BEFORE | INTO)">
<!ELEMENT q:query (q:document*, q:functionDefinition*, (%expression; |
%update_expression;))>
<!ELEMENT q:document (q:nameValuePair+)>
  <!ELEMENT q:nameValuePair EMPTY>
    <!ATTLIST q:nameValuePair
      name CDATA #REQUIRED
      value CDATA #REQUIRED
    >
<!ELEMENT q:functionDefinition (q:argumentDeclaration*, %expression;)>
<!ATTLIST q:functionDefinition
     functionName CDATA #REQUIRED
     datatype CDATA #REQUIRED
>
<!ELEMENT q:argumentDeclaration EMPTY>
<!ATTLIST q:argumentDeclaration
     name CDATA #REQUIRED
     datatype CDATA #REQUIRED
>
<!ELEMENT q:exprList (%expression;)*>
<!ELEMENT q:predicatedExpr (%expression;, q:predicate+)>
<!ELEMENT q:predicate ((q:rangeFrom, q:rangeTo) | %expression;)>
<!ELEMENT q:rangeFrom (%expression;)>
<!ELEMENT q:rangeTo (%expression;)>
<!ELEMENT q:variable (#PCDATA)>
<!ELEMENT q:identifier (#PCDATA)>
<!ELEMENT q:constant (#PCDATA)>
<!ATTLIST q:constant
     datatype CDATA #IMPLIED
```

```
>
<!ELEMENT q:function (%expression;)*>
<!ATTLIST q:function
     name CDATA #REQUIRED
>
<!ELEMENT q:flwr ((q:forAssignment | q:letAssignment)+, q:where?, q:return)>
<!ELEMENT q:forAssignment %expression;>
<!ATTLIST q:forAssignment
     variable CDATA #REQUIRED
>
<!ELEMENT q:letAssignment %expression;>
<!ATTLIST q:letAssignment
     variable CDATA #REQUIRED
>
<!ELEMENT q:where (%expression;)>
<!ELEMENT q:return (%expression;)>
<!ELEMENT q:ifThenElseExpr (%expression;, %expression;, %expression;)>
<!ELEMENT q:sortBy (%expression;, q:sortfield+)>
<!ELEMENT q:sortfield (%expression;)>
<!ATTLIST q:sortfield
     order %ORDER_LITERALS; "ASCENDING"
>
<!ELEMENT q:quantifier (q:quantifierAssignment, %expression;)>
<!ATTLIST q:quantifier
     type %QUANTIFIER_TYPE; "SOME"
>
<!ELEMENT q:quantifierAssignment (%expression;)>
<!ATTLIST q:quantifierAssignment
     variable CDATA #REQUIRED
>
<!ELEMENT q:elementConstructor (q:tagName, q:attributeConstructor*, (%expres-
sion;)*)>
<!ELEMENT q:tagName (q:identifier | q:variable)>
<!ELEMENT q:attributeConstructor (q:attributeName, q:attributeValue)>
<!ELEMENT q:attributeName (q:identifier | q:variable)>
<!ELEMENT q:attributeValue (%expression;)>
<!ELEMENT q:step (%expression;, %expression;)>
<!ATTLIST q:step
     axis %AXIS_TYPE; #REQUIRED
     abbreviated (true | false) "true"
>
<!ELEMENT q:dot EMPTY>
<!ELEMENT q:dotdot EMPTY>
<!ELEMENT q:nodeKindTest (q:piTargetTest?)>
<!ATTLIST q:nodeKindTest
     kind %NODE_KIND; #REQUIRED
>
<!ELEMENT q:piTargetTest (#PCDATA)>

<!ELEMENT q:updateList (%update_expression;)*>
<!ELEMENT q:delete (%expression;)>
<!ELEMENT q:insert (%expression;, %expression;)>
<!ATTLIST q:insert
     target %INSERT_KIND; #REQUIRED
>
<!ELEMENT q:rename (%expression;, %expression;)>
<!ELEMENT q:replace (%expression;, %expression;)>
<!ELEMENT q:flwu ((q:forAssignment | q:letAssignment)+, q:where?,
%update_expression;)>
<!ELEMENT q:ifUpdateExpr (%expression;, %update_expression;,
(%update_expression;)?)>
```

Appendix B - Use Cases

The use cases in this chapter were produced to get a feeling for XML updates and as a testbed for the update processor. They are based on the data of the XQuery use cases [CFMR01b].

B.1 Use Case "XMP": Experiences and Exemplars

B.1.1 Document Type Definitions (DTD)

Most of the example queries in this use case are based on a bibliography document named "bib.xml" with the following DTD:

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

B.1.2 Sample Data

```
<bib>
     <book year="1994">
          <title>TCP/IP Illustrated</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price> 65.95</price>
     </book>
     <book year="1992">
          <title>Advanced Programming in the Unix environment</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price>65.95</price>
     </book>
     <book year="2000">
          <title>Data on the Web</title>
          <author><last>Abiteboul</last><first>Serge</first></author>
          <author><last>Buneman</last><first>Peter</first></author>
          <author><last>Suciu</last><first>Dan</first></author>
          <publisher>Morgan Kaufmann Publishers</publisher>
          <price> 39.95</price>
     </book>
     <book year="1999">
          <title>The Economics of Technology and Content for Digital TV</title>
          <editor>
               <last>Gerbarg</last><first>Darcy</first>
               <affiliation>CITI</affiliation>
          </editor>
          <publisher>Kluwer Academic Publishers</publisher>
```

```
          <price>129.95</price>
     </book>
</bib>
```

B.1.3 DTD for Q12 and Q13

Q12 and Q13 also use information on book reviews and prices from a separate data source named "reviews.xml" with the following DTD:

```
<!ELEMENT reviews (entry*)>
<!ELEMENT entry (title, price, review)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT review (#PCDATA)>
```

B.1.4 Sample Data for Q12 and Q13

```
<reviews>
     <entry>
          <title>Data on the Web</title>
          <price>34.95</price>
          <review>
               A very good discussion of semi-structured database
               systems and XML.
        </review>
     </entry>
     <entry>
          <title>Advanced Programming in the Unix environment</title>
          <price>65.95</price>
          <review>
               A clear and detailed discussion of UNIX programming.
        </review>
     </entry>
     <entry>
          <title>TCP/IP Illustrated</title>
          <price>65.95</price>
          <review>
               One of the best books on TCP/IP.
        </review>
     </entry>
</reviews>
```

B.1.5 Queries and Results

Q1: Adding a new book to the document.

Solution in XQuery:

```
UPDATE
INSERT
     <book year="1999">
          <title>Java in a Nutshell</title>
          <author><last>Flanagan</last><first>David</first></author>
          <publisher>O'Reilly</publisher>
          <price>29.95</price>
     </book>
INTO document("data/xmp-data.xml")/bib
```

Expected results:

```
<bib>
     ...
     <book year="1999">
          <title>The Economics of Technology and Content for Digital TV</title>
          <editor>
               <last>Gerbarg</last><first>Darcy</first>
               <affiliation>CITI</affiliation>
          </editor>
          <publisher>Kluwer Academic Publishers</publisher>
          <price>129.95</price>
     </book>
     <book year="1999">
          <title>Java in a Nutshell</title>
          <author><last>Flanagan</last><first>David</first></author>
          <publisher>O'Reilly</publisher>
          <price>29.95</price>
     </book>
</bib>
```

Q2: Deleting a book with a specific title.

Solution in XQuery:

```
UPDATE
DELETE document("data/xmp-data.xml")/bib/book[title="TCP/IP Illustrated"]
```

Alternative solution in XQuery:

```
UPDATE
FOR $a IN document("data/xmp-data.xml")/bib/book
WHERE $a/title = "TCP/IP Illustrated"
DELETE $a
```

Expected results:

```
<bib>
     <book year="1992">
          <title>Advanced Programming in the Unix environment</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price>65.95</price>
     </book>
     <book year="2000">
          <title>Data on the Web</title>
          <author><last>Abiteboul</last><first>Serge</first></author>
          <author><last>Buneman</last><first>Peter</first></author>
          <author><last>Suciu</last><first>Dan</first></author>
          <publisher>Morgan Kaufmann Publishers</publisher>
          <price> 39.95</price>
     </book>
     <book year="1999">
          <title>The Economics of Technology and Content for Digital TV</title>
          <editor>
               <last>Gerbarg</last><first>Darcy</first>
               <affiliation>CITI</affiliation>
          </editor>
          <publisher>Kluwer Academic Publishers</publisher>
          <price>129.95</price>
     </book>
</bib>
```

Q3: Updating the value of the attribute year for a book with specific title.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/xmp-data.xml")/bib/book
WHERE $a/title = "TCP/IP Illustrated"
REPLACE $a/@year WITH ATTRIBUTE year {"1996"}
```

Expected results:

```
<book year="1996">
     <title>TCP/IP Illustrated</title>
     <author><last>Stevens</last><first>W.</first></author>
     <publisher>Addison-Wesley</publisher>
     <price>65.95</price>
</book>
```

Q4: Updating the value of the element publisher for a book with specific title.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/xmp-data.xml")/bib/book
WHERE $a/title = "TCP/IP Illustrated"
REPLACE $a/publisher/text() WITH "foo"
```

Expected results:

```
<book year="1994">
     <title>TCP/IP Illustrated</title>
     <author><last>Stevens</last><first>W.</first></author>
     <publisher>foo</publisher>
     <price>65.95</price>
</book>
```

Q5: Updating the value of all price elements relatively to the old price.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/xmp-data.xml")/bib/book/price/text()
REPLACE $a WITH $a * 1.05
```

Expected results:

```
<bib>
     <book year="1994">
          <title>TCP/IP Illustrated</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price>69.25</price> </book>
     <book year="1992">
          <title>Advanced Programming in the Unix environment</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price>69.25</price>
     </book>
     ...
</bib>
```

Q6: Rename the element author to writer.

Solution in XQuery:

```
UPDATE
RENAME document("data/xmp-data.xml")/bib/book/author AS "writer"
```

Expected results:

```
<bib>
     <book year="1994">
          <title>TCP/IP Illustrated</title>
          <writer><last>Stevens</last><first>W.</first></writer>
          <publisher>Addison-Wesley</publisher>
          <price> 65.95</price>
     </book>
     ...
</bib>
```

Q7: Rename the attribute year to published.

Solution in XQuery:

```
UPDATE
RENAME document("data/xmp-data.xml")/bib/book/@year AS "published"
```

Expected results:

```
<bib>
     <book published="1994">
          <title>TCP/IP Illustrated</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price> 65.95</price>
     </book>
     ...
</bib>
```

Q8: Add a new instances element before the price element.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/xmp-data.xml")/bib/book/price
INSERT
     <instances>0</instances>
PRECEDING $a
```

Expected results:

```
<bib>
     <book year="1994">
          <title>TCP/IP Illustrated</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <instances>0</instances>
          <price> 65.95</price>
     </book>
     ...
</bib>
```

Q9: Add a new instances attribute into all book elements.

Solution in XQuery:

```
UPDATE
```

```
FOR $a IN document("data/xmp-data.xml")/bib/book
INSERT ATTRIBUTE instances { "0" }
INTO $a
```

Expected results:

```
<bib>
     <book year="1994" instances="0">
          <title>TCP/IP Illustrated</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price>65.95</price>
     </book>
     ...
</bib>
```

Q10: Deepen structure by adding the first and last element from author into an additional name tag.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/xmp-data.xml")/bib/book/author
LET $as := $a/*
DELETE $as
INSERT
     <name>
          { $as }
     </name>
INTO $a
```

Expected results:

```
<bib>
     <book year="1994">
          <title>TCP/IP Illustrated</title>
          <author>
               <name><last>Stevens</last><first>W.</first></name>
          </author>
          <publisher>Addison-Wesley</publisher>
          <price> 65.95</price>
     </book>
     ...
</bib>
```

Q11: Flatten structure by removing the name tag and adding the last and first elements directly to the author. Based on the results of Q10.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/xmp-data.xml")/bib/book/author
LET $b := $a/name
LET $as := $b/*
DELETE $b
INSERT $as
INTO $a
```

Expected results:

The original document as you see in B.1.2.

Q12: Adds the reviews to the books.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/xmp-data.xml")/bib/book
FOR $b IN document("data/Q5xmp-data.xml")/reviews/entry
WHERE $a/title = $b/title
INSERT
     $b/review
INTO $a
```

Expected results:

```
<bib>
     <book year="1994">
          <title>TCP/IP Illustrated</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price> 65.95</price>
          <review>
               One of the best books on TCP/IP.
          </review>
     </book>
     ...
</bib>
```

Q13: Remove all books that have no review.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/xmp-data.xml")/bib/book
LET $b := document("data/Q5xmp-data.xml")/reviews/entry[title=$a/title]
IF (empty($b)) THEN
     DELETE $a
```

Expected results:

```
<bib>
     <book year="1994">
          <title>TCP/IP Illustrated</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price> 65.95</price>
     </book>
     <book year="1992">
          <title>Advanced Programming in the Unix environment</title>
          <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
          <price>65.95</price>
     </book>
     <book year="2000">
          <title>Data on the Web</title>
          <author><last>Abiteboul</last><first>Serge</first></author>
          <author><last>Buneman</last><first>Peter</first></author>
          <author><last>Suciu</last><first>Dan</first></author>
          <publisher>Morgan Kaufmann Publishers</publisher>
          <price> 39.95</price>
     </book>
</bib>
```

B.2 Use Case "TREE": Queries that preserve hierarchy

B.2.1 Document Type Definition (DTD)

```
<!ELEMENT book (title, author+, section+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT section (title, (p | figure | section)* )>
<!ATTLIST section
     id         ID    #IMPLIED
     difficulty CDATA #IMPLIED>
<!ELEMENT p (#PCDATA)>
<!ELEMENT figure (title, image)>
<!ATTLIST figure
     width   CDATA   #REQUIRED
     height  CDATA   #REQUIRED >
<!ELEMENT image EMPTY>
<!ATTLIST image
     source  CDATA   #REQUIRED >
```

B.2.2 Sample Data

```
<book>
  <title>Data on the Web</title>
  <author>Serge Abiteboul</author>
  <author>Peter Buneman</author>
  <author>Dan Suciu</author>
  <section id="intro" difficulty="easy" >
    <title>Introduction</title>
    <p>Text ... </p>
    <section>
      <title>Audience</title>
      <p>Text ... </p>
    </section>
    <section>
      <title>Web Data and the Two Cultures</title>
      <p>Text ... </p>
      <figure height="400" width="400">
        <title>Traditional client/server architecture</title>
        <image source="csarch.gif"/>
      </figure>
      <p>Text ... </p>
    </section>
  </section>
  <section id="syntax" difficulty="medium" >
    <title>A Syntax For Data</title>
    <p>Text ... </p>
    <figure height="200" width="500">
      <title>Graph representations of structures</title>
      <image source="graphs.gif"/>
    </figure>
    <p>Text ... </p>
    <section>
      <title>Base Types</title>
      <p>Text ... </p>
    </section>
    <section>
      <title>Representing Relational Databases</title>
      <p>Text ... </p>
      <figure height="250" width="400">
```

```
        <title>Examples of Relations</title>
        <image source="relations.gif"/>
      </figure>
    </section>
    <section>
      <title>Representing Object Databases</title>
      <p>Text ... </p>
    </section>
  </section>
</book>
```

B.2.3 Queries and Results:

Q1: Inserting a new section before the "Representing Relational Databases" section.

Solution in XQuery:

```
UPDATE
INSERT
     <section>
          <title>Representing XML Databases</title>
          <p>Text ...</p>
     </section>
PRECEDING document("data/tree-data.xml")/book/section[2]/section[title="Represen-
ting Relational Databases"]
```

Expected results:

```
<book>
     ...
     <section>
          <title>Representing XML Databases</title>
          <p>Text ...</p>
     </section>
     <section>
          <title>Representing Relational Databases</title>
          <p>Text ... </p>
          <figure height="250" width="400"> <
               title>Examples of Relations</title>
               <image source="relations.gif"/>
          </figure>
     </section>
     ...
</book>
```

Q2: Moving a section to an other place.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/tree-data.xml")/book/section[2]/section
WHERE $a/title = "Representing Object Databases"
DELETE $a
INSERT $a
PRECEDING document("data/tree-data.xml")/book/section[2]/section[title="Represen-
ting Relational Databases"]
```

Expected results:

```
<book>
     ...
     <section>
```

```
          <title>Representing Object Databases</title>
          <p>Text ... </p>
     </section>
     <section>
          <title>Representing Relational Databases</title>
          <p>Text ... </p>
          <figure height="250" width="400">
               <title>Examples of Relations</title>
               <image source="relations.gif"/>
          </figure>
     </section>
</section>
</book>
```

Q3: Creating a new top level section out of two other sections.

Solution in XQuery:

```
UPDATE
LET $as := document("data/tree-data.xml")/book/section[2]/section[2 TO 3]
DELETE $as
INSERT
     <section id="databases" difficulty="medium">
          { $as }
     </section>
INTO document("data/tree-data.xml")/book
```

Expected results:

```
<section id="databases" difficulty="medium">
     <section>
          <title>Representing Relational Databases</title>
          <p>Text ... </p>
          <figure height="250" width="400">
               <title>Examples of Relations</title>
               <image source="relations.gif"/>
          </figure>
     </section>
     <section>
          <title>Representing Object Databases</title>
          <p>Text ... </p>
     </section>
</section>
```

Q4: Update the path of the source attributes.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/tree-data.xml")/book//image/@source
REPLACE $a WITH ATTRIBUTE source {("images/", $a)}
```

Expected results:

```
<figure height="400" width="400">
     <title>Traditional client/server architecture</title>
     <image source="images/csarch.gif"/>
</figure>
```

Q5: Promote the structuring of two sections.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/tree-data.xml")/book/section[2]/section
WHERE contains($a/title, "Representing")
DELETE $a
INSERT $a
INTO $a/../..
```

Expected results:

```
...
          <section>
               <title>Base Types</title>
               <p>Text ... </p>
          </section>
     </section>
     <section>
          <title>Representing Relational Databases</title>
          <p>Text ... </p>
          <figure height="250" width="400">
               <title>Examples of Relations</title>
               <image source="relations.gif"/>
          </figure>
     </section>
     <section>
          <title>Representing Object Databases</title>
          <p>Text ... </p>
     </section>
</book>
```

Q6: Demotes the structering of two sections. Based on the result of Q5.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/tree-data.xml")/book/section
WHERE contains($a/title, "Representing")
DELETE $a
INSERT $a
INTO document("data/tree-data.xml")/book/section[2]
```

Expected results:
The original document as you can see it in B.2.2.

Q7: Removes all title elements and adds this information into an title attribute of their parents.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/tree-data.xml")/book//title
DELETE $a
INSERT ATTRIBUTE title { $a/text() }
INTO $a/..
```

Expected results:

```
<book title="Data on the Web">
     <author>Serge Abiteboul</author>
     <author>Peter Buneman</author>
     <author>Dan Suciu</author>
     <section id="intro" difficulty="easy" title="Introduction">
```

```
          <p>Text ... </p>
          <section title="Audience">
               <p>Text ... </p>
          </section>
          <section title="Web Data and the Two Cultures">
               <p>Text ... </p>
               <figure height="400" width="400" title="Traditional client/server
architecture">
                    <image source="csarch.gif"/>
               </figure>
               <p>Text ... </p>
          </section>
     </section>
     ...
</book>
```

Q8: Remove the p elements and add their content directly to the parent element.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/tree-data.xml")/book//p
REPLACE $a WITH $a/text()
```

Expected results:

```
<section id="intro" difficulty="easy" >
     <title>Introduction</title>
     Text ...
     <section>
          <title>Audience</title>
          Text ...
     </section>
     <section>
          <title>Web Data and the Two Cultures</title>
          Text ...
          <figure height="400" width="400">
               <title>Traditional client/server architecture</title>
               <image source="csarch.gif"/>
          </figure>
          Text ...
     </section>
</section>
```

Q9: Insert text into a mixed content element. Based upon Q8.

Solution in XQuery:

```
UPDATE
INSERT "A very important part of text."
PRECEDING document("data/tree-data.xml")/book/section[2]/text()[1]
```

Expected results:

```
<section id="syntax" difficulty="medium" >
     <title>A Syntax For Data</title>
     A very important part of text. Text ...
     <figure height="200" width="500">
          <title>Graph representations of structures</title>
          <image source="graphs.gif"/>
     </figure>
     Text ...
     ...
```

```
</section>
```

Q10: Creates a new document for the book DTD and adds the root element "book".

Solution in XQuery:

```
DOCUMENT   name="newBook.xml"
           uri="http://www.example.com/books"
           create="first" # one of "first", "always"
           overwrite="no"

UPDATE
LET $l := document("http://www.example.com/books/newBook.xml")
IF (empty($l/book))
      INSERT <book/> INTO $l
```

Q11: Deletes a document.

Solution in XQuery:

```
UPDATE
DELETE document("http://www.example.com/books/newBook.xml")
```

B.3 Use Case "R" - Access to Relational Data

This use case is based on three separate input documents named users.xml, items.xml, and bids.xml. Each of the documents represents one of the tables in the relational database, using the following DTDs:

B.3.1 Document Type Definition (DTD)

```
<!DOCTYPE users [
  <!ELEMENT users   (user_tuple*)>
  <!ELEMENT user_tuple (userid, name, rating?)>
  <!ELEMENT userid  (#PCDATA)>
  <!ELEMENT name    (#PCDATA)>
  <!ELEMENT rating  (#PCDATA)>
]>
<!DOCTYPE items [
  <!ELEMENT items       (item_tuple*)>
  <!ELEMENT item_tuple  (itemno, description, offered_by, start_date?, end_date?,
reserve_price? )>
  <!ELEMENT itemno      (#PCDATA)>
  <!ELEMENT description (#PCDATA)>
  <!ELEMENT offered_by  (#PCDATA)>
  <!ELEMENT start_date  (#PCDATA)>
  <!ELEMENT end_date    (#PCDATA)>
  <!ELEMENT reserve_price (#PCDATA)>
]>
<!DOCTYPE bids [
  <!ELEMENT bids      (bid_tuple*)>
  <!ELEMENT bid_tuple (userid, itemno, bid, bid_date)>
  <!ELEMENT userid    (#PCDATA)>
  <!ELEMENT itemno    (#PCDATA)>
  <!ELEMENT bid       (#PCDATA)>
  <!ELEMENT bid_date  (#PCDATA)>
]>
```

B.3.2 Sample Data

USERS

USERID	NAME	RATING
U01	Tom Jones	B
U02	Mary Doe	A
U03	Dee Linquent	D
U04	Roger Smith	C
U05	Jack Sprat	B
U06	Rip Van Winkle	B

ITEMS

ITEMNO	DESCRIPTION	OFFERED_BY	START_DATE	END_DATE	RESERVE_PRICE
1001	Red Bicycle	U01	99-01-05	99-01-20	40
1002	Motorcycle	U02	99-02-11	99-03-15	500
1003	Old Bicycle	U02	99-01-10	99-02-20	25
1004	Tricycle	U01	99-02-25	99-03-08	15
1005	Tennis Racket	U03	99-03-19	99-04-30	20
1006	Helicopter	U03	99-05-05	99-05-25	50000
1007	Racing Bicycle	U04	99-01-20	99-02-20	200
1008	Broken Bicycle	U01	99-02-05	99-03-06	25

BIDS

USERID	ITEMNO	BID	BID_DATE
U02	1001	35	99-01-07
U04	1001	40	99-01-08
U02	1001	45	99-01-11
U04	1001	50	99-01-13
U02	1001	55	99-01-15
U01	1002	400	99-02-14
U02	1002	600	99-02-16
U03	1002	800	99-02-17
U04	1002	1000	99-02-25
U02	1002	1200	99-03-02
U04	1003	15	99-01-22
U05	1003	20	99-02-03
U01	1004	40	99-03-05
U03	1007	175	99-01-25
U05	1007	200	99-02-08
U04	1007	225	99-02-12

B.3.3 Queries and Results

Q1: Remove all bicycles from the bids.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/R-bids.xml")/bids/bid_tuple
FOR $b IN document("data/R-items.xml")/items/item_tuple
WHERE contains($b/description, "Bicycle") AND $a/itemno = $b/itemno
DELETE $a
```

Expected results:

BIDS

USERID	ITEMNO	BID	BID_DATE
U01	1002	400	99-02-14
U02	1002	600	99-02-16
U03	1002	800	99-02-17
U04	1002	1000	99-02-25
U02	1002	1200	99-03-02
U01	1004	40	99-03-05

Q2: Remove all bids from users that have rating C or D.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/R-bids.xml")/bids/bid_tuple
FOR $b IN document("data/R-users.xml")/users/user_tuple
WHERE ($b/rating = "C" OR $b/rating = "D") AND $a/userid = $b/userid
DELETE $a
```

Expected results:

BIDS

USERID	ITEMNO	BID	BID_DATE
U02	1001	35	99-01-07
U02	1001	45	99-01-11
U02	1001	55	99-01-15
U01	1002	400	99-02-14
U02	1002	600	99-02-16
U02	1002	1200	99-03-02
U05	1003	20	99-02-03
U01	1004	40	99-03-05
U05	1007	200	99-02-08

Q3: Deletes all bids for items that have an end_date lying in the past if the number of tuples in bids is higher than 500.

Solution in XQuery:

```
UPDATE
LET $bs := document("data/R-bids.xml")/bids/bid_tuple
IF (count($bs) > 500) THEN
     FOR $a IN $bs
     FOR $b IN document("data/R-items.xml")/items/item_tuple
     WHERE ($a/itemno = $b/itemno) and ($b/end_date < today())
     DELETE $a
```

Expected results:
No item would be deleted because the number of bid_tuple is less than 500.

Q4: Inserts a new bid for itemno 1001 at the correct place.

Solution in XQuery:

```
UPDATE
LET $as := document("data/R-bids.xml")/bids/bid_tuple[itemno = 1001]
FOR $a in $as
WHERE $a/bid = max($as/bid)
INSERT
     <bid_tuple>
          <userid>U04</userid>
          <itemno>1001</itemno>
          <bid>60</bid>
          <bid_date>99-01-16</bid_date>
     </bid_tuple>
FOLLOWING $a
```

Expected results:

BIDS

USERID	ITEMNO	BID	BID_DATE
U02	1001	35	99-01-07
U04	1001	40	99-01-08
U02	1001	45	99-01-11
U04	1001	50	99-01-13
U02	1001	55	99-01-15
U04	1001	60	99-01-16
U01	1002	400	99-02-14
---	---	---	---

Q5: Inserts a first bid for the helicopter.

Solution in XQuery:

```
UPDATE
LET $a := document("data/R-bids.xml")/bids/bid_tuple[itemno = 1007]
INSERT
     <bid_tuple>
```

```
          <userid>U01</userid>
          <itemno>1006</itemno>
          <bid>52000</bid>
          <bid_date>99-01-16</bid_date>
     </bid_tuple>
PRECEDING first($a)
```

Expected results:

BIDS

USERID	ITEMNO	BID	BID_DATE
U01	1004	40	99-03-05
U01	1006	52000	99-01-16
U03	1007	175	99-01-25
U05	1007	200	99-02-08
U04	1007	225	99-02-12

Q6: Renames all itemno elements to itemid elements.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/R-bids.xml")/bids/bid_tuple/itemno
     UNION document("data/R-items.xml")/items/item_tuple/itemno
RENAME $a
AS "itemid"
```

Expected results:

ITEMS

ITEMID	DESCRIPTION	OFFERED_BY	START_DATE	END_DATE	RESERVE_PRICE

BIDS

USERID	ITEMID	BID	BID_DATE

Q7: Replaces the bid of one user for one item to 110% of the actual maximum bid for that item, or to 300 when the maximum bid is over 300.

Solution in XQuery:

```
UPDATE
FOR $a IN document("data/R-bids.xml")/bids/bid_tuple
LET $maxBid := max(document("data/R-bids.xml")/bids/bid_tuple[itemno="1007"]/bid)
WHERE $a/userid="U03" AND $a/itemno="1007"
REPLACE $a/bid
WITH <bid>
     {
               IF ($maxBid * 1.1 < 300) THEN
                    $maxBid * 1.1
```

```
            ELSE
                  300
}
</bid>
```

Expected results:

BIDS

---	---	---	---
U03	1007	248	99-02-13
---	---	---	---

Appendix C - Examples of chapter 4 in XQueryX syntax

C.1 Example 7

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:insert target="BEFORE">
    <q:elementConstructor>
      <q:tagName>
     <q:constant datatype="CHARSTRING">warning</q:constant>
      </q:tagName>
      <q:constant datatype="CHARSTRING">High Blood Pressure!</q:constant>
    </q:elementConstructor>
    <q:step axis="DESCENDANT">
      <q:predicatedExpr>
     <q:identifier>blood_pressure</q:identifier>
     <q:predicate>
       <q:function name="GT">
         <q:identifier>systolic</q:identifier>
         <q:constant datatype="INTEGER">180</q:constant>
       </q:function>
     </q:predicate>
      </q:predicatedExpr>
    </q:step>
  </q:insert>
</q:query>
```

C.2 Example 8

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:insert target="AFTER">
    <q:elementConstructor>
      <q:tagName>
     <q:constant datatype="CHARSTRING">warning</q:constant>
      </q:tagName>
      <q:constant datatype="CHARSTRING">High Blood Pressure!</q:constant>
    </q:elementConstructor>
    <q:step axis="DESCENDANT">
      <q:predicatedExpr>
     <q:identifier>blood_pressure</q:identifier>
     <q:predicate>
       <q:function name="GT">
         <q:identifier>systolic</q:identifier>
         <q:constant datatype="INTEGER">180</q:constant>
       </q:function>
     </q:predicate>
      </q:predicatedExpr>
    </q:step>
  </q:insert>
</q:query>
```

C.3 Example 9

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:insert target="INTO">
    <q:attributeConstructor>
      <q:attributeName>
     <q:identifier>warning</q:identifier>
      </q:attributeName>
      <q:attributeValue>
     <q:constant datatype="CHARSTRING">High Blood Pressure!</q:constant>
      </q:attributeValue>
    </q:attributeConstructor>
    <q:step axis="DESCENDANT">
      <q:predicatedExpr>
     <q:identifier>blood_pressure</q:identifier>
     <q:predicate>
       <q:function name="GT">
         <q:identifier>systolic</q:identifier>
         <q:constant datatype="INTEGER">180</q:constant>
       </q:function>
     </q:predicate>
      </q:predicatedExpr>
    </q:step>
  </q:insert>
</q:query>
```

C.4 Example 10

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:insert target="INTO">
    <q:elementConstructor>
      <q:tagName>
     <q:constant datatype="CHARSTRING">warning</q:constant>
      </q:tagName>
      <q:constant datatype="CHARSTRING">High Blood Pressure!</q:constant>
    </q:elementConstructor>
    <q:step axis="DESCENDANT">
      <q:predicatedExpr>
     <q:identifier>blood_pressure</q:identifier>
     <q:predicate>
       <q:function name="GT">
         <q:identifier>systolic</q:identifier>
         <q:constant datatype="INTEGER">180</q:constant>
       </q:function>
     </q:predicate>
      </q:predicatedExpr>
    </q:step>
  </q:insert>
</q:query>
```

C.5 Example 11

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:delete>
    <q:step axis="DESCENDANT">
```

```
      <q:predicatedExpr>
     <q:identifier>blood_pressure</q:identifier>
     <q:predicate>
       <q:function name="GT">
         <q:identifier>systolic</q:identifier>
         <q:constant datatype="INTEGER">180</q:constant>
       </q:function>
     </q:predicate>
      </q:predicatedExpr>
    </q:step>
  </q:delete>
</q:query>
```

C.6 Example 12

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:replace>
    <q:function name="EQUALS">
      <q:step axis="DESCENDANT">
     <q:identifier>job</q:identifier>
      </q:step>
      <q:constant datatype="CHARSTRING">bit banger</q:constant>
    </q:function>
    <q:elementConstructor>
      <q:tagName>
     <q:constant datatype="CHARSTRING">profession</q:constant>
      </q:tagName>
      <q:constant datatype="CHARSTRING">Computer Scientist</q:constant>
    </q:elementConstructor>
  </q:replace>
</q:query>
```

C.7 Example 13

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:rename>
    <q:function name="EQUALS">
      <q:step axis="DESCENDANT">
     <q:identifier>job</q:identifier>
      </q:step>
      <q:constant datatype="CHARSTRING">bit banger</q:constant>
    </q:function>
    <q:constant datatype="CHARSTRING">profession</q:constant>
  </q:rename>
</q:query>
```

C.8 Example 14

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:ifUpdateExpr>
    <q:function name="GT">
      <q:function name="count">
     <q:step axis="DESCENDANT">
       <q:identifier>employee</q:identifier>
     </q:step>
```

```
      </q:function>
      <q:constant datatype="INTEGER">1000</q:constant>
    </q:function>
    <q:ifUpdateExpr>
      <q:function name="count">
     <q:function name="GT">
       <q:step axis="DESCENDANT">
         <q:predicatedExpr>
           <q:identifier>employee</q:identifier>
           <q:predicate>
          <q:function name="EQUALS">
            <q:identifier>sex</q:identifier>
            <q:identifier>male</q:identifier>
          </q:function>
           </q:predicate>
         </q:predicatedExpr>
       </q:step>
       <q:constant datatype="INTEGER">100</q:constant>
     </q:function>
      </q:function>
      <q:delete>
     <q:step axis="DESCENDANT">
       <q:predicatedExpr>
         <q:identifier>employee</q:identifier>
         <q:predicate>
           <q:function name="EQUALS">
          <q:identifier>sex</q:identifier>
          <q:identifier>male</q:identifier>
           </q:function>
         </q:predicate>
       </q:predicatedExpr>
     </q:step>
      </q:delete>
    </q:ifUpdateExpr>
  </q:ifUpdateExpr>
</q:query>
```

C.9 Example 15

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:flwu>
    <q:forAssignment variable="$i">
      <q:step axis="DESCENDANT">
     <q:identifier>invoice</q:identifier>
      </q:step>
    </q:forAssignment>
    <q:letAssignment variable="$s">
      <q:step axis="DESCENDANT">
     <q:variable>$i</q:variable>
     <q:predicatedExpr>
       <q:identifier>product</q:identifier>
       <q:predicate>
         <q:function name="EQUALS">
           <q:identifier>name</q:identifier>
           <q:constant datatype="CHARSTRING">screwdriver</q:constant>
         </q:function>
       </q:predicate>
     </q:predicatedExpr>
      </q:step>
    </q:letAssignment>
    <q:where>
```

```
      <q:function name="not">
     <q:function name="empty">
       <q:variable>$s</q:variable>
     </q:function>
      </q:function>
    </q:where>
    <q:updateList>
      <q:insert target="BEFORE">
     <q:elementConstructor>
       <q:tagName>
         <q:constant datatype="CHARSTRING">product</q:constant>
       </q:tagName>
      <q:elementConstructor>
         <q:tagName>
           <q:constant datatype="CHARSTRING">name</q:constant>
         </q:tagName>
         <q:constant datatype="CHARSTRING">Hammer</q:constant>
      </q:elementConstructor>
      <q:elementConstructor>
         <q:tagName>
           <q:constant datatype="CHARSTRING">price</q:constant>
         </q:tagName>
         <q:constant datatype="CHARSTRING">15.40</q:constant>
      </q:elementConstructor>
     </q:elementConstructor>
     <q:variable>$s</q:variable>
      </q:insert>
      <q:delete>
     <q:variable>$s</q:variable>
      </q:delete>
    </q:updateList>
  </q:flwu>
</q:query>
```

C.10 Example 16

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:flwu>
    <q:forAssignment variable="$i">
      <q:step axis="DESCENDANT">
     <q:identifier>invoice</q:identifier>
      </q:step>
    </q:forAssignment>
    <q:letAssignment variable="$s">
      <q:step axis="DESCENDANT">
     <q:variable>$i</q:variable>
     <q:predicatedExpr>
       <q:identifier>product</q:identifier>
       <q:predicate>
         <q:function name="EQUALS">
           <q:identifier>name</q:identifier>
           <q:constant datatype="CHARSTRING">screwdriver</q:constant>
         </q:function>
       </q:predicate>
     </q:predicatedExpr>
      </q:step>
    </q:letAssignment>
    <q:forAssignment variable="$a">
      <q:variable>$s</q:variable>
    </q:forAssignment>
    <q:updateList>
```

```
      <q:delete>
     <q:variable>$a</q:variable>
      </q:delete>
      <q:insert target="BEFORE">
     <q:elementConstructor>
       <q:tagName>
         <q:constant datatype="CHARSTRING">product</q:constant>
       </q:tagName>
      <q:elementConstructor>
         <q:tagName>
           <q:constant datatype="CHARSTRING">name</q:constant>
         </q:tagName>
         <q:constant datatype="CHARSTRING">Hammer</q:constant>
      </q:elementConstructor>
      <q:elementConstructor>
         <q:tagName>
           <q:constant datatype="CHARSTRING">price</q:constant>
         </q:tagName>
         <q:function name="TIMES">
           <q:constant datatype="INTEGER">2</q:constant>
           <q:step axis="CHILD">
          <q:variable>$a</q:variable>
          <q:identifier>price</q:identifier>
           </q:step>
         </q:function>
       </q:elementConstructor>
     </q:elementConstructor>
     <q:variable>$a</q:variable>
      </q:insert>
    </q:updateList>
  </q:flwu>
</q:query>
```

List of Literature

[Alp99] Alphaworks IBM, *XMLTreeDiff Update Languages*, http://www.alphaworks.ibm.com/aw.nsf/techmain/xmltreediff, 1999

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J.L. Wiener, *The Lorel Query Language for Semistructured Data*, http://www-db.stanford.edu/pub/papers/lorel96.ps, April 1997

[BFRW01] A. Brown, M. Fuchs, J. Robie, P. Wadler, *MSL - A Model for W3C XML Schema*, http://www.cs.bell.labs.com/who/wadler/papers/msl/mls.pdf, May 2001

[BHL99] W3C Recommendation, *Namespaces in XML*, http://www.w3.org/TR/1999/REC-xml-names-19990114, January 1999

[BM01] W3C Recommendation, *XML Schema Part 2: Datatypes*, http://www.w3.org/TR/2000/REC-xmlschema-2-20010502, May 2001

[Bos97] John Bosak, Sun Microsystems, *XML, Java and the future of the Web*, http://www.ibiblio.org/pub/sun-info/standards/xml/why/xmlapps.htm, Mars 1997

[BPSM00] W3C Recommendation, *Extensible Markup Language (XML) 1.0 (Second Edition)*, http://www.w3.org/TR/2000/REC-xml-20001006, October 2000

[CCF+01] W3C Working Draft, *XQuery 1.0: An XML Query Language*, http://www.w3.org/TR/2001/WD-xquery-20010607, June 2001

[CD99] W3C Recommendation, *XML Path Language (XPath) Version 1.0*, http://www.w3.org/TR/1999/REC-xpath-19991116, November 1999

[CFMR01a] W3C Working Draft, *XML Query Requirements*, http://www.w3.org/TR/2001/WD-xmlquery-req-20010215, February 2001

[CFMR01b] W3C Working Draft, *XML Use Cases*, http://www.w3.org/TR/2001/WD-xmlquery-use.cases-20010608, June 2001

[CT01] W3C Proposed Recommendation, *XML Information Set*, http://www.w3.org/TR/2001/PR-xml-infoset-20010810, August 2001

[Fal01] W3C Recommendation, *XML Schema Part 0: Primer*, http://www.w3.org/TR/2000/REC-xmlschema-0-20010502, May 2001

[FFM+01] W3C Working Draft, *XQuery 1.0 Formal Semantics*, http://www.w3.org/TR/2001/WD-query-semantics-20010607, June 2001

[FM01] W3C Working Draft, *XQuery 1.0 and XPath 2.0 Data Model*, http://www.w3.org/TR/2001/WD-query-datamodel-20010607, June 2001

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Resuable Object-Oriented Software*, Addison Wesley, October 1994

[HM93] G. Hutton, E. Meijer, *Monadic Parser Combinators*, , January 1993

[KPES00] P. Koopman, R. Plasmeijer, M. v. Eekelen, S. Smetsers, *Functional Programming in CLEAN*, Katholieke Universiteit Nijmegen, September 2000

[LM00] A. Laux, L. Martin, *XUpdate - XML Update Language*, http://www.xmldb.org/xupdate/xupdate-wd.html, September 2000

[Mai98] D. Maier, *Database Desiderata for an XML Query Language*, http://www.w3.org/TandS/QL/QL98/pp/maier.html, 1998

[Mar00] L. Martin, *Requirements for XML Update Language*, http://www.xmldb.org/xupdate/xupdate-req.html, November 2000

[May01] W. May, *A Logic-Based Approach to XML Integration*, http://www.informatik.uni-freiburg.de/~may/lopix/report.ps, 2001

[MS00] Metamata and Sun Microsystems, *JavaCC version 2.0*, http://www.webgain.com/products/metamata/java_doc.html, October 2000

[MRS01] W3C Working Draft, *XML Syntax for XQuery 1.0 (XQueryX)*, http://www.w3.org/TR/2001/WD-xqueryx-20010607, June 2001

[OMG01] Object Managment Group (OMG), *Unified Modeling Language (UML) version 1.4*, ftp://ftp.omg.org/pub/docs/ad/01-02-13.pdf, February 2001

[Pan96] S.E. Panitz, *Parsen für Linguisten und andere Programmierlaien,* , January 1996

[TBMM01] W3C Recommendation, *XML Schema Part 1: Structures*, http://www.w3.org/TR/2000/REC-xmlschema-1-20010502, May 2001

[TIHW01] I. Tatarinov, Z.G. Ives, A.Y. Halevy, D.S. Weld, *Updating XML*, http://www.cs.washington.edu/homes/zives/research/updatingXML.pdf, May 2001

[Wal98] Norman Walsh, *A Technical Introduction to XML*, http://www.xml.com/pub/a/98/10/guide0.html, Ocotber 1998